

В. М. Кирюхин, С. М. Окулов

М Е Т О Д И К А РЕШЕНИЯ ЗАДАЧ ПО ИНФОРМАТИКЕ

МЕЖДУНАРОДНЫЕ ОЛИМПИАДЫ



**Москва
БИНОМ. Лаборатория знаний
2007**

Кириухин В. М.

К43 Методика решения задач по информатике. Международные олимпиады / В. М. Кириухин, С. М. Окулов. — М.: БИНОМ. Лаборатория знаний, 2007. — 600 с.: ил.

ISBN 978-5-94774-680-8

Приведен анализ решений задач 18 международных олимпиад по информатике для школьников (1989–2006 гг.). Основной акцент сделан на обсуждении методики решения задач и на самостоятельной подготовке школьников (студентов) при решении задач по олимпиадной информатике.

Книга предназначена для школьников и студентов, увлекающихся информатикой и желающих сделать информатику своей профессией, а также для преподавателей информатики, не ограничивающихся в своей работе рамками стандартных образовательных программ.

УДК 519.85(023)
ББК 32.973-018

Учебное издание

Кириухин Владимир Михайлович, **Окулов** Станислав Михайлович

МЕТОДИКА РЕШЕНИЯ ЗАДАЧ ПО ИНФОРМАТИКЕ. МЕЖДУНАРОДНЫЕ ОЛИМПИАДЫ

Ведущий редактор *Д. Усенков*

Художник *Ф. Инфантэ*

Художественный редактор *О. Лапко*

Компьютерная верстка *Л. Катуркина*

Корректор *Е. Клитина*

Подписано в печать 23.03.07. Формат 60х90 ¹/₁₆.
Гарнитура Школьная. Бумага офсетная. Печать офсетная.
Усл. печ. л. 37,5. Тираж 3000 экз. Заказ 1192

Издательство «БИНОМ. Лаборатория знаний».

Адрес для переписки: 119071, Москва, а/я 32.

Телефон: (499)157-1902. E-mail: Lbz@aha.ru

<http://www.Lbz.ru>

Отпечатано в полиграфической фирме «Полиграфист»
160001, г. Вологда, ул. Челюскинцев, 3.

ISBN 978-5-94774-680-8

© Кириухин В. М., Окулов С. М., 2007
© БИНОМ. Лаборатория знаний, 2007

ПРЕДИСЛОВИЕ

Зададим себе вопрос: как анализировать сложные задачи по информатике? Как объяснять их школьнику? Публикации на эту тему строятся по принципу: делай так-то и так-то, и ты получишь результат. А что предшествовало этому «деланию так» автором (авторами) публикаций, — обычно не раскрывается, остается в тени.

В данной книге на конкретных задачах сделана попытка показать возникновение самого этого процесса «делания», что оказалось гораздо сложнее, чем просто привести описание решения в виде некоего текста программы (пусть даже и с доказательствами математических фактов, если они лежат в основе задачи).

Бытует точка зрения, что «простому» учителю информатики не под силу подготовить школьника к олимпиадам высокого уровня по информатике, что эта работа является уделом «избранных». Мы категорически не согласны с такой позицией, ибо исходим из того, что главным является создание творческой среды на уроке, в школе. Безусловно, учителю необходимо целенаправленно работать над данной проблематикой, но быть «асом» в решении задач совершенно не обязательно! Удивительно, но в основе практически любой сложности лежит простота. Увидеть (найти) эту простоту, идти от этой простоты к решению задачи в процессе совместной деятельности со школьником — это и есть лейтмотив работы учителя. О том, как это делать, мы и попытались написать, выбрав для анализа самый сложный (по статусу) тип олимпиадных задач по информатике.

Перед учителем информатики по-прежнему, как и 1985 году (разве что, может быть, с меньшей — но не намного — остротой), стоят те же самые проблемы: чему учить? как учить? какие учебники использовать? Эти проблемы остались, несмотря на возросшие возможности компьютера, как осталось неизменным и предназначение учителя — развивать школьника в рамках своего предмета так, чтобы он стал Личностью с большой буквы. Остался неизменным и такой критерий оценки деятельности учителя, как успешность выступления его учеников на предметной олимпиаде.

Олимпиады по информатике признаны мировым сообществом как на уровне школьного курса информатики, так и в высшей школе, поэтому речь пойдет именно о них, — точнее, о методике анализа сложных задач по информатике, ибо по статусу международные олимпиады школьников содержат именно такой материал.

Текст формулировок задач в этой книге максимально приближен к исходным вариантам, предлагаемым на олимпиадах. Сокращения и изменения минимальны, что позволяет представить как процесс развития содержательной части задач, так и требований, предъявляемых школьнику.

СТРУКТУРА КНИГИ

Ниже приведен разбор задач всех международных олимпиад школьников по информатике с 1989 по 2006 год. Разбором задачи мы называем ее обсуждение, если материала достаточно для проведения занятия; другими словами, по нему можно, на наш взгляд, «сконструировать» занятие. В противном случае, если материала недостаточно или задача не очень «ложится» для изложения в аудитории, мы просто говорим об идее ее решения.

Заканчивается книга «путеводителем по задачам», в котором приводится классификация задач как по сложности, так и по их тематике.

Профессиональная подготовка школьника к будущей деятельности в области информатики не может ограничиваться рамками образовательного стандарта, даже если рассматривать физико-математический профиль. В целом требуется организовать творческую среду в образовательном учреждении, но эта проблематика выходит за рамки данной книги, и вопросам ее создания посвящена книга одного из авторов*. В настоящей же книге речь идет о *содержательной стороне* этой стадии образования и о *методике проведения занятий*. При этом последний вопрос дан не в отдельных главах, не в виде общих рассуждений о том, «что такое хорошо, что такое плохо», а представлен на конкретном материале обсуждения каждой задачи. Вместо традиционного разбора задач или общего описания схем ее решения мы практически старались дать конспект проведения занятия и даже показать особенности общения («краеугольный камень» профессиональной педагогики) во время его проведения (не говоря опять же лишних слов об этом, ибо все это заложено в самой структуре обсуждения).

Возвращаясь к содержательному аспекту образовательного процесса, можно сказать, что в этой части работы со школьниками он должен включать в себя элективный курс, рассчитанный на несколько лет. Причем по содержанию он должен состоять из небольшого количества тем, изучение которых повторяется каждый раз на новом уровне развития школьника.

* Окулов С. М. Информатика: Развитие интеллекта школьника. М.: БИНОМ. Лаборатория знаний, 2005.

Большинство из этих тем должно носить фундаментальный характер в информатике, и материала для знакомства с ними достаточно в учебной литературе. Перечислим основные из них:

- перебор вариантов и методы его сокращения;
- динамическое программирование;
- сортировка и поиск;
- обработка последовательностей;
- комбинаторика;
- алгоритмы на графах;
- элементы вычислительной геометрии.

В нашем «путеводителе» каждая задача отнесена к одной из этих тем. Кроме того, выделен ряд других тем: «задачи на идею», задачи на технику программирования, «задачи без продолжения». Благодаря этому учитель имеет возможность выбрать задачу, необходимую ему для проведения занятий; школьник или студент в зависимости от изучаемой темы и уровня своего развития (в «путеводителе» дается оценка сложности задач) берет очередную задачу для своей деятельности. Таким образом, «путеводитель» позволяет организовать работу с книгой не на уровне простого знакомства и последовательного чтения, а постоянно, в течение всего процесса изучения элективного курса.

Заметим, что деятельность по работе с задачей не сводится к прочтению ее обсуждения. Она требует анализа алгоритмической части решения, поиска контрпримеров и особых случаев, разработки как основного программного кода, так и дополнительных программ, достаточно полного тестирования, исследования решения. Практика написания какого-либо варианта решения и проверка его на простых тестах — это начальная стадия в длительном процессе становления профессионала.

Благодарности

В первую очередь, хотелось бы выразить признательность нашему соавтору по первому варианту книги* Антону Валерьевичу Лапунову. Алексей Ефремов оказал незаменимую помощь в программной реализации задач олимпиад последних лет. Математический талант Александра Николаевича Семенова позволил «сгенерировать» идеи решения отдельных задач. Студенты Вятского государственного гуманитарного университета — Роман Веснин, Максим Корчемкин, Иван Матанцев, Роман Шарыгин — своим участием в решении отдельных самых сложных задач способствовали выходу в свет этой книги. Ульяна Токмакова выполнила профессиональную подготовку рисунков для издания.

* Кирюхин В. М., Лапунов А. В., Окулов С. М. Задачи по информатике. Международные олимпиады 1989–1996 гг. М.: АБФ, 1996.

ВВЕДЕНИЕ

Идея проведения международных олимпиад по информатике (МОИ) была высказана на 24-й конференции ЮНЕСКО болгарским делегатом, профессором Сендовым в октябре 1987 г. в Париже. В мае 1989 г. ЮНЕСКО организовала и провела первую такую олимпиаду в болгарском городе Правец. Олимпиада сразу привлекла к себе повышенное внимание, уже тогда в ней приняли участие 13 стран — гораздо больше, чем стран-участниц первых международных олимпиад по математике, физике и другим наукам.

Если на первых международных олимпиадах золотые медали наших ребят рассматривались скорее как случайность, то начиная с 1993 г. российские школьники считаются постоянными фаворитами. Подтверждением является тот факт, что кубок чемпиона мира по информатике пять раз побывал в нашей стране и до 2005 г. по этому показателю России не было равных! До сих пор держится рекорд российской команды, которая в 2000 г. в Пекине завоевала четыре золотые медали и первое место в неофициальном командном зачете, а Михаил Баутин стал чемпионом мира. В общей же сложности без медалей российские школьники ни разу не приезжали домой с 1991 г. А по общему количеству медалей наша команда уступает только две медали китайским школьникам, оставив далеко позади представителей других стран. (В табл. 2 и 3 перечислены чемпионы прошедших международных олимпиад по информатике и все советские и российские школьники, завоевавшие там золотые медали.)

Официальный сайт международной олимпиады по информатике — <http://www.ioinformatics.org/>. Официальный сайт секретариата международной олимпиады по информатике — <http://olympiads.win.tue.nl/ioi/>.

ЦЕЛИ МЕЖДУНАРОДНОЙ ОЛИМПИАДЫ ПО ИНФОРМАТИКЕ

Международная олимпиада по информатике проводится каждый год со следующими целями:

- выявление и объединение талантливых школьников из разных стран мира в области информатики, проведение соревнований для них и создание условий для их дальнейшего развития;
- объединение педагогов, ученых и общественных деятелей для обмена опытом в работе с талантливыми детьми из разных стран мира в области информатики, создание дружеских отношений между ними;

- привлечение внимания молодежи из разных стран мира к изучению информатики, их ранняя профориентация;
- популяризация и повышение интереса к соревнованиям по информатике среди школьников в различных странах мира;
- привлечение стран к организации международных олимпиад по информатике.

ИСТОРИЯ МЕЖДУНАРОДНОЙ ОЛИМПИАДЫ ПО ИНФОРМАТИКЕ

Начиная с 1989 г. международная олимпиада по информатике проводится ежегодно в различных странах. По своей популярности она не уступает аналогичной олимпиаде по математике, и количество участвующих в ней стран увеличивается год от года. Сейчас эта цифра уже приближается к восьмидесяти.

Страна и место проведения очередной олимпиады определяется международным олимпиадным комитетом за три года до ее начала. Чтобы принять международную олимпиаду по информатике, стране-кандидату нужно пройти достаточно сложную процедуру конкурсного отбора, поскольку желающих принять очередную олимпиаду не так уж мало!

Международные олимпиады по информатике являются важной составляющей для выявления и развития молодых дарований в мире в области информатики, что подтверждается неуклонным ростом числа стран, принимающих в ней участие (см. табл. 1). По своей популярности в мире эта олимпиада успешно конкурирует с математической, а по объему привлекаемых средств и «мощности» спонсоров уже ее превосходит. Это и понятно: именно в руках специалистов в области информационных технологий находится будущее нашей планеты в XXI веке, в котором информация является стратегическим ресурсом, а технологии ее обработки определяют экономический потенциал страны и ее место на мировой арене.

ПОРЯДОК УЧАСТИЯ СТРАН В МЕЖДУНАРОДНОЙ ОЛИМПИАДЕ ПО ИНФОРМАТИКЕ

В международной олимпиаде по информатике (МОИ) текущего года могут принимать участие только те страны, которые уже участвовали в олимпиаде в предыдущем году, или наблюдатели которых присутствовали на ней. Страны, выразившие желание участвовать в текущем году в МОИ впервые или потерявшие право участвовать в олимпиаде по какой-либо причине, должны направить в международный олимпиадный комитет по

Таблица 1

**Страны-участницы и время проведения международных
олимпиад по информатике (1989–2006 гг.)**

	Год	Город, страна	Количество стран- участниц	Количество участников
	1989	Праветц (Pravetz), Болгария, 16–19 мая	13	46
	1990	Минск, СССР, 15–21 июля	25	94
	1991	Афины, Греция, 19–25 мая	26	56
	1992	Бонн, Германия, 11–21 июля	51	142
	1993	Мендоза, Аргентина, 16–25 октября	43	151
	1994	Ханинге (Haninge), Швеция, 3–10 июля	49	188
	1995	Эйндховен (Eindhoven), Нидерланды, 26 июня–3 июля	51	260
	1996	Весзпрем (Veszprém), Венгрия, 25 июля–2 августа	58	215
	1997	Кейптаун, Южная Африка, 30 ноября–7 декабря	63	229
	1998	Сетубал (Setúbal), Португалия, 5–12 сентября	66	238
	1999	Анталия-Белек, Турция, 9–16 октября	64	254
	2000	Пекин, Китай, 23–30 сентября	71	272
	2001	Тампере, Финляндия, 14–21 июля	74	272
	2002	Йотг-Ин (Yong-In), Республика Корея, 18–25 августа	77	267
	2003	Кеноша (Kenosha), штат Висконсин (Wisconsin), США, 16–23 августа	75	269
	2004	Афины, Греция, 11–18 сентября	80	296
	2005	Новы Сонч (Nowy Sacz), Польша, 18–25 августа	72	276
	2006	Мерида, штат Ютакан, Мексика, 13–20 августа	75	284

информатике заявку на участие и двоих своих представителей в качестве наблюдателей, после чего они получают право участвовать в полном формате в олимпиаде следующего года.

В состав делегации от каждой страны-участницы МОИ могут входить руководитель команды, его заместитель и не более четырех учащихся, которым ко времени начала олимпиады исполнилось не более 20 лет и которые в период с сентября по декабрь предшествовавшего олимпиаде года являлись учащимися средних общеобразовательных учебных заведений.

Руководитель команды и его заместитель должны быть преподавателями или профессорами в своей стране и иметь опыт участия в предыдущих международных олимпиадах по информатике. В состав делегации могут также входить дополнительные лица: тренеры участников, консультанты и специалисты в области информатики, а также воспитатели и другой обслуживающий персонал.

РУКОВОДЯЩИЕ ОРГАНЫ МЕЖДУНАРОДНОЙ ОЛИМПИАДЫ ПО ИНФОРМАТИКЕ

Главным руководящим органом МОИ является *международное жюри (General Assembly)*, куда входят все руководители команд-участниц олимпиады. Для выполнения текущей работы международное жюри избирает:

- Президента олимпиады;
- Исполнительного директора;
- международный комитет (International Committee);
- международный научный комитет (International Scientific Committee);
- международную техническую рабочую группу (International Technical Working Group).

Президент МОИ осуществляет непосредственное руководство международным олимпиадным движением по информатике. Он избирается международным жюри на три года. В настоящее время Президентом МОИ является представитель Китая, доктор Zide Du.

Исполнительный директор МОИ выполняет административные функции, обеспечивает финансовую поддержку олимпиады и ее дальнейшее развитие. В настоящее время Исполнительным директором МОИ является представитель Финляндии, доктор Jari Koivisto.

Международный комитет выполняет всю работу, связанную с организацией и проведением олимпиад. В состав международного комитета входят:

- Президент МОИ;
- представитель страны-организатора олимпиады предыдущего года;
- представитель страны-организатора текущей олимпиады;
- три представителя стран-организаторов трех последующих олимпиад;
- пять избираемых на трехлетний период представителей стран-участниц олимпиады;
- Исполнительный директор (без права голоса).

В период с 1990 по 1993 г. и с 2000 по 2003 г. членом международного комитета МОИ избирался представитель России, В. М. Кириухин. Благодаря этому многие идеи развития олимпиадного движения по информатике, характерные для России, были привнесены и в развитие международного олимпиадного движения, и наоборот.

Международный научный комитет отвечает за научную сторону содержания олимпиадных заданий и за порядок проведения соревнований. В состав международного научного комитета входят:

- представитель страны-организатора олимпиады предыдущего года;
- представитель страны-организатора текущей олимпиады;
- представитель страны-организатора последующей олимпиады;
- три избираемых на трехлетний период представителей стран-участниц олимпиады;
- председатель международной технической рабочей группы.

Международная техническая рабочая группа обеспечивает работу проверяющей системы во время проведения соревнований. Необходимость в этом органе МОИ возникла с появлением автоматизированной системы проверки решений участников олимпиады. В ведении данного органа находятся также все вопросы дальнейшего совершенствования и развития этой системы. Председатель международной технической рабочей группы избирается международным жюри сроком на три года. Состав этой группы формируется страной-организатором текущей олимпиады.

Всю работу по организации и проведению текущей олимпиады осуществляет организационный комитет олимпиады. Работу по непосредственной подготовке олимпиадных заданий выполняет научный комитет, в состав которого входят представители страны-организатора текущей олимпиады.

ПРАВИЛА ПРОВЕДЕНИЯ СОРЕВНОВАНИЯ

В настоящее время международная олимпиада по информатике проводится в два тура; длительность каждого тура — 5 часов. На каждом туре участникам предлагается три задачи.

Перед началом соревнований всем участникам олимпиады предоставляется возможность ознакомиться со своими рабочими местами и опробовать в реальном режиме работу программного обеспечения, предназначенного для проведения соревнований. С этой целью в программе олимпиады предусмотрен двухчасовой пробный тур. Задачи для пробного тура заранее размещаются на сайте организаторов олимпиады, их можно решить до начала пробного тура и принести с собой на компакт-дисках. Участие в пробном туре для всех участников является обязательным.

Начиная с 2007 г. во время соревнований на всех компьютерах участников устанавливается программная среда GNU/Linux. В отличие от прошлых олимпиад, Microsoft Windows на компьютерах участников устанавливаться не будет.

На всех компьютерах участников, как правило, устанавливается следующее программное обеспечение:

- KDE;
- WindowMaker;
- Web-браузеры (Firefox, Konqueror);
- редакторы (mcedit, joe, vim, kate, kwrite, kdevelop, emacs, rhide);
- компиляторы (gcc 3.3, g++ 3.3, Free Pascal 2.0.2 с документацией);
- отладчики (gdb, ddd);
- документация STL.

Официальным языком на МОИ является английский. В связи с этим, накануне каждого тура, после утверждения текстов олимпиадных заданий международным жюри, руководители команд осуществляют перевод всех текстов заданий с английского на родной язык. На это время все контакты руководителей команд и участников соревнований прекращаются; более того, все мобильные телефоны и средства связи у участников соревнований изымаются и возвращаются им после окончания соответствующего тура.

Процедура перевода заданий является очень ответственным этапом во время проведения олимпиады и требует от руководителей команд отличного владения английским языком и высокого уровня подготовки в области информатики. Мало того, что надо правильно перевести текст заданий, чтобы у участников было однозначное понимание, что от них требуется во время решения задания, так еще в процессе перевода возникает боль-

шое количество вопросов к международному научному комитету по правильности формулировок в заданиях, которые часто приводят к их корректировке и повторному утверждению международным жюри!

Все участники соревнований перед началом тура получают тексты олимпиадных заданий на английском и на родном языке. В течение одного часа с момента начала тура каждый участник олимпиады имеет право задать на родном языке вопросы по условиям заданий членам международного научного комитета. Вопросы должны задаваться в письменном виде и формулироваться так, чтобы ответ был либо «да», либо «нет». Тексты вопросов переводятся руководителями команд с родного языка на английский и передаются в научный комитет. Ответы международного научного комитета на вопросы участников могут быть следующими: «да», «нет» или «без комментариев».

Участникам олимпиады категорически запрещается во время туров пользоваться личными компьютерами, калькуляторами, электронными записными книжками, средствами связи (пейджерами, мобильными телефонами и т. д.), принесенными с собой электронными носителями информации (дискетами, CD- и DVD-дисками, модулями флэш-памяти и пр.), а также учебной литературой и заранее заготовленными личными записями. Разрешается приносить с собой только словари для перевода с английского языка на родной и обратно.

В решениях задач участникам запрещается использовать какие-либо средства, нарушающие работу проверяющей системы.

Во время тура участники олимпиады имеют право общаться только с представителями оргкомитета, находящимися в помещении, где проводится соревнование. В случае возникновения не по вине участника во время тура сбоев в работе компьютера или используемого программного обеспечения время, затраченное на восстановление работоспособности компьютера, может быть компенсировано. По истечении времени тура участникам олимпиады запрещается выполнять какие-либо действия на компьютере.

ХАРАКТЕРИСТИКА ОЛИМПИАДНЫХ ЗАДАНИЙ

При выборе олимпиадных заданий для МОИ международный научный комитет, в соответствии с существующими правилами, руководствуется следующими соображениями:

- задачи должны иметь алгоритмический характер;
- решения задач должны быть реализуемыми с использованием алгоритмических языков Pascal, C или C++;

- задачи должны обладать свойством *вариативности*, т. е. решение задачи может быть получено с использованием различных алгоритмов, отличающихся по сложности реализации и по эффективности.

Большое значение при проверке решений олимпиадных заданий играет *эффективность* используемых в них алгоритмов. Оценка эффективности осуществляется в процессе тестирования путем использования набора тестов, специальным образом разработанного научным комитетом.

Задачи МОИ можно разбить на три типа.

К *задачам первого типа* относятся стандартные задачи («Batch tasks»), решением которых является программа, формирующая выходной файл по заданному входному файлу.

Задачи второго типа являются интерактивными («Reactive tasks»). Решением таких задач также является программа, однако, в отличие от задач первого типа, вместо чтения исходных данных из входного файла и записи результата в выходной файл эта программа должна обмениваться данными с другой программой, определенной в условии задачи.

В *задачах третьего типа* («Output tasks») решением является уже не программа, как в задачах первого и второго типов, а файлы выходных данных, соответствующие заданным в условии задачи входным файлам.

Решение задачи, которое участник олимпиады сдает на проверку, должно удовлетворять следующим ограничениям:

- объем файла с исходным текстом программы не должен превышать 100 кбайт;
- время компиляции программы не должно превышать 30 секунд;
- максимальное время работы программы на одном тесте не должно превышать величины, указанной в условии задачи;
- максимальный объем памяти, используемой в процессе работы программы, не должен превышать величину, указанную в условии задачи.

При нарушении двух первых ограничений задание на проверку не принимается, а участник получает за эту задачу нуль баллов. При нарушении двух других ограничений при работе программы на каком-либо тесте баллы не начисляются за прохождение этого теста.

Вместе с текстами олимпиадных задач перед началом тура каждый участник получает таблицу с информацией о том, как должны быть представлены результаты решения каждой задачи, включая имена рабочих каталогов, входных и выходных

шое количество вопросов к международному научному комитету по правильности формулировок в заданиях, которые часто приводят к их корректировке и повторному утверждению международным жюри!

Все участники соревнований перед началом тура получают тексты олимпиадных заданий на английском и на родном языке. В течение одного часа с момента начала тура каждый участник олимпиады имеет право задать на родном языке вопросы по условиям заданий членам международного научного комитета. Вопросы должны задаваться в письменном виде и формулироваться так, чтобы ответ был либо «да», либо «нет». Тексты вопросов переводятся руководителями команд с родного языка на английский и передаются в научный комитет. Ответы международного научного комитета на вопросы участников могут быть следующими: «да», «нет» или «без комментариев».

Участникам олимпиады категорически запрещается во время туров пользоваться личными компьютерами, калькуляторами, электронными записными книжками, средствами связи (пейджерами, мобильными телефонами и т. д.), принесенными с собой электронными носителями информации (дискетами, CD- и DVD-дисками, модулями флэш-памяти и пр.), а также учебной литературой и заранее заготовленными личными записями. Разрешается приносить с собой только словари для перевода с английского языка на родной и обратно.

В решениях задач участникам запрещается использовать какие-либо средства, нарушающие работу проверяющей системы.

Во время тура участники олимпиады имеют право общаться только с представителями оргкомитета, находящимися в помещении, где проводится соревнование. В случае возникновения не по вине участника во время тура сбоев в работе компьютера или используемого программного обеспечения время, затраченное на восстановление работоспособности компьютера, может быть компенсировано. По истечении времени тура участникам олимпиады запрещается выполнять какие-либо действия на компьютере.

ХАРАКТЕРИСТИКА ОЛИМПИАДНЫХ ЗАДАНИЙ

При выборе олимпиадных заданий для МОИ международный научный комитет, в соответствии с существующими правилами, руководствуется следующими соображениями:

- задачи должны иметь алгоритмический характер;
- решения задач должны быть реализуемыми с использованием алгоритмических языков Pascal, C или C++;

- задачи должны обладать свойством *вариативности*, т. е. решение задачи может быть получено с использованием различных алгоритмов, отличающихся по сложности реализации и по эффективности.

Большое значение при проверке решений олимпиадных заданий играет *эффективность* используемых в них алгоритмов. Оценка эффективности осуществляется в процессе тестирования путем использования набора тестов, специальным образом разработанного научным комитетом.

Задачи МОИ можно разбить на три типа.

К *задачам первого типа* относятся стандартные задачи («Batch tasks»), решением которых является программа, формирующая выходной файл по заданному входному файлу.

Задачи второго типа являются интерактивными («Reactive tasks»). Решением таких задач также является программа, однако, в отличие от задач первого типа, вместо чтения исходных данных из входного файла и записи результата в выходной файл эта программа должна обмениваться данными с другой программой, определенной в условии задачи.

В *задачах третьего типа* («Output tasks») решением является уже не программа, как в задачах первого и второго типов, а файлы выходных данных, соответствующие заданным в условии задачи входным файлам.

Решение задачи, которое участник олимпиады сдает на проверку, должно удовлетворять следующим ограничениям:

- объем файла с исходным текстом программы не должен превышать 100 кбайт;
- время компиляции программы не должно превышать 30 секунд;
- максимальное время работы программы на одном тесте не должно превышать величины, указанной в условии задачи;
- максимальный объем памяти, используемой в процессе работы программы, не должен превышать величину, указанную в условии задачи.

При нарушении двух первых ограничений задание на проверку не принимается, а участник получает за эту задачу нуль баллов. При нарушении двух других ограничений при работе программы на каком-либо тесте баллы не начисляются за прохождение этого теста.

Вместе с текстами олимпиадных задач перед началом тура каждый участник получает таблицу с информацией о том, как должны быть представлены результаты решения каждой задачи, включая имена рабочих каталогов, входных и выходных

файлов, опции компилятора, заголовки программ при использовании различных языков программирования и правила приема решения на проверку.

Во всех заданиях используемые входные и выходные файлы состоят из одной или нескольких строк, каждая из которых заканчивается символом конца строки. Условия формирования каждой строки определены в тексте описания задачи. Все исходные данные во входных файлах являются корректными, если это специально не оговорено в условии задачи.

Олимпиадные задания, предлагаемые на МОИ, постоянно развиваются и совершенствуются. Для олимпиады, прошедшей в 2006 г. в Мексике, последние новшества были следующими:

- для задач стандартного типа программы, являющиеся решениями, должны были читать входные данные из текстового файла и писать выходные данные в текстовый файл;
- для интерактивных задач взаимодействие с другими программами осуществлялось с использованием стандартного входа и стандартного выхода;
- участники имели возможность посылать свои решения на проверку тестирующей системе неограниченное количество раз;
- для некоторых заданий был предусмотрен неоднократный прогон тестов, а итоговое количество баллов в этом случае определялось по наихудшему результату.

Ответственность за формирование пакета олимпиадных задач несет международный научный комитет. Представители всех стран — участниц олимпиады могут до конца установленного срока предложить свои рекомендуемые задачи, которые могут быть приняты для дальнейшей проработки или отвергнуты. Основная работа по окончательной доработке задач проводится научным комитетом страны-организатора олимпиады под контролем международного научного комитета.

Основными критериями отбора олимпиадных задач являются следующие показатели:

- оригинальная формулировка задачи или оригинальная идея ее решения;
- в тексте условия задачи не должны встречаться термины и понятия, выходящие за пределы среднего образования в большинстве стран мира; в крайних случаях, они должны быть определены или конкретизированы;
- задача должна быть однозначно определена, т. е. в ее формулировке не должно быть неоднозначностей;
- задача не должна требовать для своего решения специальных знаний;

- формулировка задачи должна предполагать наличие этапа формализации при ее решении, т. е. перехода от неформальной постановки задачи к формальной;
- задача должна быть разумной сложности и трудоемкости.

СОДЕРЖАНИЕ ОЛИМПИАДНЫХ ЗАДАНИЙ

В отличие от международных олимпиад по физике, химии и биологии, содержание олимпиад по информатике официально не определено в каких-либо документах МОИ. Такая же ситуация сложилась и в других международных олимпиадах (по математике, географии, астрономии и лингвистике). На это есть ряд причин, и в первую очередь это связано с тем, что нужно дать возможность олимпиадным заданиям развиваться (поскольку любая программа является консервативной, а вносить каждый раз в нее изменения достаточно сложно из-за вообще сложностей в процедурах внесения каких-либо изменений в официальные документы любой международной олимпиады).

Опыт проведения международных олимпиад по информатике показывает, что содержание и основные знания, умения и навыки, востребованные при решении олимпиадных заданий международного уровня, строятся на основе сформированного уровня освоения школьниками содержания информатики и информационных технологий. В частности, можно отметить следующие компетентностные качества, которые выявляются в результате участия школьников в МОИ:

- углубленные знания в сфере математических основ информатики;
- развитые умения и навыки в теории алгоритмов;
- устойчивые знания об информационных процессах, типах информации, способах представления и передачи информации;
- устойчивые практические навыки самостоятельного построения компьютерных моделей;
- расширенные представления о составе компьютера, программном принципе работы компьютера и прикладном программном обеспечении;
- беглое владение клавиатурным вводом на русском и английском языке;
- свободное владение компьютерным графическим интерфейсом;
- свободные навыки работы с компьютером: с файловой системой, операционной системой, архивирующими, конвертирующими средствами, программными приложениями, поисковой системой;

- устойчивые навыки работы с общими и персональными ресурсами в локальной компьютерной сети;
- устойчивые навыки работы со специализированным ПО (среды программирования, трансляторы, отладчики);
- свободные навыки работы с инструментальными средствами глобальной компьютерной сети (регистрация, передача данных, защита информации);
- развитое чувство самоконтроля и ответственности;
- навыки самостоятельного планирования заданий;
- проявление волевых качеств.

Важной особенностью задач МОИ является их ориентация на проверку развития у обучаемых теоретического мышления, логики, а также творческих способностей и интуиции. В отличие от задач студенческих олимпиад по информатике, которые, помимо всего прочего, направлены еще и на проверку обученности участников олимпиады, задачи МОИ предоставляют участникам возможность без специальных знаний решать нестандартные и новые для них задачи. Каждая задача должна позволять участникам сделать для себя небольшое открытие и в полной мере раскрыть имеющийся у них творческий потенциал.

Олимпиадные задачи МОИ отличаются значительным тематическим разнообразием. Из опыта проведения международных олимпиад в прошлые годы можно выделить следующие разделы информатики, характерные для решения многих задач, предлагавшихся на международных олимпиадах по информатике:

- комбинаторика;
- сортировка и поиск;
- обработка последовательностей;
- алгоритмы на графах;
- элементы вычислительной геометрии;
- перебор вариантов и методы его сокращения;
- динамическое программирование.

Олимпиадные задания МОИ гармонично отражают не только теоретические разделы курса информатики, но и основные этапы решения задач с использованием компьютеров:

- формализацию задачи;
- выбор формального метода и разработку алгоритма решения задачи, включая оценку правильности и сложности алгоритма;
- программирование алгоритма и отладку программы; тестирование полученной программы.

Выполнение олимпиадных заданий МОИ предъявляет к участникам этих олимпиад ряд специальных требований. Так, участники должны:

- иметь необходимые навыки работы с персональными компьютерами, знать основные характеристики компьютеров (в частности, каким образом эти характеристики влияют на решаемые задачи и как обходить имеющиеся ограничения, например, по памяти и быстродействию);
- знать в необходимом объеме и уметь работать при решении прикладных задач с широко распространенными операционными системами, в том числе и с Linux;
- знать и уметь использовать на практике методы формализации поставленных задач; в частности, правильно переходить от словесного описания постановки задачи к ее формальному описанию с учетом заданных технических и содержательных ограничений;
- знать и уметь использовать при решении задач основные типы алгоритмов (алгоритмы работы с различными структурами данных, базовые алгоритмы на графах, алгоритмы организации перебора и динамического программирования, целочисленной арифметики, моделирования, компьютерной геометрии и др.);
- уметь использовать понятие сложности алгоритмов при решении задач, чтобы в используемых или разрабатываемых алгоритмах их сложность находилась в оптимальном соответствии с заданными в условии задачи ограничениями;
- знать и уметь работать с одной или несколькими интегрированными системами программирования при решении олимпиадных задач (состав языков программирования определяется организаторами конкретных олимпиад; например, для заключительного этапа всероссийской олимпиады это C/C++, Pascal, Visual Basic);
- знать методы и уметь использовать на практике современные технологии программирования, а также отладки и тестирования программ.

Очевидно, что чем выше уровень олимпиады, тем сложнее предлагаемые задачи, а значит, больший уровень знаний и умений требуется от участников.

СИСТЕМА ОЦЕНИВАНИЯ РЕШЕНИЙ УЧАСТНИКОВ

Каждая из предложенных на олимпиаде задач оценивается до 100 баллов. Максимальное число баллов за каждый тур составляет 300, а каждый участник олимпиады по результатам решения задач двух туров может в итоге набрать 600 баллов. Важно отметить, что решение каждой из задач предполагает использование широкого спектра алгоритмов, и оценки зависят

только от того, насколько эффективным было предложенное участником решение.

Этап проверки решений участников является очень ответственным и трудным в проведении соревнований на МОИ. За прошедшее время с первой международной олимпиады по информатике многое изменилось в технологии тестирования программ участников. Раньше процесс проверки решений требовал привлечения большого количества специалистов, поскольку каждая задача тестировалась в присутствии участника и каждый тест при проверке решения запускался вручную членом международного научного комитета. Все это накладывало определенные ограничения на комплект тестов и систему оценки для каждой задачи, так как сложно было в достаточном объеме протестировать задачу ввиду временных ограничений на процесс тестирования.

С появлением компьютерных сетей технология проверки решений на МОИ немного улучшилась, поскольку не надо при тестировании загружать программы участников в проверяющий компьютер с дискеты. Дискеты теперь использовались лишь для резервного хранения решений на случай, если произойдут какие-либо сбои в работе сервера или компьютерной сети. Однако все равно приходилось каждый тест запускать вручную.

Стремительное развитие информационных технологий привнесло много нового и в процесс проверки решений участников. Начало его автоматизации было положено в 1995 г. на международной олимпиаде по информатике в Нидерландах. С тех пор процесс проверки решений стремительно совершенствовался, и сейчас решение каждой задачи проверяется в автоматическом режиме с помощью *автоматизированной проверяющей системы*.

Автоматизированная проверяющая система функционирует в рамках специализированной программной системы проведения соревнований, которая обеспечивает в течение тура выполнение следующих функций:

- пересылку решений участников на проверку проверяющей системой;
- получение результатов собственного тестирования, когда каждый участник может послать на тестирование свою программу с разработанным им набором тестов;
- сохранение файлов и их восстановление;
- распечатку нужных файлов.

Ответственность за сохранение и восстановление файлов во время тура целиком и полностью лежит на участниках. Учитывая необходимость обеспечения высокой надежности и безопасности работы программного обеспечения, используемого участ-

никами во время туров, в правила проведения соревнований включены очень жесткие требования. В частности, во время туров категорически запрещается:

- использовать в любой форме доступ к сети;
- использовать системные вызовы;
- читать и создавать файлы, не указанные в тексте задач;
- использовать любые попытки воздействия на систему безопасности сети и систему оценки решений участников;
- запускать на исполнение программы, не указанные в тексте задач;
- изменять настройки файловой системы;
- читать информацию в файловой системе.

Кроме того, в правила соревнований в последние годы введены следующие ограничения:

- запрещается использование дискет во время туров;
- требуется архивировать входные данные, используемые участниками во время туров для проверки своих решений с использованием проверяющей системы;
- использование проверяющей системы для проверки решений прекращается за 30 минут до окончания тура;
- участники имеют возможность послать свои решения на проверку проверяющей системой ограниченное количество раз;
- установлены ограничения для каждого участника на количество тестирований программ во время тура с использованием проверяющей системы;
- введены ограничения на количество использований принтера во время тура;
- в распоряжение участников не предоставляются средства для подсчета времени исполнения программы на сервере проверяющей системы.

Чтобы решение было принято на проверку проверяющей системой, необходимо сначала послать его на проверку на сервер системы, а затем — чтобы решение прошло тестирование на тесте, приведенном в условии задачи. При отправке решения на проверку участник олимпиады должен в соответствующем диалоговом окне проверяющей системы задать необходимые данные (имя задачи; язык программирования или конкретный компилятор, поскольку для языков Pascal и C в данный момент поддерживается несколько компиляторов; имя файла, содержащего исходный текст решения). После реакции системы на отправленное решение участник может просмотреть результат: выполнены ли тесты, которые указаны в условии задачи, и принята ли задача на дальнейшую проверку.

Процесс обработки переданного на проверку файла с исходным кодом программы успешно завершается, если выполняются все требования, предъявляемые к соответствующему заданию. Одна часть из этих требований является общей для всех заданий, другая — описывается в тексте каждой конкретной задачи.

Если результатом решения задачи были выходные файлы, то процесс передачи их на проверку несколько отличается от описанной процедуры принятия на проверку решения в виде исполняемого файла. В этом случае перед тем, как послать выходной файл на проверку, необходимо поместить в соответствующем окне проверяющей системы некоторую дополнительную информацию; предварительного тестирования здесь не осуществляется, проверяется лишь соответствие формата передаваемых на проверку выходных файлов формату, описанному в условии.

После окончания тура все принятые системой на проверку участники проходят тестирование в автоматическом режиме. При проверке решений задач используются различные методы их проверки и оценки. По результатам проверки системой формируется протокол, позволяющий определить для участников олимпиады правильность выполненного ими решения задач или типы ошибок, выявленных системой в процессе тестирования решения.

После объявления предварительных результатов каждого тура специализированная программная система предоставляет каждому участнику возможность доступа:

- к сданным на проверку решениям;
- к наборам тестов, используемых при проверке решений участников;
- к результатам проверки решений.

Если сам процесс проверки решений может быть полностью автоматизирован, то процесс разработки самих тестов от этого не упростился, а наоборот, усложнился. Конечно, об использовании на олимпиадах по информатике полной системы тестов для каждой задачи говорить не приходится, но можно разработать тесты, которые полностью проверяют логику возможных алгоритмов, используемых при ее решении.

Важно отметить, что, помимо автоматизации процесса проверки решений, такие системы позволили более тщательно проверять работоспособность программ участников. Если раньше количество тестов для каждой задачи было ограничено технологическими возможностями, то сейчас таких ограничений практически нет, и при тестировании используется то количе-

ство тестов, которое международный научный комитет считает нужным.

ОПРЕДЕЛЕНИЕ ПОБЕДИТЕЛЕЙ ОЛИМПИАДЫ

В соответствии с существующими правилами проведения МОИ количество победителей и призеров олимпиады, награжденных золотыми, серебряными и бронзовыми медалями, составляет 50% от общего числа участников. Количество золотых, серебряных и бронзовых медалей при этом соответствует отношению 1 : 2 : 3.

При определении победителей и призеров олимпиады международное жюри руководствуется только количеством набранных баллов. При подведении итогов члены жюри имеют только таблицу с цифрами, показывающими итоговые баллы и количество участников, их набравших. На основании этих данных определяются границы, распределяющие медали среди победителей и призеров олимпиады. Понятно, что не всегда удастся строго соблюдать соотношение 1 : 2 : 3 при распределении медалей, поэтому незначительное отклонение от названного соотношения допускается соответствующим решением международного жюри.

Помимо награждения медалями, каждый победитель и призер олимпиады получает сертификат, подтверждающий факт получения медали соответствующего достоинства. Кроме того, окончательные результаты олимпиады размещаются на официальных сайтах МОИ.

ПОРЯДОК ОТБОРА СБОРНОЙ КОМАНДЫ СТРАНЫ НА МЕЖДУНАРОДНУЮ ОЛИМПИАДУ ПО ИНФОРМАТИКЕ

Несмотря на то, что в содержательном плане всероссийские и международные олимпиады не противоречат друг другу, условия проведения и требования к участникам международных олимпиад по информатике несколько отличаются. В связи с этим для успешной подготовки к таким соревнованиям два раза в год, зимой (в январе — феврале) и летом (в конце июня — начале июля) Рособразованием проводятся специальные учебно-тренировочные сборы для кандидатов в сборную команду России по информатике.

Подобные мероприятия, с одной стороны, имеют характер отборочных; в частности, именно по результатам летних сборов формируется состав сборной команды России для участия в

международных олимпиадах. С другой стороны, они также носят тренировочный характер, особенно для школьников невыпускных классов. Поэтому на сборы всегда приглашаются не только те, кто реально претендует на место в сборной команде России, но и школьники младших классов, которые успели себя проявить на всероссийской олимпиаде, чтобы в более раннем возрасте приобщить их к олимпиадному движению и создать для них все необходимые условия для дальнейшего роста.

За прошедшие с начала первой олимпиады по информатике годы удалось сформировать хорошо зарекомендовавшую себя на практике систему подготовки к международным олимпиадам. Наряду с изучением многих теоретических вопросов информатики, необходимых для решения олимпиадных задач, освоением навыков работы с используемым на международных олимпиадах программным обеспечением и с решением большого количества задач, соответствующих уровню международных олимпиад и в условиях, максимально приближенных к соревнованиям этого уровня, на сборах всегда формируется неповторимая атмосфера единения близких по духу людей. Достигается это благодаря тому, что вместе с опытными педагогами в сборах всегда принимают участие многие бывшие победители и призеры международных и национальных олимпиад. Их бесценный опыт участия в этих олимпиадах и в решении олимпиадных задач оказывается особенно полезным для следующего поколения «сборников», а небольшая разница в возрасте позволяет всем участникам сборов чувствовать себя членами одной большой олимпиадной семьи.

В процессе проведения учебно-тренировочных сборов постоянно ведется мониторинг успешности всех участников сборов. Те четыре школьника, которые к концу сборов имеют максимальный рейтинг, включаются в состав сборной команды России и представляют нашу страну на МОИ.

Таблица 2

**Чемпионы международных олимпиад по информатике
(1989–2006 гг.)**

Год, страна	Фамилии и имена чемпионов	Страна
1989 г., Правец, Болгария	Teodor Tonchev	Болгария
1990 г., Минск, СССР	Tsvetomir Petrov	Болгария
1991 г., Афины, Греция	Igor Maly	Чехословакия

Год, страна	Фамилии и имена чемпионов	Страна
1992 г., Бонн, Германия	Pinit Asavanuchit Nathan Bronson Gao Chen Jitat Fagcharoenphon Huss Fredrik BomJun Kim Viet NguyenTuan Matej Ondrusek Peter Laszlo Smith Shawn Wu Xing Yang Yunhe	Таиланд США Китай Таиланд Швеция Корея Вьетнам Чехословакия Венгрия США Китай Китай
1993 г., Мендоза, Аргентина	Martin Mares Mehdi Foadgar Radu-Lucian Lupsa Fedric Huss	Чехия Иран Румыния Швеция
1994 г., Ханinge, Швеция	Виктор Баргачев	Россия
1995 г., Эйндховен, Нидерланды	Виктор Баргачев	Россия
1996 г., Веспрем, Венгрия	Daniel Kral	Чехия
1997 г., Кейптаун, ЮАР	Владимир Мартьянов	Россия
1998 г., Сетубал, Португалия	Мартьянов Владимир Zhunping Zhang Mihal Stroe Daniel Mark	Россия Китай Румыния ЮАР
1999 г., Анталия, Турция	Chen Hong	Китай
2000 г., Пекин, Китай	Михаил Баутин	Россия
2001 г., Тампере, Финляндия	Reid Barton	США
2002 г., Сеул, Южная Корея	Wanyeong Jung	Южная Корея
2003 г. Чикаго, США	Hwan-Seung Yeo	Южная Корея
2004 г., Афины, Греция	Paul Jefferys	Великобритания
2005 г., Новы-Сонч, Польша	Weidong Hu Eric Price Yuan Zhou Юрий Зновяк	Китай США Китай Украина
2006 г., Мерида, Мексика	Filip Wolski	Польша

Таблица 3

**Российские (советские) золотые призы
международных олимпиад по информатике (1989–2006 гг.)**

Страна, год	ФИО победителей	Город
1989 г., 16–19 мая Правец, Болгария	Чяпайтис А.	Каунас
1990 г., 12–21 июля Минск, СССР	Козлов Дмитрий Датуашвили Георгий	Тбилиси Ленинград
1993 г., 16–25 октября, Мендоза, Аргентина	Миронов Илья	Санкт-Петербург
1994 г., 3–10 июля, Ханинге, Швеция	Баргачев Виктор Елизаров Роман Лапунов Антон	Санкт-Петербург Санкт-Петербург Киров
1995 г., 26 июня–3 июля Эйндховен, Нидерланды	Баргачев Виктор Сандлер Марк	Санкт-Петербург Нижний Новгород
1996 г., 25 июля–1 августа, Весэпрем, Венгрия	Сандлер Марк Дуров Николай Матюхин Виктор	Нижний Новгород Санкт-Петербург Киров
1997 г., 30 ноября–7 декабря Кейптаун, ЮАР	Мартьянов Владимир	Нижний Новгород
1998 г., 5–12 сентября, Сетубал, Португалия	Мартьянов Владимир	Нижний Новгород
1999 г., 9–16 октября, Анталья, Турция	Пастухов Роман Мартьянов Владимир Бабенко Максим	Оренбург Нижний Новгород Саратов
2000 г., 23–30 сентября Пекин, Китай	Баутин Михаил Митричев Петр Пастухов Роман Круглов Алексей	Нижний Новгород Москва Оренбург Нижний Новгород
2001 г., 14–21 июля Тампере, Финляндия	Румянцев Андрей	Москва
2002 г., 18–25 августа Сеул, Южная Корея	Митричев Петр Калинин Петр	Москва Нижний Новгород
2003 г., 18–25 августа Чикаго, США	Орлов Дмитрий	Саратов
2004 г., 11–18 сентября, Афины, Греция	Батузов Кирилл Шавлюгин Евгений Зыкова Александра Гольдштейн Виталий	Саратов Владивосток, Приморский край Санкт-Петербург Саратов

Страна, год	ФИО победителей	Город
2005 г., 18–25 августа Новы-Сонч, Польша	Копелиович Сергей	Санкт-Петербург
2006 г., 13–20 августа Мерида, Мексика	Копелиович Сергей Разенштейн Илья Денисов Денис	Санкт-Петербург Нижний Новгород Петрозаводск, Республика Карелия

ПЕРВАЯ МЕЖДУНАРОДНАЯ ОЛИМПИАДА

Условие задачи

Дана последовательность из $2*n$ ($n \leq 5$) ячеек. Две соседние из них — пустые, а в остальных расположены $n - 1$ символов A и $n - 1$ символов B . Пример для $n = 5$:

A	B	B	A			A	B	A	B

Правило перемещения. Содержимое любых двух непустых ячеек, сохраняя их порядок, можно пересылать в пустые ячейки.

Цель. Используя правило перемещения, достичь конфигурации, в которой все символы A расположены левее всех символов B . Местоположение пустых ячеек после перемещений не имеет значения.

Задание. Составить программу, которая:

- Вводит с клавиатуры начальную конфигурацию в виде последовательности символов A , B и нулей для пустых ячеек, а также моделирует перемещения. Каждое перемещение задается номером (от 1 до $n - 1$) первой из двух соседних ячеек, содержимое которых пересылается в пустые ячейки. Программа должна выводить на экран конфигурацию после каждого перемещения.
- Для заданной начальной конфигурации определяет по крайней мере один план перемещений, с помощью которого можно достичь цели, или сообщает, что такого плана не существует (вывод должен содержать начальную конфигурацию, промежуточные конфигурации после каждого перемещения, а также заключительную конфигурацию).
- Находит некоторый план, достигающий цели за минимальное число перемещений.

Обсуждение задачи

Ответим на два вопроса. Во-первых, подсчитаем количество различных последовательностей для заданного значения n . Количество способов размещения «00» в последовательности длины $2*n$ равно $2*n - 1$. Для каждого способа расположения «00» в оставшихся $2*n - 2$ позициях $n - 1$ букву A можно разместить C_{2n-2}^{n-1} способами. Получаем $(2*n - 1) * C_{2n-2}^{n-1}$ вариантов. Так, при $n = 5$ количество возможных состояний нашей полоски ячеек не превосходит $9 * C_8^4 = 630$.

Во-вторых, нам требуется определить отношение порядка на множестве последовательностей. Другими словами, для каждой последовательности мы должны знать, какая последовательность ей предшествует и какая следует за ней. Как только это будет сделано, у нас сразу появится возможность перебирать последовательности. Заметим, что, удалив из описывающей конфигурацию строки символов нули, мы получим строку из $n - 1$ символа A и $n - 1$ символа B . Позиции символов A в полученной строке образуют $(n - 1)$ -элементное подмножество множества $[1 \dots 2*n - 2]$. Все такие сочетания из $2*n - 2$ элементов по $n - 1$ мы можем пронумеровать натуральными числами в лексикографическом порядке*. Пусть k — номер сочетания, соответствующего полученной строке, а нули в исходной строке занимают позиции с номерами p и $p + 1$. Тогда исходной строке можно поставить в соответствие пару чисел (p, k) . При таком соответствии каждая последовательность кодируется парой чисел (p, k) , где $1 \leq p \leq 2*n - 1$ и $1 \leq k \leq C_{2n-2}^{n-1}$, причем каждой такой паре соответствует какая-либо позиция, а разные позиции кодируются разными парами (при заданном n). В табл. 1.1 (при $n = 3$) приведено соответствие между последовательностями и парами описывающих их чисел. Жирным шрифтом выделены конечные последовательности. Заметим, что этим последовательностям соответствуют пары чисел типа $(p, 1)$, где p — позиция первого нуля.

Таблица 1.1

№	Последовательность	№	Последовательность	№	Последовательность
1, 1	00AABV	2, 5	B00ABA	4, 3	ABV00A
1, 2	00ABAB	2, 6	B00BAA	4, 4	BAV00B
1, 3	00ABBA	3, 1	AA00BV	4, 5	BAV00A
1, 4	00BAAB	3, 2	AB00AB	4, 6	BVA00A
1, 5	00BABA	3, 3	AB00BA	5, 1	AAVB00
1, 6	00VBAA	3, 4	BA00AB	5, 2	ABAB00
2, 1	A00ABV	3, 5	BA00BA	5, 3	ABBA00
2, 2	A00BAB	3, 6	BV00AA	5, 4	BAAB00
2, 3	A00BBA	4, 1	AAV00B	5, 5	BAVA00
2, 4	B00AAB	4, 2	ABA00B	5, 6	BVAA00

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004.

Что нам требуется уметь делать к этому моменту времени? По последовательности определять пару чисел (p, k) и, наоборот, по паре чисел определять последовательность. Эти задачи известны в комбинаторике — по комбинаторному объекту определять его номер в лексикографическом порядке и выполнять обратное преобразование. Предположим, что мы уже умеем решать эти задачи, и продолжим рассуждения.

Пусть нам дана последовательность $BBA00A$ ($n = 3$). Соответствующая ей пара чисел равна $(4, 6)$. Попробуем прорисовать вручную по таблице все возможные переходы из одной последовательности в другую. Результат этой работы представлен на рис. 1.1. Можно назвать его *графом состояний*. Оказывается, что из десяти исходных последовательностей достичь ни одной из конечных последовательностей нельзя. Однако при $n > 3$ из любой начальной последовательности всегда достигается одна из конечных.

Итак, мы находимся в состоянии $(4, 6)$. Из рис. 1.1 ясно, что далее требуется идти в состояние $(1, 3)$, а затем — в $(3, 3)$, $(5, 3)$ и $(2, 1)$. Возникает вопрос: как зафиксировать (сформировать) связи между состояниями? С помощью двух матриц A и How (один из возможных способов), где вторая матрица требуется для вывода последовательности перемещений:

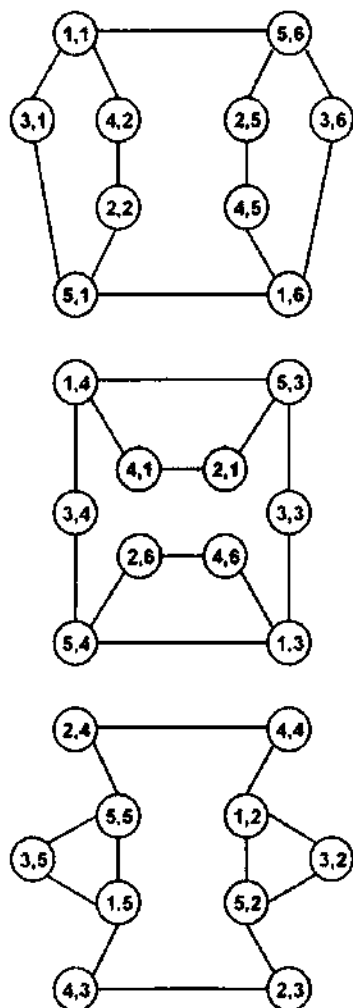


Рис. 1.1. Граф состояний при $n = 3$

$$A[p, k] = \begin{cases} 0, & \text{если состояние } (p, k) \text{ конечное,} \\ -1, & \text{если из состояния } (p, k) \text{ недостижимо} \\ & \text{ни одно конечное состояние,} \\ t & - \text{минимальное количество перемещений из} \\ & \text{состояния } (p, k) \text{ в одно из конечных состояний;} \end{cases}$$

$$How[p, k] = \begin{cases} -1, & \text{если из состояния } (p, k) \text{ недостижимо} \\ & \text{ни одно конечное состояние или оно конечное,} \\ p_{\text{новое}} & - \text{позиция первого нуля при переходе} \\ & \text{в следующее состояние.} \end{cases}$$

Индекс p изменяется от 1 до $2n - 1$, а индекс k — от 1 до C_{2n-2}^{n-1} . Для рассматриваемого примера ($n = 3$) матрицы A и How имеют вид:

$$A = \begin{Bmatrix} 0 & -1 & 3 & 1 & -1 & 1 \\ 0 & 1 & -1 & -1 & 2 & 4 \\ 0 & -1 & 2 & 2 & -1 & 2 \\ 0 & 1 & -1 & -1 & 2 & 4 \\ 0 & -1 & 1 & 3 & -1 & 1 \end{Bmatrix}, \quad How = \begin{Bmatrix} -1 & -1 & 3 & 4 & -1 & 5 \\ -1 & 5 & -1 & -1 & 5 & 5 \\ -1 & -1 & 5 & 1 & -1 & 5 \\ -1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & 2 & 3 & -1 & 1 \end{Bmatrix}.$$

Вернемся к нашему примеру. По последовательности $BBA00A$ мы определили пару чисел, ей соответствующих: (4, 6). Значение $A[4, 6] = 4$. Значит, за четыре хода эта последовательность преобразуется в конечную.

Первый ход: $How[4, 6] = 1$ — перемещаем «00» в первую позицию последовательности и получаем: $00ABBA$. Вычисляем пару чисел: (1, 3).

Второй ход: $How[1, 3] = 3$, перемещаем «00» в третью позицию последовательности и получаем $AB00BA$. Вычисляем пару чисел: (3, 3).

Третий ход: $How[3, 3] = 5$, перемещаем «00» в пятую позицию последовательности и получаем $ABBA00$. Вычисляем пару чисел: (5, 3).

Четвертый ход: $How[5, 3] = 2$, перемещаем «00» во вторую позицию последовательности и получаем $A00ABB$.

Осталось понять логику формирования значений элементов матриц A и How . Продемонстрируем ее:

{Помечаем все конечные вершины нулями}

For <всех пар (p, k) > **Do**

If <Пара (p, k) задает конечное состояние> **Then**

$A[p, k] := 0$

Else $A[p, k] := -1$;

{Расставляем метки}

$t := -1$;

Repeat

```

t:=t+1;
ok:=True;
For <всех пар (p,k) с A[p,k]=t> Do
  For <всех состояний, описываемых парой (q,j),
из которых можно попасть в (p,k)> Do
    If A[q,j]=-1 Then Begin
      A[q,j]:=t+1;
      How[q,j]:=p;
      ok:=False;
    End;
Until ok;

```

Итак, начиная с конечных состояний, мы просматриваем все состояния, из которых могут быть достигнуты конечные состояния, подсчитывая при этом, за сколько перемещений это можно сделать (матрица *A*), и фиксируя путь в матрице *How*.

Тесты, использовавшиеся на международной олимпиаде, приведены в табл. 1.2.

Таблица 1.2

№	Последовательность	Ожидаемый ответ
1	00ABAB	Нет решения
2	0ABA0BAB	Некорректно
3	00ABABABAB	Решение за 4 перемещения
4	ABBA00ABAB	Решение за 3 перемещения

Методические рекомендации

1. Данная задача требует знаний по комбинаторике, а именно решения основных четырех задач при работе с комбинаторными объектами (в данном случае — с сочетаниями): подсчета количества комбинаторных объектов, генерации комбинаторных объектов в определенном порядке (обычно — в лексикографическом), получения номера комбинаторного объекта в соответствии с введенным отношением порядка и определения комбинаторного объекта по его номеру.
Ключевой момент: только после того, как нами будет осознан порядок на множестве неких объектов, можно подключать к решению задачи всю мощь компьютера.
2. Основная логика решения — это ни что иное как модификация известного *метода поиска в ширину*. В данном случае он сводится к тому, что из всех конечных состояний отыскива-

ются состояния, из которых они достижимы. Эти состояния помечаются меткой «1». Затем рассматриваются все состояния с меткой «1» и осуществляется поиск тех состояний, из которых мы попадаем в состояния с меткой «1». Найденные состояния (если они есть) помечаются меткой «2». Эти состояния мы считаем помеченными, но не обработанными. Процесс продолжается обработкой состояний с меткой «2» (отыскиваются состояния с меткой «3») и т. д., пока появляются необработанные, но помеченные состояния. Лучше всего тема поиска в ширину осваивается при изучении алгоритмов на графах*.

3. Самым простым вариантом реализации выражения $\text{For } \langle \text{всех пар } (p, k) \text{ с } A[p, k] = t \rangle \text{ Do}$ является просмотр элементов матрицы A . Однако если мы будем хранить эти состояния (помеченные, но необработанные) в очереди, то такой просмотр не потребует, а потребует нам умение работать с абстрактной структурой данных — *очередью*. Развивая эту мысль дальше, можно говорить о рекурсивной и не рекурсивной логике работы с элементами очереди.

Резюмируя методические особенности задачи, можно сформулировать следующие требования к алгоритмическим знаниям школьника:

- комбинаторика, отношение порядка на множестве комбинаторных объектов;
- поиск в ширину — как методология решения задачи путем ее разбивки на подзадачи;
- абстрактные типы данных.

Самостоятельные исследования

1. Для решения задачи можно использовать только одну матрицу. Как это сделать?
2. Анализируя рис. 1.1, можно увидеть, что конечные состояния обладают, если так можно выразиться, разным «весом». Так, состояние (3, 1) недостижимо ни из какого состояния, то есть его «вес» равен нулю. Сколько таких состояний существует при заданном значении n ? (Возможна другая модификация вопроса: найти «веса» всех конечных состояний.)
3. В задаче имеется ограничение на значение n ($n \leq 5$). Временная сложность алгоритма — полиномиальная. Естественно, возникает вопрос о решении задачи для больших значений n . Если до значения $n = 7$ задача решается без особых

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С.143–144.

проблем, то при $n > 7$ она уже является предметом отдельного рассмотрения.

4. Факт связности графа состояний при $n > 3$ установлен нами на эвристическом уровне: «при $n = 4$ он связан, т. е. нет состояний, из которых не достигается хотя бы одно конечное состояние». Далее делается вывод о том, что он связан и при больших значениях n . Так ли это? Формальная логика требует доказательства этого утверждения.

ЗАДАЧА ПЕРВОГО ТУРА**Условие задачи**

Задана таблица размером 4×4 , в каждой клетке которой, кроме двух, содержится одно из чисел от 1 до 14 (все числа разные). Оставшиеся две клетки пустые. Пример — табл. 2.1.

Таблица 2.1

7	3	5	14
	4	9	13
1		2	10
11	8	12	6

Таблица 2.2

1	2	3	4
5	6	7	8
9	10	11	12
13	14		

Правило перемещения. Число из любой клетки может быть перемещено в любую незанятую соседнюю клетку по горизонтали или по вертикали. Клетка, в которой ранее размещалось число, становится пустой.

Цель. Необходимо с помощью указанного правила выполнять по шагам преобразование произвольной исходной таблицы в конечную табл. 2.2.

Задание. Написать программу, которая:

- осуществляет ввод с клавиатуры исходной таблицы и вывод ее на экран (пустые клетки могут быть закодированы нулями);
- выполняет преобразование введенной таблицы в табл. 2.2;
- на каждом шаге выдает на экран слева матрицу до хода, справа — матрицу после хода и указывает номер хода (1, 2, 3 и т. д.) так, что в конце работы программы будет показано полное число сделанных ходов;
- минимизирует число ходов, требуемых для решения задачи.

Идея решения

В дальнейшем, по аналогии с известной игрой «Пятнашки», мы будем называть числа, стоящие в клетках, *фишками*. Про пустую же клетку мы будем говорить, что в ней находится «пустая» фишка.

Для начала решим задачу с упрощенным правилом перемещения. Пусть разрешается обмен между соседними клетками

независимо от того, заняты они или нет. Реализуем пункты задания при этом правиле. Если мы это сделаем, то затем останется только добавить логику учета перемещения фишек через пустые клетки.

В массиве A фиксируем местоположения всех фишек. Так, если $A[2, 3] = 10$, то фишка с номером 10 находится во второй строке и в третьем столбце. Столбцы нумеруются слева направо, строки — сверху вниз.

Общая логика сводится в этом случае к одной-единственной строке:

For $n:=1$ **To** 14 **Do**

 <Перенести фишку n на свое место>

Обозначим через (x, y) координаты клетки, в которой находится фишка с номером n , а через (px, py) — место, где она должна находиться (рис. 2.1).

На рис. 2.1 выделен путь перемещения фишки на свое место. Он состоит из отдельных шагов:

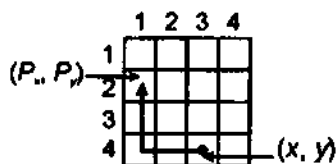


Рис. 2.1. Пример перемещения фишки

Procedure Move(dx, dy : Integer);

Begin

 <переместить фишку n с клетки (x, y) в клетку $(x+dx, y+dy)$ >;

$x:=x+dx$;

$y:=y+dy$;

 <вывести таблицу до перемещения и после перемещения>;

End;

В целом же полное перемещение реализуется с помощью процедуры:

Procedure Post;

{ n, x, y — глобальные переменные}

Begin

 { Вычисляем px и py }

$px:=(n-1) \bmod 4 + 1$;

$py:=(n-1) \div 4 + 1$;

 <Находим в таблице клетку (x, y) с фишкой n >

$dx:=\text{Sgn}(px-x)$;

$dy:=\text{Sgn}(py-y)$;

While $x \neq px$ **Do** Move($dx, 0$);

While $y \neq py$ **Do** Move($0, dy$);

End;

Таким образом, если бы в задаче не было пустых клеток, а правило перемещения сводилось к обмену значений в соседних клетках, то задание было бы выполнено.

Перейдем к дальнейшему рассмотрению задачи. Итак, у нас есть две пустые клетки, а перемещение фишки на свое место возможно только через пустую клетку. Идея решения в этом случае достаточно проста. Перед каждым ходом на пути из клетки (x, y) в клетку (px, py) необходимо ставить пустую фишку. Координаты пустой фишки мы обозначим через (sx, sy) . При этом мы выбираем ближайшую к клетке (x, y) пустую фишку, кроме особых случаев.

На какое место ставить пустую фишку? Это клетка с координатами $(x + dx, y)$ при перемещении по оси x и $(x, y + dy)$ при перемещении по оси y . Процедура Move при этом незначительно изменится. Необходимо поменять местами пустую фишку и фишку с номером n , а также запомнить новое место пустой фишки. Кроме того, мы будем придерживаться следующих ограничений, упрощающих логику расстановки фишек: во-первых, фишку с номером n , которую мы устанавливаем на свое место, при перемещении пустой фишки перемещать не разрешается, а во-вторых, ранее установленные на свои места фишки больше трогать нельзя.

Один из основных вопросов, который возникает при решении этой задачи: зачем нужны две пустые фишки? Ответ на него дают особые случаи в расстановке фишек. Пусть на своем месте находятся фишки с номерами 1, 2, 3 (рис. 2.2а). Очередь — за фишкой с номером 4, причем при перемещении фишки 4 мы договорились не трогать фишки с номерами 1, 2, 3.

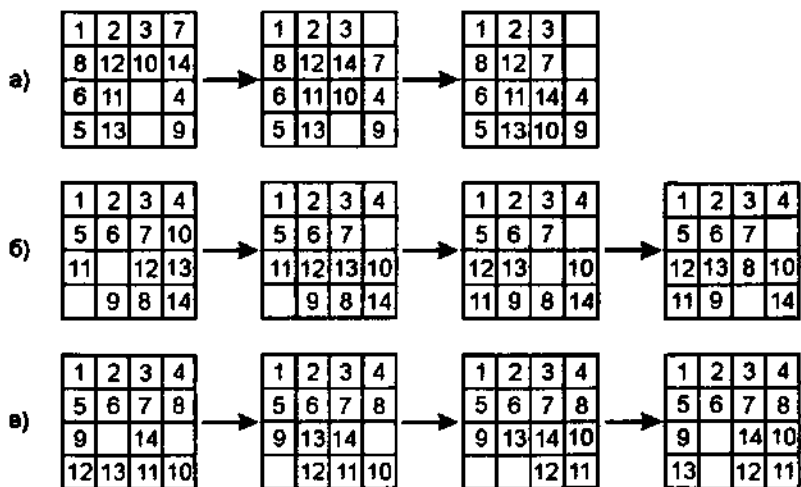


Рис. 2.2. Особые случаи при перемещении фишек

Наши действия: помещаем в клетку (1, 4) пустую фишку (рис. 2.2а), а затем обычным образом «гоним» фишку с номером 4 на свое место, устанавливая каждый раз перед ней другую пустую фишку (рис. 2.2а). Коллизий при этом не возникает, так как есть две пустые фишки и одна из них стоит на том месте, куда мы должны поставить фишку с номером 4. Аналогичная ситуация — с фишкой 8, когда фишки 1, 2, 3, 4, 5, 6, 7 уже стоят на своих местах (рис. 2.2б). Чтобы реализовать эту идею для фишек с номерами 13 и 14, фишку 13 необходимо устанавливать на свое место после фишки 9 (рис. 2.2в), а фишку 14 — после фишки 10. Поэтому данные фишки мы будем устанавливать в следующем порядке: 1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 10, 14, 11, 12.

В особом случае — для фишек 4, 8, 13 и 14 — необходимо сначала установить одну из пустых фишек в клетку (px , py). После этого дальнейшая работа с указанными фишками ничем не отличается от установки фишек с другими номерами. Для фишек 4 и 8 мы вначале перемещаем пустую фишку по оси x , а затем — по оси y ; для фишек 13 и 14 — по оси y , а затем — по оси x , поэтому ранее установленные на свои места фишки не затрагиваются.

В общем случае при каждом передвижении устанавливаемой фишки нам необходимо выбирать одну из двух пустых фишек. Будем выбирать ту из них, для перемещения которой на нужное место требуется наименьшее число ходов. Его, для перемещения пустой фишки из клетки (x_1, y_1) в клетку (x_2, y_2) можно оценить снизу выражением $Abs(x_1 - x_2) + Abs(y_1 - y_2)$. В особых случаях требуемое количество ходов не равно этой оценке, а больше нее на 2 или 4: ведь сами эти случаи возникают из-за того, что при перемещении пустой фишки мы не имеем права трогать устанавливаемую и ранее установленные фишки.

Методическое отступление. Поиск и «прорисовка» особых случаев перемещения пустых фишек создают на занятии творческую атмосферу.

Методические указания

Это — типичная задача на технику программирования. После изучения темы «Двумерные массивы» она может быть использована в качестве задания для самостоятельной работы.

Самостоятельные исследования

1. Игра «хамелеон» проводится на доске с девятью клетками, которые соединены отрезками, как показано на рис. 2.3. На восьми фишках выписаны буквы: «х», «а», «м», «е», «л», «е», «о», «н». Фишки случайным образом расставлены на

клетках 1–8. Клетка 9 пуста. Цель игры состоит в том, чтобы, передвигая фишки по отрезкам, разместить их в правильном порядке, т. е. чтобы при чтении по часовой стрелке, начиная с первой клетки, получалось слово «хамелеон». Требуется написать программу для решения этой задачи.

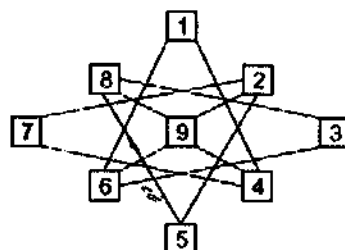


Рис. 2.3. Игровое поле в игре «хамелеон»

2. Круг разбит на 8 секторов (рис. 2.4). Какие-то два соседних сектора пусты, а в остальных расположены буквы: «Р», «О», «С», «С», «И», «Я» в некотором порядке (по одной в каждом секторе).

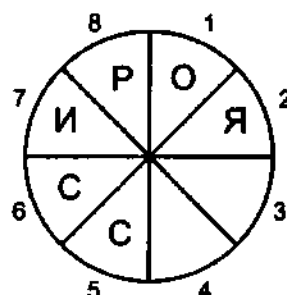


Рис. 2.4. Пример первоначального размещения букв в секторах круга

За один ход разрешается взять две любые буквы и перенести их в пустые сектора, сохранив порядок. При этом секторы, которые перенесенные буквы занимали до этого хода, становятся пустыми.

Состояние круга мы будем называть «правильным», если начиная с сектора 1 по часовой стрелке можно прочесть слово «РОССИЯ». При этом местоположение пустых секторов может быть произвольным.

Для заданного состояния круга требуется найти кратчайшую последовательность ходов, переводящих круг в «правильное» состояние. Если решений несколько, то необходимо вывести только один вариант.

ЗАДАЧА ВТОРОГО ТУРА

Условие задачи

В картинной галерее каждый сторож работает в течение некоторого непрерывного отрезка времени. Расписанием стражи называется множество пар $[T_1(i), T_2(i)]$ — моментов начала и конца дежурства i -го сторожа из интервала $[0, EndTime]$.

Для заданного расписания стражи требуется:

- проверить, в любой ли момент в галерее находится не менее двух сторожей;

если условие пункта а) не выполняется, то:

- б) перечислить все интервалы времени с недостаточной охраной (менее двух сторожей);
- в) добавить наименьшее число сторожей с заданной, одинаковой для всех длительностью дежурства так, чтобы получить правильное расписание, удовлетворяющее условию пункта а);
- г) проверить, можно ли обойтись без добавления новых сторожей, если разрешается сдвигать время дежурства каждого сторожа с сохранением длительности его дежурства;
- д) при положительном ответе на пункт г) составить расписание с наименьшим числом сдвигов.

Входные данные (все моменты времени задаются в целых минутах):

- *EndTime* — момент окончания стражи (момент начала — 0);
- *n* — число сторожей;
- $T_1(i), T_2(i), i = 1, \dots, n$ — моменты начала и окончания дежурства *i*-го сторожа;
- *Length* — длительность дежурства каждого дополнительного сторожа.

Выходные данные:

- 1) ответ на пункт а) в форме «да»/«нет»;
- 2) при ответе «нет» на пункт а) — список пар (*k, l*) — начал и концов всех малоохраняемых интервалов с указанием числа сторожей в каждом (0 или 1);
- 3) число дополнительных сторожей и моменты начала и окончания дежурства каждого дополнительного сторожа;
- 4) ответ на пункт г) в форме «да»/«нет»; если «да», то номера сторожей, смена которых сдвигается, и значения сдвигов;
- 5) ответ на пункт д) — наименьшее число сторожей, смена которых сдвигается, их номера и значения сдвигов.

Примечание. Программа должна допускать независимое тестирование пунктов в), г), д).

Обсуждение задачи

Рассмотрим первые два пункта задания. Подсчитаем для каждого *j* от 0 до *EndTime* - 1 в элементе массива *time[j]* количество сторожей, которые находятся в галерее в промежуток времени [*j, j + 1*]. Если для какого-то значения *j* их больше двух, то записывать в *time[j]* мы будем только двойку.

После заполнения массива *time* имеем:

$$time[j] = \begin{cases} 0, & \text{если в момент времени } j \text{ в галерее} \\ & \text{нет ни одного сторожа,} \\ 1, & \text{если в момент времени } j \text{ в галерее} \\ & \text{дежурит один сторож,} \\ 2, & \text{если в момент времени } j \text{ в галерее} \\ & \text{дежурят два и более сторожа.} \end{cases}$$

Данные в массиве *time* являются исходными для получения ответа на первое задание (пункт а) и для формирования интервалов времени с недостаточной охраной. Эти интервалы фиксируются в массиве *NoGuard* : Array [1..n, 1..3] Of Integer. В каждой строке массива при этом записывается начальное время, конечное время и количество дежурящих сторожей (0 или 1).

Пример.

Пусть *EndTime* = 5, количество сторожей *n* = 6, а интервалы дежурств: (0, 1), (0, 1), (3, 4), (3, 4), (3, 5) и (0, 2). Тогда массив *time* = (2, 1, 0, 2, 1), а массив *NoGuard* будет выглядеть, как показано в табл. 2.3, итерация 1.

Таблица 2.3

Номер итерации	<i>n</i>	Начальное время	Конечное время	Количество дежурящих сторожей
1	1	1	2	1
1	2	2	3	0
1	3	4	5	1
2	1	2	3	1
2	2	4	5	1
3	1	4	5	1

Для завершения выполнения второго пункта задания (пункт б) нам остается вывести значения элементов массива *NoGuard*.

Рассмотрим третий пункт задания (пункт в). Пусть мы ввели длительность дежурства, равную 2. В анализируемом примере при первой итерации массив *NoGuard* имеет вид, приведенный в табл. 2.3.

Добавим одного сторожа с интервалом дежурства (1, 3) и после этого вновь сформируем массив *NoGuard* (табл. 2.3,

итерация 2). Примем на работу еще одного сторожа с интервалом дежурства (2, 4) и опять проверим достаточность охраны (табл. 2.3, итерация 3). Так как у нас по прежнему есть интервал с недостаточной охраной, то добавим очередного сторожа с интервалом дежурства (3, 5). После этого в любой момент времени в галерее будет находиться не менее двух сторожей.

Рассмотрим четвертый пункт задания (пункт г). Идея решения заключается в разбиении множества дежурств сторожей на два подмножества так, чтобы сумма времени дежурств сторожей в каждом из подмножеств была больше или равна *EndTime*. Очевидно, что достаточно найти одно такое подмножество, так как второе получается его дополнением до всего множества дежурств сторожей. Нахождение подобного разбиения возможно только в случае, если сумма времени дежурств всех сторожей (*Sum*) больше или равна $2 * EndTime$. При этом сумма времени дежурств сторожей из первого подмножества (*cnt*) должна удовлетворять условию: $EndTime \leq cnt \leq Sum - EndTime$, иначе сторожа из второго подмножества не смогут обеспечить необходимой охраны галереи. При нахождении же такого разбиения график дежурства сторожей из каждого подмножества составляется по принципу: когда первый заканчивает дежурство, второй начинает дежурить.

Логика поиска разбиения на подмножества приведена ниже:

```

Var ok : Boolean; {Найдено ли решение?}
Procedure Разбиение(k : Integer); {k — номер сторожа}
  Var j : Integer;
  Begin
    If cnt > Sum - EndTime Then Exit;
    If cnt >= EndTime Then
      Begin
        ok := True; {Решение найдено}
      Exit;
    End;
    For j := k To n Do
      Begin
        cnt := cnt + <продолжительность дежурства сторожа j>;
        Разбиение(j+1);
        cnt := cnt - <продолжительность дежурства сторожа j>;
        If ok Then
          <запомним, что сторож j — в первом подмножестве>;
      End;
    End;
  End;

```

Вызывать процедуру Разбиение необходимо следующим образом:

```
cnt:=0;  
ok:=False;  
Разбиение(1);
```

Пятый пункт задания (пункт д) предполагает решение задачи для небольших значений n . При этом мы сначала рассматриваем всевозможные сдвиги времени дежурства одного сторожа (вначале первого, затем второго и так далее) и пытаемся «залатать дыры» в расписании. Если это не удастся, то переходим к «двухэлементным» сдвигам, т. е. к сдвигам времени дежурств каких-либо двух из наших n сторожей, затем — к «трехэлементным» и т. д. до « n -элементных» сдвигов, пока не будет найдено правильное расписание:

```
Var ok : Boolean; {Найдено ли решение?}  
Procedure Правильное_расписание;  
Var k : Integer; {Количество сторожей в подмножестве}  
Begin  
  k:=0; ok:=False;  
  Repeat  
    k:=k+1;  
    For <всех  $k$ -элементных подмножеств множества  
сторожей> Do  
      If <можно "заткнуть дыры" в расписании  
с помощью сторожей из выбранного подмножества>  
      Then Begin  
        ok:=True;  
        Break;  
      End;  
  Until (k=n) Or ok;  
  If ok Then <Вывод расписания>  
  Else WriteLn('Правильное расписание получить  
нельзя.');
```

Итак, одна из подзадач здесь — это генерация k -элементных подмножеств n -элементного множества. Она хорошо известна* и не требует разъяснений. Вторая подзадача — перебор всех возможных способов «затыкания дыр» в расписании с помощью

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 61–64.

сторожей из выбранного подмножества. Ее решение — модификация пункта (в) основной задачи.

Методические рекомендации

Первые три задания (пункты а–в) являются обычными учебными задачами при работе с массивами. Четвертый и пятый пункты (г и д) — предполагают для небольших значений n традиционный перебор вариантов и знание основных комбинаторных объектов.

Самостоятельные исследования

1. Поиск решения для пятого пункта, основанного не на переборе вариантов, является нетривиальной задачей.
2. В задаче не дается ограничение на значение n . Требуется экспериментальным путем установить значение n , при котором решение, основанное на переборе вариантов, будет работать больше 1 секунды.

ЗАДАЧА ПЕРВОГО ТУРА

Условие задачи

Пронумеровать позиции в матрице (таблице) размером 5×5 следующим образом. Если номер i ($1 \leq i < 25$) соответствует позиции с координатами (x, y) , то номер $i + 1$ может соответствовать позиции с координатами (z, w) , вычисляемыми по одному из следующих правил:

- 1) $(z, w) = (x \pm 3, y)$;
- 2) $(z, w) = (x, y \pm 3)$;
- 3) $(z, w) = (x \pm 2, y \pm 2)$.

Требуется:

- написать программу, которая последовательно нумерует позиции матрицы 5×5 при заданных координатах позиции, в которой проставлен номер 1 (результаты должны быть представлены в виде заполненной матрицы);
- вычислить число всех возможных расстановок номеров для всех начальных позиций, расположенных в правом верхнем треугольнике матрицы, включая ее главную диагональ.

Пример. Если в качестве начальной позиции в матрице выбрана позиция с координатами $(2, 2)$, то на данном шаге координаты позиции с номером 2 в соответствии с представленными правилами могут быть: $(2, 5)$, $(5, 2)$ или $(4, 4)$ (на рис. 3.1 они помечены звездочкой).

Примечание. Будет оцениваться, похож ли вывод вашей программы на наш пример (рис. 3.1).

Оценка решения:

- Пункт первый — 50 баллов.
- Пункт второй — 25 баллов.
- Вывод — 15 баллов.
- Жюри — 10 баллов (стиль программирования, элегантность).

Идея решения

Эта задача — ни что иное как модификация хорошо известной задачи об обходе доски шахматным конем. Поэтому здесь мы приведем лишь ответы на второй пункт условия задачи.

	1	2	3	4	5
1					
2		1			*
3					
4				*	
5		*			

Рис. 3.1. Возможные позиции при перемещении из $(2, 2)$

В табл. 3.1 для каждой требуемой начальной позиции указано количество всех возможных расстановок. Общее их количество равно 7340.

Таблица 3.1

552	548	552	548	552
	412	400	412	548
		352	400	552
			412	548
				552

Методические указания

Данная задача в свете нынешнего состояния дел в олимпиадном движении носит простой учебный характер по реализации классической схемы перебора с возвратом и подчеркивает стремительность развития сложности задач олимпиад по информатике. Для изучения темы «Перебор с возвратом» существует стандартный набор задач: «расстановка ферзей на шахматной доске», «поиск обхода конем шахматной доски», «поиск выхода из произвольного лабиринта» и т. д.* Данная задача дополняет этот перечень.

Самостоятельные исследования

1. При каком значении n значения в табл. 3.1 будут превышать максимальное значение для типа данных Integer (32767)?
2. При каких значениях n время работы программы на вашем компьютере будет равно 1 с? 2 с? 10 с?

ЗАДАЧА ВТОРОГО ТУРА

Условие задачи

S-терм — это последовательность символов S и скобок, определяемая рекурсивно следующим образом:

- символ S есть S -терм;
- если M и N — S -термы, то выражение (MN) есть также S -терм.

Пример S-терма: $((((SS)(SS))S)(SS))$.

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 79–140.

Правые скобки не несут информации и могут опускаться. В этом случае вышеприведенный S -терм выглядит так:
 $((((SS(SSS(SS$

1. Напишите процедуру «gensterm» для порождения S -термов. Она должна для заданного n заполнять n текстовых файлов (n = длина = число символов « S »), каждый из которых содержит все S -термы длины 1, 2, ..., n соответственно. Внутри файла S -термы разделяются символом «;». В конце каждого файла должен стоять символ «.» (точка).

Напишите программу, которая по заданному целому n ($n \leq 10$) выполняет описанную выше процедуру и выдает на дисплей все сгенерированные S -термы.

Рассмотрим исчисление S -термов. Единственное алгебраическое правило (S -правило), которое может быть использовано, состоит в следующем: любой подтерм S -терма, имеющий вид $((SA)BC)$, где A , B и C — также S -термы, может быть переписан как $((AC)(BC))$, то есть $\text{Context1}(((SA)BC)) \rightarrow \text{Context2} \rightarrow \text{Context1}((AC)(BC))\text{Context2}$.

Применение этого правила к S -терму называется *редукцией* S -терма. Возможны разные способы (стратегии) выбора подтермов для применения S -правила. Последовательное применение S -правила к S -терму до тех пор, пока это возможно, называется *нормализацией*.

Пример цепочки редукции S -терма:

$$\begin{aligned} &(((SS)(SS))S)(SS) \rightarrow (((SS)((SS)S))(SS) \rightarrow \\ &((S(SS))(((SS)S)(SS))) \rightarrow ((S(SS))((S(SS))(S(SS)))) \end{aligned}$$

2. Предложите эффективную структуру данных для представления S -термов, облегчающую применение S -правила. Написать две процедуры: «readterm» и «printterm». Первая из них преобразует S -термы в вашу структуру данных из формы, порождаемой процедурой «gensterm»; вторая преобразует S -термы из вашей структуры в форму, порождаемую процедурой «gensterm». Ваша программа должна демонстрировать эти преобразования.

3. Напишите процедуру «reduce», выполняющую один шаг редукции в соответствии с S -правилом над заданным подтермом S -терма в вашем представлении. Программа должна продемонстрировать это.

4. Напишите процедуру «normalize», которая в заданном S -терме должна последовательно выбирать подтермы и применять S -правило до тех пор, пока дальнейшие редукции станут невозможными, либо число шагов достигнет некоторого максимума, например 30. Программа должна продемонстрировать это.

В табл. 3.1 для каждой требуемой начальной позиции указано количество всех возможных расстановок. Общее их количество равно 7340.

Таблица 3.1

552	548	552	548	552
	412	400	412	548
		352	400	552
			412	548
				552

Методические указания

Данная задача в свете нынешнего состояния дел в олимпиадном движении носит простой учебный характер по реализации классической схемы перебора с возвратом и подчеркивает стремительность развития сложности задач олимпиад по информатике. Для изучения темы «Перебор с возвратом» существует стандартный набор задач: «расстановка ферзей на шахматной доске», «поиск обхода конем шахматной доски», «поиск выхода из произвольного лабиринта» и т. д.* Данная задача дополняет этот перечень.

Самостоятельные исследования

1. При каком значении n значения в табл. 3.1 будут превышать максимальное значение для типа данных Integer (32767)?
2. При каких значениях n время работы программы на вашем компьютере будет равно 1 с? 2 с? 10 с?

ЗАДАЧА ВТОРОГО ТУРА

Условие задачи

S-терм — это последовательность символов S и скобок, определяемая рекурсивно следующим образом:

- символ S есть S -терм;
- если M и N — S -термы, то выражение (MN) есть также S -терм.

Пример S-терма: $(((((SS)(SS))S)(SS))$.

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 79–140.

Правые скобки не несут информации и могут опускаться. В этом случае вышеприведенный S -терм выглядит так: $((((SS(SSS(SS$.

1. Напишите процедуру «gensterm» для порождения S -термов. Она должна для заданного n заполнять n текстовых файлов (n = длина = число символов « S »), каждый из которых содержит все S -термы длины 1, 2, ..., n соответственно. Внутри файла S -термы разделяются символом «;». В конце каждого файла должен стоять символ «.» (точка).

Напишите программу, которая по заданному целому n ($n \leq 10$) выполняет описанную выше процедуру и выдает на дисплей все сгенерированные S -термы.

Рассмотрим исчисление S -термов. Единственное алгебраическое правило (S -правило), которое может быть использовано, состоит в следующем: любой подтерм S -терма, имеющий вид $((SA)BC)$, где A , B и C — также S -термы, может быть переписан как $((AC)(BC))$, то есть $\text{Context1}(((SA)BC)) \rightarrow \text{Context2} \rightarrow \text{Context1}((AC)(BC))\text{Context2}$.

Применение этого правила к S -терму называется *редукцией* S -терма. Возможны разные способы (стратегии) выбора подтермов для применения S -правила. Последовательное применение S -правила к S -терму до тех пор, пока это возможно, называется *нормализацией*.

Пример цепочки редукции S -терма:

$$\begin{aligned} &(((SS)(SS))S)(SS) \rightarrow (((SS)((SS)S))(SS)) \rightarrow \\ &((S(SS))(((SS)S)(SS))) \rightarrow ((S(SS))((S(SS))(S(SS)))) \end{aligned}$$

2. Предложите эффективную структуру данных для представления S -термов, облегчающую применение S -правила. Написать две процедуры: «readterm» и «printterm». Первая из них преобразует S -термы в вашу структуру данных из формы, порождаемой процедурой «gensterm»; вторая преобразует S -термы из вашей структуры в форму, порождаемую процедурой «gensterm». Ваша программа должна демонстрировать эти преобразования.

3. Напишите процедуру «reduce», выполняющую один шаг редукции в соответствии с S -правилом над заданным подтермом S -терма в вашем представлении. Программа должна продемонстрировать это.

4. Напишите процедуру «normalize», которая в заданном S -терме должна последовательно выбирать подтермы и применять S -правило до тех пор, пока дальнейшие редукции станут невозможными, либо число шагов достигнет некоторого максимума, например 30. Программа должна продемонстрировать это.

5. Объедините все процедуры в одну программу, которая:
- а) запрашивает у пользователя длину n ;
 - б) порождает с помощью процедуры «gensterm» все S -термы заданной длины;
 - в) преобразует эти S -термы в ваше представление;
 - г) нормализует их (если это возможно);
 - д) выводит в качестве результата нормализованные S -термы;
 - е) выводит последовательно число шагов редукции, совершенных над каждым S -термом, либо сообщение «not normalized», если нормализация требует более 30 шагов;
 - ж) выводит число ненормализованных термов и общее число всех S -термов заданной длины n .

Оценка решения:

- Пункт 1 — 20 баллов.
- Пункт 2 — 25 баллов.
- Пункт 3 — 15 баллов.
- Пункт 4 — 20 баллов.
- Пункт 5 — 10 баллов.
- Жюри — 10 баллов (стиль программирования, элегантность).

Обсуждение задачи

В дальнейшем количество букв «S» в S -терме мы будем называть *порядком* S -терма. Заметим, что общее число символов «S» и скобок в S -терме порядка n равно $3 \cdot n - 2$.

Будем последовательно генерировать S -термы порядков 1, 2, 3, ..., n . S -термы одного порядка свяжем в список. Для этого можно использовать следующую структуру данных:

```
Type ptr = ^sterm;
      sterms = Record
          txt    : String[3*nMax-2];
          next   : ptr;
      End;
```

Поле `txt` этой структуры будет хранить символьное представление S -терма, а поле `next` — указывать на запись для следующего S -терма того же порядка. Кроме того, заведем массив для хранения указателей на первые элементы списков, соответствующих S -термам различных порядков.

Объясним работу процедуры `GenSterm` на примере генерации S -термов шестого порядка. Число 6 может быть получено как сумма следующих пар чисел: 1+5, 2+4, 3+3, 4+2, 5+1. Значит, взяв S -терм первого порядка и объединяя его последовательно с S -термами пятого порядка по заданному в условии задачи

правилу, мы получаем все S -термы шестого порядка. Далее следует перебирать поочередно все S -термы порядков 2, 3, 4, 5 и к каждому из них присоединять всевозможные S -термы порядков 4, 3, 2, 1 соответственно. Таким образом будут получены все S -термы шестого порядка.

Схематично генерацию S -термов можно описать так:

```
Procedure GenSterm; {n - глобальная переменная}
Var len : Integer;
Begin
  For len:=1 To n Do Begin
    <Открываем новый файл для S-термов длины len>;
    If len=1 Then Begin
      <Создаем список из единственного S-терма
        порядка 1>;
      <Записываем его в файл>;
    End
    Else Begin
      <Инициализируем новый список>;
      For i:=1 To len Do
        For <всех S-термов M порядка i> Do
          For <всех S-термов N порядка len-i> Do
            Begin
              <Объединяем два S-терма в один новый (MN)>;
              <Добавляем его в конец списка>;
              <Записываем его в файл>;
            End;
          End;
        <Закрываем файл для S-термов длины len>;
      End;
    End;
  End;
```

Методическое отступление. «Ручная прокрутка логики» для конкретного значения n , например, для $n = 5$ позволит уяснить суть операции объединения S -термов.

Для решения второго пункта задания заметим, что вовсе не обязательно создавать какую-либо свою структуру данных: для успешного выполнения редукции вполне достаточно хранить S -терм в переменной типа String. Единственным недостатком такого представления данных является возможность переполнения переменной типа String в результате выполнения нескольких шагов редукции.

Опишем альтернативную структуру данных для представления S -термов. По всей видимости, наилучшим образом S -терм может быть представлен в виде *двоичного дерева*. Например,

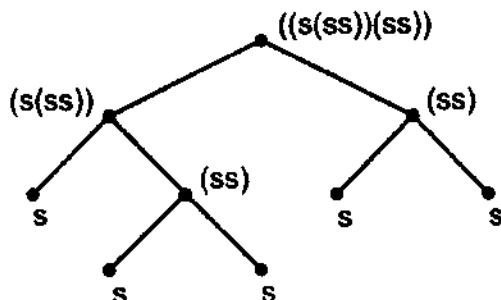


Рис. 3.2. Пример представления S-терма в виде двоичного дерева

S-терм $((S(SS))(SS))$ соответствует двоичному дереву, показанному на рис. 3.2.

Каждый узел этого дерева представлен двумя указателями — на левое и правое поддеревья. Для «листьев» дерева оба этих указателя равны nil. Тогда на Паскале структуру «узел дерева» можно описать следующим образом:

```

Type ptr = ^stern;
      stern = Record
          fir : ptr;
          sec : ptr;
      End;
  
```

Переопределим стандартную процедуру New для начальной инициализации S-термов:

```

Procedure New(Var sp : ptr);
Begin
    System.New(sp);
    sp^.fir:=nil;
    sp^.sec:=nil;
End;
  
```

Опишем действия процедур ReadTerm и PrintTerm. Процедура ReadTerm должна преобразовывать S-термы из символьной формы в вид двоичного дерева. Ее примерная схема:

```

Procedure ReadTerm(S : String; Var sp : ptr);
{По строке S создаем дерево, указатель на корень
 которого помещаем в sp}
Begin
    New(sp);
    If S<>'S' Then Begin
  
```

```

{В этом случае S='('+M+N+')'}
<Выделяем из строки S подтермы M и N>;
ReadTerm(M, sp^.fir);
{Строим дерево для 1-го подтерма}
ReadTerm(N, sp^.sec);
{Строим дерево для 2-го подтерма}
End;
End;

```

Процедура PrintTerm, обеспечивающая вывод элементов двоичного дерева в виде строки символов, выглядит совсем просто:

```

Procedure PrintTerm(sp : ptr);
{Выводим элементы дерева, на которое указывает sp}
Begin
  If sp^.fir=nil Then Write('S')
  Else Begin
    Write('(');
    PrintTerm(sp^.fir); {Выводим 1-й подтерм}
    PrintTerm(sp^.sec); {Выводим 2-й подтерм}
    Write(')');
  End;
End;

```

Методическое отступление. Программу реализации описанных преобразований S-термов можно дополнить вводом изображения дерева на экране с помощью клавиатуры и выводом соответствующего ему символьного представления, а также обратным преобразованием — построением дерева из символьного представления.

Опишем редукцию S-термов, представленных в символьной форме. К подтерму можно применить S-правило тогда и только тогда, когда он начинается с цепочки символов ((S. Первое вхождение этой цепочки в строку S, предназначенную для хранения S-терма, можно найти с помощью вызова функции Pos('((S', S). Если она вернет значение 0, то к данному S-терму нельзя применить S-правило; в противном случае из строки S требуется поочередно выделить подтермы A, B, C и затем заменить подстроку (((SA)B)C) на подстроку ((AC)(BC)). Выделение подтерма из строки S, начиная с заданной позиции i, осуществляется с помощью функции SubTerm:

```

Function SubTerm(i : Integer) : String;
  Var t : String;
      k : Integer;

```

```

Begin
  t:='';
  k:=0; {k - баланс скобок}
  Repeat
    t:=t+S[i];
    Case S[i] Of
      '(' : k:=k+1;
      ')' : k:=k-1;
    End;
    i:=i+1;
  Until k=0;
  SubTerm:=t;
End;

```

Редукция S -термов, представленных в виде дерева, осуществляется сложнее. Пусть sp — указатель на корень поддерева, соответствующего подтерму, к которому применяется S -правило. Применение S -правила возможно, если выполняются следующие условия:

```

sp^.fir <> nil
sp^.fir^.fir <> nil
sp^.fir^.fir^.fir <> nil
sp^.fir^.fir^.fir^.fir = nil

```

Методическое отступление. Графическое изображение выполнения условий применимости S -правила позволяет осознать этот сложный тип адресации к данным.

При выполнении условий применимости S -правила редукция осуществляется с помощью процедуры Reduce:

```

Procedure Reduce(sp : ptr);
  Var a,b,c : ptr;
  Begin
    a:=sp^.fir^.fir^.sec;
    b:=sp^.fir^.sec;
    c:=sp^.sec;
    New(sp^.sec);
    sp^.fir^.fir:=a;
    sp^.fir^.sec:=c;
    sp^.sec^.fir:=b;
    sp^.sec^.sec:=c;
  End;

```

Методическое отступление. Здесь графическая прорисовка действий процедуры просто обязательна! Дальнейшее обсуждение задачи без этой работы малопродуктивно.

Полученная структура данных уже не является деревом, так как вершина, на которую указывает *c*, является сыном одновременно для двух других вершин; тем не менее, процедура *PrintTerm* по-прежнему работает корректно. Чтобы полученная структура оставалась деревом (что необходимо для решения задачи по пунктам 4 и 5), необходимо получить «копию» (процедура *CopyTree*) того поддерева, на которое указывает *c*:

```
Procedure CopyTree(s : ptr; Var t : ptr); {t := копия s}  
Begin  
  If s=nil Then t:=nil  
  Else Begin  
    New(t);  
    CopyTree(s^.fir,t^.fir);  
    CopyTree(s^.sec,t^.sec);  
  End;  
End;
```

С учетом сказанного, последние строки процедуры *Reduce* модифицируются так:

```
...  
  New(sp^.sec);  
  sp^.fir^.fir:=a;  
  sp^.fir^.sec:=c;  
  CopyTree(c,cl);  
  sp^.sec^.fir:=b;  
  sp^.sec^.sec:=cl;  
End;
```

Решение пункта 4 задачи мы оформим в виде процедуры *Normalize*:

```
Procedure Normalize(sp : ptr);  
{Нормализуем дерево, на которое указывает sp}  
  Const Max = 30; {Максимальное количество редукций}  
  Var cnt : Integer;  
      ok : Boolean;  
  Begin  
    cnt:=0;  
    Repeat  
      cnt:=cnt+1;  
      PrintTerm(sp);  
      Write('    -> ');  
      ok:=True;
```

```

    <Ищем подтерм, к которому можно применить
S-правило>;
    If <нашли> Then Begin
        <Применяем S-правило>;
        ok:=False;
    End;
    Until (cnt=Max+1) Or ok;
    If ok Then WriteLn(cnt-1, ' steps needed')
    Else WriteLn('not normalized');
End;

```

Методическое отступление. До написания конечного программного кода следует обработать вручную ряд примеров для произвольных *S*-термов по выполнению редукции до тех пор, пока это возможно.

Вместо генерации *S*-термов в символьном виде и их последующего перевода в форму двоичного дерева возможно также и другое решение — хранение *S*-терма в требуемом виде. Для этого следует модифицировать структуру данных для представления *S*-терма. Необходимо ввести поле *next* типа *ptr*, с помощью которого связываются *S*-термы одного порядка. Тогда каждому *S*-терму будет соответствовать запись, структура которой показана на рис. 3.3 и в следующем формализованном виде:

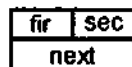


Рис. 3.3. Структура записи, соответствующая *S*-терму

```

Type ptr = ^sterm;
sterm = Record
    fir, sec : ptr;
    next : ptr;
End;

```

Схема процедуры *GenSterm* (пункт 1 формулировки задачи) при этом не изменяется. Изменится лишь реализация шага <Объединяем два *S*-терма *M* и *N* в один новый (*MN*)>: если раньше при этом создавалась новая запись типа *sterm* — и ее полю *txt* присваивалась конкатенация полей *txt* подтермов *M* и *N*, то при данной реализации полю *fir* мы присваиваем адрес подтерма *M*, а полю *sec* — адрес подтерма *N*. Фрагмент обсуждаемой структуры данных (*beg* — массив для хранения адресов первых элементов списков) приведен на рис. 3.4. Напомним, что перед нормализацией *S*-терма порядка *n* необходимо получить его копию в виде дерева с помощью вызова процедуры *CopyTree*.

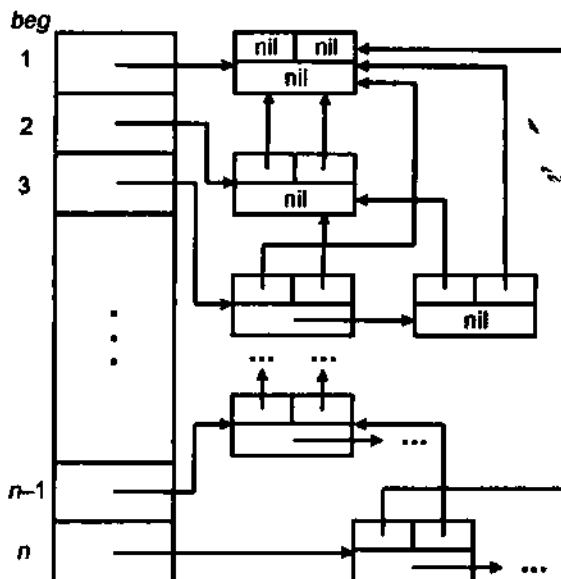


Рис. 3.4. Фрагмент структуры данных для описания множества S-термов

Данные об общем количестве S-термов различных порядков приведены в табл. 3.2.

Таблица 3.2

Длина	1	2	3	4	5	6	7	8	9	10
Количество S-термов	1	1	2	5	15	42	132	429	1430	4862
Количество ненормализованных S-термов	—	—	—	—	—	—	2	41	284	1506

Числа, записанные во второй строке табл. 3.2, — это ни что иное, как числа Э. Каталана, вычисляемые по формуле:

$$K_n = \frac{C_{2n}^n}{n+1}.$$

Методические указания

Задача сложная, и ей должно предшествовать изучение темы «Динамические структуры данных»^{*}. Однако эту задачу следует

^{*} Окулов С. М. Основы программирования. М.: БИНОМ. Лаборатория знаний, 2005. С. 346–436.

проработать, ибо ее идейная основа получила дальнейшее развитие в различных олимпиадных задачах*.

Самостоятельные исследования

1. Продолжите заполнение табл. 3.2 настолько, насколько это возможно.
2. Изучите проблему генерации чисел Э. Каталана** и установите ее связь с обсуждаемой задачей.

* Андреева Е. В., Гуровиц В. М., Матюхин В. А. Московские олимпиады по информатике. М.: МЦНМО, 2006; Скиена С. С., Ревилла М. А. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. М.: КУДИЦ-ОБРАЗ, 2005.

** Окулов С. М. Дискретная математика (теория и практика решения задач по информатике). М.: БИНОМ. Лаборатория знаний, 2007.

ЗАДАЧА ПЕРВОГО ТУРА: «ОСТРОВА В МОРЕ»

Условие задачи

Расположение островов в море представлено с помощью сетки размером $n \times n$. Каждый остров обозначается символом «*» в узле этой сетки. Задача заключается в том, чтобы восстановить карту морских островов по закодированной информации о распределении островов по горизонталям и вертикалям.

Для иллюстрации принципа кодирования рассмотрим следующую карту и соответствующие ей коды (рис. 4.1). Числа справа от карты на этом рисунке являются кодами и представляют порядок и размер групп островов в соответствующих горизонталях сетки. Например, цифры «1 2» в первой строке означают, что первая горизонталь содержит группу из одного острова, за которой следует группа из двух островов. Слева и справа от каждой группы островов расположено море произвольной протяженности. Аналогично, последовательность «1 1 1» в первом столбце под картой островов означает, что первая вертикаль содержит три группы островов, в каждой из которых один остров, и так далее.

*		*	*			1 2
	*	*	*		*	3 1
*		*		*		1 1 1
	*	*	*	*	*	5
*	*		*		*	2 1 1
			*			1
1	1	4	2	2	1	
1	2		3		2	
1						

Рис. 4.1. Пример карты и соответствующих ее кодов

Постановка задачи. Написать программу, которая выполняет следующие действия (шаги) до тех пор, пока данный входной файл, содержащий несколько независимых блоков информации, не будет прочитан полностью:

1. Считывает очередной блок информации из входного ASCII-файла (структура этого файла приведена в примере 1) и отображает его на экране.

Пример 1:

```

6
1 2 0      — строка для первой горизонтали
3 1 0
1 1 1 0
5 0
2 1 1 0
1 0
1 1 1 0    — строка для первого столбца
1 2 0
4 0
2 3 0
2 0
1 2 0

```

Пример 2:

Блок входной информации	Решение
4	Столбцы 1 2 3 4
0	Строка 1
1 0	Строка 2 *
2 0	Строка 3 * *
0	Строка 4
0	
1 0	
2 0	
0	

Пример 3:

```

2
0
0
0
2 0
2 0

```

Для этих данных не существует карты.

Пример 4:

```

2
1 0
1 0
1 0
1 0

```

Этим данным удовлетворяют две различные карты.

Каждый блок информации начинается с размеров квадратной сетки, за которыми следует кодовая информация о горизонтальном и вертикальном распределении островов. Код для каждой отдельной горизонтали и вертикали является отдельной строкой файла и представляет собой последовательность чисел, разделенных пробелами. Заканчивается каждая строка нулем. При этом сначала идет информация обо всех горизонталях, а затем — обо всех вертикалях.

2. Восстанавливает карту (или все карты в случае не единственного решения, как в примере 4) и отображает ее (их) на экране.

3. Добавляет карту (карты) в конец выходного ASCII-файла. Каждое пустое место в сетке должно быть представлено двумя пробелами, а каждый остров — символом «*» (звездочка и пробел). В случае неоднозначного решения различные карты должны быть отделены друг от друга пустой строкой. Если карту восстановить невозможно, в соответствующей строке выходного файла написать «no map». Решения, относящиеся к различным блокам информации входного файла, должны быть отделены в выходном файле строкой с надписью «next problem».

Технические ограничения:

1. n должно быть не меньше 1 и не больше 8.
2. Результирующая программа должна быть помещена в текстовый ASCII-файл с именем C:\IOI\DAY-1\413-PROG.xxx. Расширение .xxx для PASCAL — .PAS, для C — .C, для BASIC — .BAS, для LOGO — .LOG.
3. Имя входного ASCII-файла должно быть C:\IOI\DAY-1\413-SEAS.IN.

Идея решения

Введем понятие *символьного кода* строки и столбца. Из строки (столбца) карты удалим лишние пробелы; — другими словами, сделаем так, чтобы после каждой группы островов (в том числе последней) шел в точности один пробел. Все начальные пробелы, находящиеся перед самой первой группой островов, также удаляются. Получившуюся при этом цепочку символов мы назовем символьным кодом взятой строки (столбца). Например, символьный код второй строки карты, приведенной в условии задачи, равен «*****_».

Коды по строкам и по столбцам, считанные из файла, преобразуем в символьные коды и запишем в матрицы *hor* и *ver* соответственно. На рис. 4.2 показано это преобразование.

Восстановление карты осуществляется слева направо и сверху вниз. При этом используется стандартный прием решения задач такого рода — *перебор с возвратом (backtracking)*.

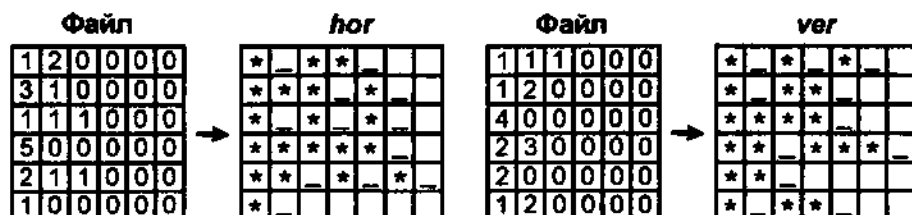


Рис. 4.2. Преобразование данных из файла в массив символьных кодов

Введем массивы *CurHor* и *CurVer* для хранения текущих (символьных) кодов всех строк и столбцов, т. е. кодов уже заполненных их частей. Их использование повышает наглядность обсуждения.

Рекурсивный алгоритм *Rec*(*i*, *j*:Integer) пытается заполнить клетки карты, начиная с клетки (*i*, *j*), всеми возможными способами. Первоначально в клетку (*i*, *j*) записывается символ «*», а затем — «_». В обоих случаях текущие коды *i*-й строки и *j*-го столбца модифицируются. Если новые текущие коды не противоречат кодам, считанным из файла, вызывается процедура *Rec*(*i*, *j*+1) (*j* < *n*) или *Rec*(*i*+1, 1) (*j* = *n*, *i* < *n*), либо найденная карта проверяется на предмет совпадения текущих кодов с заданными (*i* = *j* = *n*). Причем в последнем случае коды типа «***_*» и «***_*_» (см. ранее показанный пример) считаются «равными».

Методическое отступление. «Ручная» трассировка логики алгоритма *Rec* (при *n* ≤ 5) с использованием одного из возможных способов реализации (см. далее по тексту) позволяет достичь ясного понимания школьником сути метода перебора с возвратом.

```
{Максимальное значение n}
Const NMax = 8;
{Глобальные переменные}
Var {Размер матрицы}
    n : Integer;
    {Символьные коды строк и столбцов}
    hor, ver,
    {Текущие символьные коды}
    CurHor, CurVer : Array [1..NMax] Of String [NMax+1];
    {Массив для описания восстанавливаемой карты}
    Map : Array [1..NMax, 1..NMax] Of Char;
    {Некоторые вспомогательные процедуры и функции}
Procedure Add(Var s : String; ch : Char);
Begin {Добавляет к строке s символ ch}
```

```
s:=s+ch;
End;
Procedure Del(Var s : String);
Begin {Удаляет из s последний символ}
  s[0]:=s[0]-1; {В нулевом байте переменной
                типа String хранится длина строки}
End;
Function Head(Var s,t : String) : Boolean;
Begin {Проверяет, является ли s началом t}
  Head:=(s=Copy(t,1,Length(s)));
End;
Procedure Rec(i,j : Integer);
Var qi,qj : Boolean;
Begin
  If i>n Then <Проверить карту; если она правильная,
  то вывести ее>
  Else If j>n Then Rec(i+1,1)
  Else Begin
    Map[i,j]:='*';
    Add(CurHor[i],'*');
    Add(CurVer[j],'*');
    If Head(CurHor[i],hor[i]) And
      Head(CurVer[j],ver[j])
    Then Rec(i,j+1);
    Del(CurHor[i]);
    Del(CurVer[j]);
    Map[i,j]:=' ';
    qi:=(j>1) And (Map[i,j-1]='*');
    qj:=(i>1) And (Map[i-1,j]='*');
    If qi Then Add(CurHor[i],' ');
    If qj Then Add(CurVer[j],' ');
    If Head(CurHor[i],hor[i]) And
      Head(CurVer[j],ver[j])
    Then Rec(i,j+1);
    If qi Then Del(CurHor[i]);
    If qj Then Del(CurVer[j]);
  End;
End;
```

Заметим, что в окончательном (оптимизированном) варианте программного кода строковые массивы *CurHor* и *CurVer* можно исключить. В них нет необходимости, поскольку их элементы являются «началами» соответствующих элементов массивов *hor* и *ver*, и поэтому достаточно хранить лишь длины этих «начал».

Методические рекомендации

Данная задача своей наглядностью украсит изучение темы «Перебор с возвратом».

Самостоятельные исследования

Задача сформулирована для очень небольших значений n . Найдите вариант решения для значений $n \leq 50$. При этом требуется составить таблицу, содержащую значение n ($n = 5, 8, 10, 15, 20, \dots$) и время работы компьютера по вашей программе. Заметим, что во многих изданиях сегодня публикуются «японские кроссворды», разгадка которых осуществляется на этих же принципах, с $n = 50$, и человек справляется с ними без особого труда.

ЗАДАЧА ВТОРОГО ТУРА: «ПОКОРЕНИЕ ВЕРШИНЫ»

Условие задачи

Клуб альпинистов состоит из p членов, с номерами от 1 до p . Каждый альпинист поднимается в гору с одной и той же скоростью, а скорость подъема не отличается от скорости спуска. Альпинист с номером i расходует $c(i)$ единиц ресурсов в день как при подъеме, так и при спуске и может нести в каждый момент времени не больше $s(i)$ таких единиц.

Предполагается, что альпинист может достичь вершины за n дней при полной обеспеченности ресурсами как для подъема, так и для спуска. Гора может быть так высока, что один альпинист не сможет изначально нести все необходимые для подъема и спуска ресурсы. Поэтому группа альпинистов стартует в одном и том же месте и в одно и то же время, чтобы обеспечить восхождение. Альпинист может начать спускаться, не достигнув вершины, отдав при этом все ненужные ему для спуска ресурсы другим альпинистам, которые должны быть в состоянии их взять. Альпинисты не отдыхают в течение экспедиции.

Задача заключается в составлении расписания восхождения для клуба альпинистов. По крайней мере один альпинист должен достичь вершины горы, и все альпинисты, включенные в группу восхождения, должны возвратиться в начальную точку, израсходовав все ресурсы.

Постановка задачи. Написать программу, которая выполняет следующее:

1. Вводит с клавиатуры число дней n , требуемое для достижения вершины горы, число p членов клуба и числа $s(i)$, $c(i)$ для всех i от 1 до p .

Все входные данные целочисленные, вещественные данные вводиться не будут. Повторить запрос на ввод данных, не имеющих смысла.

2. Определяет расписание для восхождения, а также номера альпинистов $a(1), a(2), \dots, a(k)$, образующих группу для восхождения, и числа $m(j)$ (для всех j от 1 до k), которые обозначают количество единиц ресурсов, взятых каждым альпинистом на старте.

Сообщает, если нельзя составить такого расписания для заданных $n, s(i)$ и $c(i)$.

3. Выводит следующую информацию на экран:

- а) число k альпинистов, участвующих в восхождении (размер группы восхождения);
- б) необходимое для этого общее количество единиц ресурсов;
- в) номера участвующих в восхождении альпинистов $a(1), a(2), \dots, a(k)$;
- г) для каждого $a(j)$ — какое количество единиц ресурсов он возьмет с собой со старта;
- д) для каждого $a(j)$ — количество дней, через которое он начнет спускаться.

4. Найденное расписание должно быть близким к оптимальному. Расписание является оптимальным, если:

- а) число участвующих в восхождении альпинистов, необходимое для достижения вершины одним из них, минимально;
- б) среди всех расписаний для групп, удовлетворяющих условию а), общее количество единиц ресурсов, взятых с собой со старта, минимальное.

Технические ограничения:

1. Поместите вашу результирующую программу в текстовый ASCII-файл с именем `C:\IOI\DAY-2\422-PROG.xxx`. Расширение `.xxx` для *PASCAL* — `.PAS`, для *C* — `.C`, для *BASIC* — `.BAS`, для *LOGO* — `.LOG`.
2. Программа не должна воспринимать входные данные, если $n < 1$ или $n > 100$, а также $p < 1$ или $p > 20$.

Пример. Возможен следующий диалог с вашей программой:

Число дней, необходимое для достижения вершины, — 4.

Число альпинистов в клубе — 5.

Максимальный ресурс для альпиниста 1 — 7.

Ежедневный расход ресурса для альпиниста 1 — 1.

Максимальный ресурс для альпиниста 2 — 8.

Ежедневный расход ресурса для альпиниста 2 — 2.

Максимальный ресурс для альпиниста 3 — 12.
 Ежедневный расход ресурса для альпиниста 3 — 2.
 Максимальный ресурс для альпиниста 4 — 15.
 Ежедневный расход ресурса для альпиниста 4 — 3.
 Максимальный ресурс для альпиниста 5 — 7.
 Ежедневный расход ресурса для альпиниста 5 — 1.
 2 альпиниста требуются для покорения вершины.
 10 единиц ресурса в целом для этого необходимо.
 Взбираться будут альпинисты 1, 5.
 Альпинист 1 возьмет 7 единиц ресурса и начнет спускаться через 4 дня.
 Альпинист 5 возьмет 3 единицы ресурса и начнет спускаться через 1 день.
 Хотите спланировать восхождение для другого альпинистского клуба? (Y/N) — Y.
 Число дней, необходимое для достижения вершины, — 2.
 Число альпинистов в клубе — 1.
 Максимальный ресурс для альпиниста 1 — 3.
 Ежедневный расход ресурса для альпиниста 1 — 1.
 Восхождение невозможно.
 Хотите спланировать восхождение для другого альпинистского клуба? (Y/N) — N.
 До свидания.

Идея решения

Согласно пункту 4 условия задачи, нам требуется найти минимальное k , такое, что существуют группы из k альпинистов, способные осуществить восхождение, либо установить, что такого k не существует. Если k найдено, то из всех групп размера k , способных осуществить восхождение, выбирается та, для которой необходимо минимальное количество ресурсов.

Введем следующие типы данных:

```

Type AlpSet = Set Of 1..p;
AlpArr = Array [1..p] Of Integer;
  
```

Исходными данными для нашей программы являются целые числа n и p , а также массивы c и s типа AlpArr. Расписание восхождения можно полностью описать тремя переменными: необходимым числом ресурсов (переменная типа Integer), группой восхождения (переменная типа AlpSet) и массивом, i -й элемент которого указывает, через сколько дней после начала подъема начнет спускаться альпинист с номером i . Элементы этого массива формируются, естественно, только для альпинистов, участвующих в восхождении (переменная типа AlpArr).

Для хранения наилучшего из найденных расписаний введем следующую тройку переменных:

```
Var res : Integer;
    MinM : AlpSet;
    d : AlpArr;
```

Функция подсчета суммы элементов массива a с номерами из множества m будет полезна в дальнейшем обсуждении:

```
Function Sum(m:AlpSet;a:AlpArr):Integer;
  Var i,cnt : Integer;
  Begin
    cnt:=0;
    For i ∈ m Do cnt:=cnt+a[i];
    Sum:=cnt;
  End;
```

Например, $\text{Sum}(m, s)$ возвращает суммарный ресурс группы альпинистов с номерами из множества m , который они могут взять с собой, а с помощью вызова $\text{Sum}(m, c)$ подсчитывается количество единиц ресурсов, расходуемых той же группой за один день.

Заметим, что если в процессе перебора (а «идейной основой» задачи здесь вновь является перебор с возвратом) уже найдено хотя бы одно расписание, то имеет смысл искать лишь те расписания, которые требуют менее res ресурсов.

Предположим, что у нас уже имеется функция с заголовком **Function** Расписание(m : AlpSet) : Boolean, которая для заданной группы альпинистов m пытается составить (наилучшее) расписание восхождения, требующее менее res ресурсов. Если это невозможно, данная функция возвращает False; в противном случае функция возвращает True, а тройка переменных (res , M и d) при выходе из нее описывает найденное (более оптимальное) расписание.

Формализованная запись общего алгоритма решения задачи имеет вид:

```
k:=0;
res:=MaxInt; {MaxInt — максимальная константа типа
Integer, равна 32767}
For k:=1 To p Do Begin
  For <всех  $m \subset [1..p]$ ,  $|m|=k$ > Do {Рассматриваются
все подмножества мощности k}
    If Расписание( $m$ ) Then MinM:=m;
```

```

    If resMaxInt Then Break;
End;
If res=MaxInt Then WriteLn('Восхождение невозможно')
Else <Вывести оптимальное расписание (res, MinM, d)>;

```

Что мы здесь делаем? Мы рассматриваем все группы из одного альпиниста, затем все группы из двух альпинистов и т. д. Если хотя бы для одной группы из k альпинистов восхождение возможно, то перебор групп из большего количества альпинистов не производится.

Методическое отступление. «Ручной» расчет примеров хотя и утомителен для данной задачи, но может оказаться весьма целесообразным для понимания логики перебора.

Осталось обсудить логику работы функции Расписание. Пусть $n > 1$. После первого дня восхождения, когда уже израсходовано $res_1 = \text{Sum}(m, c)$ ресурсов, требуется выбрать подмножество $w \subset m$ ($w \neq m$) альпинистов, возвращающихся обратно (w , в частности, может быть пустым). При этом определенное количество ресурсов $res_2 = \text{Sum}(w, c)$ потребуется израсходовать на их спуск. Если для оставшейся группы альпинистов $(m - w)$ имеется расписание (наилучшее) дальнейшего восхождения, требующее менее чем $res - res_1 - res_2$ ресурсов, то у нас есть и ответ на первоначальный вопрос о восхождении группы из m альпинистов: следует перебирать все возможные подмножества w и выполнять указанную выше оценку. Тогда нами будет получена задача, аналогичная первоначальной, но с меньшим (на единицу) числом дней восхождения до вершины, что говорит о рекурсивном характере алгоритма функции Расписание.

Один из возможных способов реализации этой функции имеет вид:

```

Function Rec(m : AlpSet) : Boolean; {В глобальной
переменной t хранится текущий номер дня (t=0 в
начале работы программы)}
  Var w : AlpSet;      {Группа возвращающихся}
      ok : Boolean;    {Признак: улучшили ли мы
расписание?}
      was, res2, i : Integer;
Begin
  ok:=False;
  was:=res;
  {Может ли группа нести res-1 ресурсов?}
  If res>Sum(m,s) Then res:=Sum(m,s)+1;
  {Имеем res-1 <= Sum(m,s)}

```

```

t:=t+1;
res:=res+Sum(m,c); {Наступил новый день}
If res>Sum(m,c)*t Then {Все должны вернуться}
  If t=n Then Begin {Альпинисты уже на вершине}
    ok:=True; {Улучшили решение}
    For i ∈ m Do d[i]:=t;
    res:=Sum(m,c)*n; {Необходимо для спуска}
  End
Else Begin
  For W ⊂ M Do Begin
    res2:=Sum(w,c)*t; {Требуется для спуска
группы из w альпинистов}
    If Sum(w,s)>=res2 Then Begin {Могут нести?}
      res:=res-res2; {Израсходовали на спуск}
      If Rec(m-w) Then Begin
        ok:=True; {Найдено лучшее решение}
        For i ∈ W Do d[i]:=t;
      End;
      res:=res+res2; {Добавили израсходованное}
    End;
  End;
End;
t:=t-1;
rs:=res+Sum(m,c); {Вернулись к предыдущему дню}
If Not ok Then res:=was; {Восстановили res}
Rec:=ok;
End;

```

Методические указания

Это — задача по теме «Перебор с возвратом», она относится скорее к исследовательскому типу. Для более полного использования ее дидактического потенциала следует продумать и разработать, как минимум, генератор тестов и проверяющую программу.

Самостоятельные исследования

Как и следовало ожидать, алгоритм для больших значений n и p работает достаточно медленно. Нахождение способов сокращения перебора традиционно является проблемой для исследования. «Набросаем» возможные пути поиска решения:

1. *Нахождение хорошего начального решения.* Если каким-либо эвристическим способом получено хорошее начальное приближение к оптимальному решению (требуемое количество ресурсов записано в переменную *res*) или просто найдена хоро-

шая оценка сверху на минимально необходимое количество ресурсов, то это позволяет перебирать гораздо меньше «ветвей» дерева вариантов.

2. *Перебор с предпочтениями.* Упорядочим альпинистов по убыванию величины $s[i]/c[i]$ («выносливости») или по какому-либо другому разумному критерию. Тогда группы с высокой суммарной выносливостью следует рассмотреть в первую очередь. При этом чем раньше алгоритм находит оптимальное решение, тем больше «ветвей» отсекается в процессе перебора.
3. *Двоичный поиск значения k .* Если значение k (количество альпинистов в группе) достаточно велико или группы, обеспечивающей восхождение, не существует (т. е. невозможно ни для одной группы составить расписание), то логику работы с k целесообразно организовать на основе двоичного поиска.

ЗАДАЧИ ПЕРВОГО ТУРА

«Бусинки»

Условие задачи

Имеются бусы, состоящие из n ($n \leq 100$) бусинок, некоторые из которых красного или голубого цвета, а остальные — белые. На рис. 5.1 приведены два примера бус для $n = 29$ (цифрами отмечены позиции первой и второй бусинок, а также четырнадцатой, пятнадцатой и шестнадцатой бусинок).

Конфигурация бус задается последовательностью цветов бусинок («b» — голубая, «r» — красная, «w» — белая), начиная с бусинки номер 1. Например, бусы на рис. 5.1а задаются последовательностью: brbrrrrbbbrrrrrbrbrbbrbbbrrrrrb.

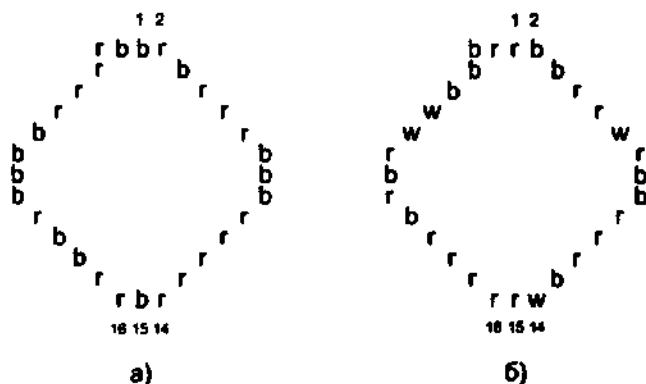


Рис. 5.1. Примеры расположения бусинок:

а) brbrrrrbbbrrrrrbrbrbbrbbbrrrrrb; б) rbbrrrrbbrrrrbwrrrrrbbrwbbbr, где «r» — красная бусинка, «b» — голубая бусинка, «w» — белая бусинка

Порвем бусы и затем начнем снимать бусинки одного цвета с первого конца, пока не встретится бусинка другого цвета. То же самое сделаем со вторым концом (бусинки, снятые с разных концов, могут быть разного цвета). Требуется определить точку такого разрыва данных бус, при котором суммарное количество бусинок, собранных с обоих концов, максимально. Например, для бус на рис. 5.1а точка разрыва может находиться между 24-й и 25-й бусинками или между 9-й и 10-й бусинками; при этом суммарное количество бусинок в обоих случаях равняется 8.

При снятии бусинок с каждого из концов белая бусинка рассматривается как бусинка голубого или красного цвета по ситуации, то есть может сниматься как с голубыми, так и с красными.

Напишите программу, которая:

1. Вводит данные из входного *ASCII*-файла с именем *NECKLACE.DAT*, каждая строка которого содержит конфигурацию бус, заданную в виде последовательности цветов, и записывает входные данные в выходной *ASCII*-файл с именем *NECKLACE.SOL*.

Пример входного файла *NECKLACE.DAT*:

```
brbrrrrbbbrrrrrbrbrbbrbbbrrrrb
bbwbrrrrwbrbrrrrrb
```

2. Для каждой конфигурации бус определяет m — максимальное число собранных бусинок и положение одной из оптимальных точек разрыва.
3. Выводит в качестве результата в выходной файл с именем *NECKLACE.SOL* число m и точку разрыва. Ответы для разных конфигураций отделяются пустой строкой.

Пример выходного файла *NECKLACE.SOL*:

```
brbrrrrbbbrrrrrbrbrbbrbbbrrrrb
8 between 9 and 10
bbwbrrrrwbrbrrrrrb
10 between 16 and 17
```

Идея решения

Идея решения этой задачи достаточно проста. Необходимо проверить все возможные точки разрыва и для каждой из них определить количество снимаемых бусинок. В символьном массиве A ($A : \text{Array}[1..3*n] \text{ Of Char}$) мы будем хранить утраченную конфигурацию бус, что поможет избежать трудностей при переходе через точку «склейки» бус, т. е. от бусинки n к бусинке 1 и наоборот. Например, бусам, изображенным на рис. 5.2а, будет соответствовать символьный массив A , приведенный на рис. 5.2б.

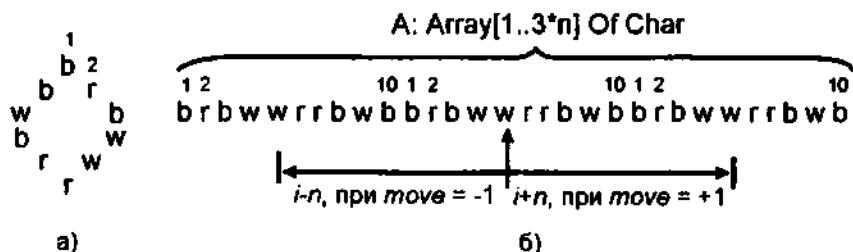


Рис. 5.2. Пример расположения бусинок и его описание в памяти компьютера

Рассмотрим точку разрыва, расположенную между бусинками с номерами k и $k+1$ ($1 \leq k < n$) или между бусинками n и 1 ($k=n$), и положим, что $i = k+1$. Возьмем i -й элемент массива A и будем двигаться от него влево, «снимая» бусинки, пока это возможно. Затем аналогичным образом будем двигаться от элемента $(i+1)$ вправо. Сложив количество бусинок, «снятых» при движении влево, и количество бусинок, «снятых» при движении вправо, получим общее количество снимаемых бусинок для данной точки разрыва.

Следует проанализировать также случай «вырожденного» расположения бус, когда в них совсем нет либо красных, либо голубых бусинок. При этом снимаются все бусинки, и нам необходимо предусмотреть какой-либо «барьер» для продвижения по массиву A . Этим «барьером», например, может служить проверка, что номер анализируемого элемента массива A принадлежит диапазону $[i-n+1 .. i+n-1]$, так как выход из этого диапазона как раз и означает «вырожденность» бус.

Рекурсивная функция `Number` решает задачу подсчета количества снимаемых бусинок в одном направлении для точки разрыва, соответствующей значению k . Общее же количество бусинок подсчитывается путем ее двойного вызова: Число бусинок: = `Number(k+n, -1, 'w') + Number(k+n+1, +1, 'w')`:

```
Function Number(j,move:Integer; c:Char) : Integer;
Begin
  If (j>i-n) And (j<i+n) And
    ((c='w') Or (A[j]='w') Or (A[j]=c))
  Then
    If c='w' Then Number:=1+Number(j+move,move,A[j])
              Else Number:=1+Number(j+move,move,c)
    Else Number:=0;
End;
```

В случае «вырожденных» бус значение `Число_бусинок` равно $2 * n - 1$, и его следует рассмотреть отдельно. Заметим, что вместо условия `((c='w') Or (A[j]='w') Or (A[j]=c))` можно записать более лаконичное: `([A[j],c]<>['b','r'])`.

Теперь в основной программе остается найти максимальное число снимаемых бусинок в зависимости от значения i .

Методическое отступление. Предложение по написанию наилучшего программного кода нерекурсивного способа подсчета значения `Число_бусинок` разнообразит занятие:

```
Function Number(j,move : Integer) : Integer;  
Var s : Set Of Char;  
    cnt : Integer;  
Begin  
    s:=['w'];  
    cnt:=0;  
    Repeat  
        s:=s+[A[j]];  
        j:=j+move;  
        cnt:=cnt+1;  
    Until (s=['r','b','w']) Or (j<i-n) Or (j>i+n);  
    Number:=cnt-1;  
End;
```

Методические указания

Данная задача не требует знания каких-либо специальных алгоритмов и методов программирования. Необходимо только умение работать с типами данных «массив» и «множество», поэтому в настоящее время описываемую задачу следует отнести к обычному учебному заданию.

Самостоятельные исследования

В качестве проблем для исследования можно предложить задачи, вызывающие некие ассоциации с приведенной выше, — скорее по формулировке, чем по способу решения:

1. *Ожерелье*. У моей маленькой сестры есть ожерелье, сделанное из цветных бусинок. Любые две соседние бусинки в ожерелье имеют один и тот же цвет в точке касания*.

Однажды ожерелье порвалось, и бусины разлетелись по всему полу. Можно ли собрать ожерелье по тому же принципу, что и старое, из тех бусин, что остались? И если можно, то как?

Входные данные. В первой строке n — количество оставшихся бусинок ($5 \leq n \leq 1000$). В каждой из n следующих строк — два целых числа, задающих цвета бусинок. Цвета задаются целыми числами от 1 до 50.

Выходные данные. Сообщение об отсутствии решения или n строк, каждая из которых содержит описание одной бусинки. Для любого i ($1 \leq i \leq n-1$) второе число строки i должно совпадать с первым числом строки $i+1$. Кроме того, второе число строки n должно быть равно первому числу строки 1. Если возможно несколько решений, то достаточно вывести одно из них.

* Скиена С. С., Ревилла М. А. Олимпиадные задачи по информатике. Руководство по подготовке к соревнованиям. М.: КУДИЦ-ОБРАЗ, 2005. С. 257–259.

Пример.**Входные данные:**

```
5
2 1
2 2
3 4
3 1
2 4
```

Выходные данные:

```
2 1
1 3
3 4
4 2
2 2
```

2. Даны две строки a и b . Требуется вывести строку x максимальной длины, состоящую из букв, таких, что существует перестановка x , являющаяся подстрокой перестановки a и одновременно являющаяся подстрокой перестановки b . Если несколько возможных строк x подходит под вышеописанные критерии, то выберите первую в алфавитном порядке*.

Максимальная длина каждой строки — 1000 символов.

Примеры входных и выходных данных:

```
pretty           e
women
walking          nw
down
the              et
street
```

«Акции»**Условие задачи**

Некоторые компании являются совладельцами других компаний, так как приобрели часть их акций. Например, компания Форда владеет 12% акций компании Мазда. Говорят, что компания A контролирует компанию B , если имеет место по меньшей мере одно из следующих условий:

- 1) $A = B$;
- 2) A владеет $> 50\%$ акций B ;

* Скиена С. С., Ревилла М. А. Олимпиадные задачи по информатике. Руководство по подготовке к соревнованиям. М.: КУДИЦ-ОБРАЗ, 2005. С. 86.

- 3) A контролирует k ($k \geq 1$) компаний $C(1), \dots, C(k)$, таких, что компания $C(i)$ владеет соответственно $X(i)\%$ акций компании B (для всех i от 1 до k), и $X(1)+X(2)+\dots+X(k) > 50$.

Задача, которую надо решить, заключается в следующем.

Дана последовательность троек чисел (i, j, p) , означающих, что компания i владеет $p\%$ акций компании j . Требуется определить все пары чисел (h, s) , при которых компания h контролирует компанию s . Общее число компаний не превышает 100.

Напишите программу, которая:

1. Читает из входного ASCII-файла с именем COMPANY.DAT последовательность троек (i, j, p) , где i, j и p — положительные целые числа. В файле могут содержаться последовательности троек чисел для нескольких тестов. Различные тестовые последовательности разделены пустой строкой.
2. Находит все пары чисел (h, s) , такие, что компания h контролирует компанию s .
3. Записывает в выходной ASCII-файл с именем COMPANY.SOL все обнаруженные пары чисел (h, s) , в которых h отличается от s . Пары (h, s) надо записать в последовательных строках в порядке возрастания h (каждая пара записывается в отдельной строке). Решения для различных тестовых наборов данных должны разделяться пустой строкой.

Пример.

COMPANY.DAT:

2	3	25
1	4	36
4	5	63
2	1	48
3	4	30
4	2	52
5	3	30

1	2	30
2	3	52
3	4	51
4	5	70
5	4	20
4	3	20

COMPANY.SOL:

4	2
4	3
4	5

2	3
2	4
2	5
3	4
3	5
4	5

Идея решения

Входные данные для этой задачи опишем матрицей A , элемент $A[i, j]$ которой означает, сколькими процентами акций компании j владеет (непосредственно) компания i . Суть решения задачи однозначно описывается комментариями к действиям процедуры Solve:

Procedure Solve;

Var B:Arr; {Type Arr = Array [1..NMax] Of Integer, где NMax=100 и матрица A описывается следующим образом - A : Array [1..NMax] Of Arr}
 Done : Set Of 1..NMax;
 i, j, p : Integer;
 ok : Boolean;

Begin

For i:=1 **To** n **Do** **Begin** {Просматриваем все компании}

B:=A[i]; {B - вспомогательный массив}

Done:={i}; {Done - множество компаний, которые контролируются компанией i}

Repeat

ok:=True; {ok - признак того, что нет новых компаний, которых могли бы контролировать компанией i}

For j:=1 **To** n **Do**

If Not (j In Done) **And** (B[j]>50) **Then** **Begin**

WriteLn(i, ' ', j); {Компания j контролируется}

Done:=Done+[j];

ok:=False; {Есть добавление в Done}

For p:=1 **To** n **Do** B[p]:=B[p]+A[j, p];

{Прибавляем ее акции к элементам массива B}

End;

Until ok;

End;

WriteLn;

End;

Для рассматриваемой компании i элемент массива $B[j]$ указывает, сколькими процентами акций компании j компания i

владеет непосредственно либо через контролируемые ею компании. Как только этот процент превысит 50, компания j также оказывается контролируемой, и ее акции добавляются в массив B . Заметим, что отношение контролирования *рефлексивно* (т. е. каждая компания контролирует саму себя) и *транзитивно* (т. е. из того, что A контролирует B и B контролирует C , следует, что A контролирует C).

Методическое отступление. Задание по «ручной» трассировке логики процедуры Solve по данным из формулировки задачи будет вполне уместно для осознания школьниками сути представленного алгоритма.

Методические указания

Данная задача является ни чем иным как вариацией на тему построения транзитивного замыкания бинарного отношения на некотором конечном множестве. Ее уместно использовать при изучении алгоритма поиска кратчайших путей между всеми парами вершин графа (С. Уоршалла, Р. Флойда) или при нахождении сильно связанных компонент ориентированного графа*.

Самостоятельные исследования

Решение описываемой задачи имеет временную сложность $O(n^3)$. Поиск путей уменьшения времени является предметом исследования, не только на школьном уровне.

«Прямоугольники»

Условие задачи

n прямоугольников различных цветов располагаются на белом прямоугольном листе бумаги, имеющем размеры A сантиметров в ширину и B сантиметров в длину. Стороны прямоугольников параллельны краям листа, а сами прямоугольники не выходят за пределы листа. В результате образуются различные одноцветные фигуры. Если два прямоугольника одного цвета имеют хотя бы одну общую точку, то они являются частями одной фигуры. Задача состоит в вычислении площади каждой из видимых одноцветных фигур для каждого цвета. A и B — четные положительные целые числа, не превосходящие 30.

Начало системы координат находится в центре листа, а оси параллельны краям листа.

Наборы данных для нескольких тестов записаны во входном ASCII-файле с именем RECTANG.DAT следующим образом:

* Ожухов С. М. Дискретная математика: теория и практика решения задач по информатике. М.: БИНОМ. Лаборатория знаний, 2007.

A, *B* и *n* находятся в первой строке каждого набора данных и разделены пробелом.

В каждой из следующих *n* строк находятся:

- целочисленные координаты точки, в которую помещена левая нижняя вершина прямоугольника;
- за ними следуют целочисленные координаты точки, в которую помещена правая верхняя вершина прямоугольника;
- затем следует цвет прямоугольника, заданный целым числом от 1 до 64; белый цвет представлен числом 1.

Порядок строк соответствует порядку, в котором прямоугольники размещались на листе от первого до последнего. Наборы данных для различных тестов разделены пустой строкой (см. пример).

Напишите программу, которая:

1. Читает очередной набор данных из входного файла с именем RECTANG.DAT.
2. Вычисляет площадь каждой из одноцветных фигур (см. пример).
3. Записывает в выходной ASCII-файл с именем RECTANG.SOL цвет и площадь каждой одноцветной фигуры, как показано ниже в примере. Эти результаты должны записываться в порядке возрастания номера цвета. Результаты для различных тестов разделяются пустой строкой.

Пример.

RECTANG.DAT:

```
20 12 5
-7 -5 -3 -1 4
-3 -3 5 3 2
-4 -2 -2 2 4
 2 -2 3 -1 12
 3 1 7 5 1

30 30 2
 0 0 5 14 2
10 -7 0 13 15
```

RECTANG.SOL:

```
1 177
2 39
4 23
12 1

1 630
2 70
15 200
```

Идея решения

Задача решается, если будет найдено решение более частной задачи вычисления площади одноцветной фигуры, составленной из прямоугольников, имеющих хотя бы одну общую точку (пример одноцветной фигуры приведен на рис. 5.3).

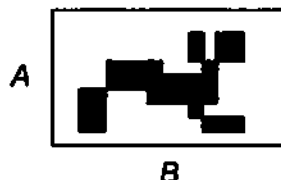


Рис. 5.3. Пример одноцветной фигуры, площадь которой равна 34 единицам

Воспользуемся классической идеей решения задачи о прохождении лабиринта. Тогда вычисление площади фигуры основывается на обходе в глубину:

- 1) начнем обход с произвольной клетки нашей фигуры, а в счетчик площади занесем 0;
- 2) попав в какую-либо клетку, увеличим счетчик площади на 1, пометим эту клетку и проанализируем по очереди все соседние клетки, которые еще не помечены и также принадлежат фигуре.

Пусть в массиве *bar* (*bar* : Array [-15..15, -15..15] Of Integer) хранятся цвета клеток. Тогда площадь фигуры (*s*), содержащей клетку (*x*, *y*), вычисляется как *s* := *Bar*[*x*, *y*]; *s* := 0; *Rec*(*x*, *y*), где процедура *Rec* выглядит следующим образом:

Procedure *Rec*(*i*, *j* : Integer); {*s*, *s* – глобальные переменные}

Begin

If (*i* >= -*A*) **And** (*i* < *A*) **And** (*j* >= -*B*) **And** (*j* < *B*) **And** (*Bar*[*i*, *j*] = *c*) **Then Begin** {Если клетка принадлежит фигуре того же цвета и не помечена}

s := *s* + 1;

Bar[*i*, *j*] := 0;

Rec(*i* - 1, *j* - 1);

Rec(*i* - 1, *j*);

Rec(*i* - 1, *j* + 1);

Rec(*i*, *j* - 1);

Rec(*i*, *j* + 1);

Rec(*i* + 1, *j* - 1);

Rec(*i* + 1, *j*);

Rec(*i* + 1, *j* + 1);

End;

End;

При решении задачи в целом следует обратить внимание на то, что первоначально все клетки поля помечаются первым цветом. И только затем на это поле «накладываются» прямоугольники других цветов, в частности, и первого цвета. Это следует из примеров, приведенных в формулировке задачи. Кроме того, перед выводом результата найденные фигуры следует отсортировать в порядке возрастания их цветов.

Методические указания

Данная задача может быть использована как простое дополнительное задание при изучении темы «Перебор с возвратом», в частности, задач на поиск выхода из лабиринта.

Написание нерекурсивного варианта реализации процедуры Res является хорошим упражнением.

Самостоятельные исследования

Вопрос о том, при каких размерах поля время работы программы будет равно, например, 1 с, потребует написания дополнительных программ и экспериментирования.

ЗАДАЧА ВТОРОГО ТУРА: «КАНАДСКИЕ АВИАЛИНИИ»

Условие задачи

Вы победили в соревновании, организованном канадскими авиакомпаниями. Приз — бесплатное путешествие по Канаде. Путешествие начинается с самого западного города, в который летают самолеты, и проходит с запада на восток, пока не достигнет самого восточного города, в который летают самолеты. Затем путешествие продолжается обратно с востока на запад, пока не достигнет начального города. Ни один из городов нельзя посещать более одного раза, за исключением начального города, который надо посетить ровно дважды (в начале и в конце путешествия). Вам также нельзя пользоваться авиакомпаниями других компаний или другими способами передвижения. Задача состоит в следующем: дан список городов и список прямых рейсов между парами городов; найти маршрут, включающий максимальное количество городов и удовлетворяющий выше-названным условиям.

Несколько наборов входных данных помещены в ASCII-файл C:\IOI\ITIN.DAT. Каждый набор содержит:

- в первой строке — количество городов n , в которые летают самолеты, и количество прямых рейсов v . n — положительное целое число, меньшее или равное 100. v — положительное целое число;

- в каждой из следующих l строк — название города, в который летают самолеты. Названия упорядочены с запада на восток, то есть i -й по порядку город находится восточнее j -го тогда и только тогда, когда $i > j$ (не существует городов на одном меридиане). Название каждого города — строка, состоящая из не более чем 15 цифр и/или латинских букв, например: AGR34 или BELA;
- в каждой из следующих u строк — названия двух городов из списка городов, разделенные пробелом. Если пара ГОРОД1 ГОРОД2 содержится в строке, это означает, что есть прямой рейс из ГОРОД1 в ГОРОД2, а также прямой рейс из ГОРОД2 в ГОРОД1.

Различные наборы входных данных разделены пустой строкой. После последнего набора данных пустой строки не будет.

Пример файла входных данных C:\IOI\ITIN.DAT:

```

8 9
Vancouver
Yellowknife
Edmonton
Calgary
Winnipeg
Toronto
Montreal
Halifax
Vancouver Edmonton
Vancouver Calgary
Calgary Winnipeg
Winnipeg Toronto
Toronto Halifax
Montreal Halifax
Edmonton Montreal
Edmonton Yellowknife
Edmonton Calgary
```

```

5 5
C1
C2
C3
C4
C5
C5 C4
C2 C3
C3 C1
C4 C1
C5 C2
```

Входные данные корректны, и их проверка не требуется. Решения, полученные для каждого набора входных данных, должны быть последовательно записаны в выходной ASCII-файл C:\IOI\ITIN.SOL следующим образом:

- в первой строке — общее количество городов из входного файла;
- в следующей строке — число m , равное количеству различных городов, посещаемых на маршруте;
- в следующих $m+1$ строках — названия городов по одному в строке в порядке их посещения.

Обратите внимание, что исходный город должен быть также выведен последним. Если для набора входных данных не существует решения, то для этого набора в выходном файле C:\IOI\ITIN.SOL должны быть только две записи: общее количество городов и сообщение «NO SOLUTION» (нет решения). Решения для разных наборов входных данных должны разделяться в выходном файле пустой строкой.

Образец выходного файла C:\IOI\ITIN.SOL для приведенного выше примера:

```
8
7
Vancouver
Edmonton
Montreal
Halifax
Toronto
Winnipeg
Calgary
Vancouver

5
NO SOLUTION
```

Поместите вашу программу-решение в ASCII-файл с именем C:\IOI\DDD.xxx. Расширение .xxx равно .BAS для QBasic, .LCN для LOGO, .C для C, .PAS для PASCAL.

Обсуждение задачи

При решении этой задачи нам будет удобнее считать, что города пронумерованы в обратном порядке — с востока на запад. Пусть $c_1, \dots, c_n$ — заданные города, а в массиве *name* (name: Array[1..n] Of String[15]) хранятся их названия.

Рассмотрим сначала более простую задачу: необходимо попасть из города c_1 в город c_n , двигаясь только на запад и посетив при этом максимально возможное число городов.

Эта задача может быть решена методом динамического программирования: последовательно для $i = 1, 2, \dots, n$ отыскиваются числа $d[i]$ — количество городов в наилучшем маршруте из c_1 в c_i или значение $-\infty$, если нет ни одного такого маршрута. При вычислении $d[i]$ ($i > 1$), как обычно в этом методе, используется факт, что числа $d[1], \dots, d[i-1]$ уже известны.

Пусть q_i — множество номеров городов, расположенных восточнее города c_i и имеющих с ним прямое воздушное сообщение. Очевидно, что в любом маршруте, связывающем c_1 и c_i , номер предпоследнего города обязан принадлежать множеству q_i . Тогда выполняются следующие равенства (максимум из нуля чисел полагаем равным $-\infty$):

$$\begin{cases} d[1] = 1, \\ d[i] = \max_{j \in q_i} (d[j] + 1) \quad (i > 1). \end{cases}$$

Количество городов в требуемом маршруте — это значение $d[n]$. Но как найти сам маршрут? Пусть $how[i]$ ($i > 1$, $d[i] \neq -\infty$) — номер предпоследнего города в наилучшем маршруте до города c_i (это то самое j , на котором достигается максимум во втором равенстве). Элементы этого массива формируются в логике нахождения наилучшего маршрута (роль значения $-\infty$ будет играть ноль):

```
d[1]:=1;
For i:=2 To n Do Begin
  d[i]:=0;
  For j∈qi Do
    If (d[j]<>0) And (d[j]+1>d[i]) Then Begin
      d[i]:=d[j]+1;
      how[i]:=j;
    End;
End;
If d[n]=0 Then WriteLn('NO SOLUTION')
Else Begin
  WriteLn(d[n]);
  Way(n);
End;
```

Тогда последовательность городов маршрута выводится так:

```
Procedure Way(i:Integer); {Выводит маршрут от ci
до c1}
Begin
```

```

If i<>1 Then Way(how[i]);
WriteLn (Name[i]);
End;

```

Вернемся, однако, к первоначальной постановке задачи. Под словами «путь из города c_i в город c_j » мы теперь будем понимать маршрут, состоящий из двух частей: от c_i до c_1 (на восток) и от c_1 до c_j (на запад), любые два города в котором (кроме, быть может, начального и конечного городов) различны. По условию задачи необходимо найти наилучший (по количеству городов) путь из c_n в c_n .

Эта задача решается по аналогии с упрощенной, рассмотренной нами ранее. Обозначим через $d[i, j]$ количество различных городов в наилучшем пути из c_i в c_j . Тогда верны следующие соотношения:

$$\begin{cases} d[1,1] = 1, \\ d[j,i] = d[i,j], \\ d[i,j] = \begin{cases} \max_{k \in q_i} (d[k,j]) + 1 & (i > j), \\ k \neq j, \text{ если } j \neq 1 \end{cases} \\ d[i,i] = \max(d[k,i]) \quad (i > 1). \end{cases}$$

Первые два равенства очевидны. Проверим третье равенство. Примем, что $i > j$ и возьмем наилучший путь из c_i в c_j . Предположим, что вторым городом в этом пути является город c_k , $k \in q_i$ (причем $k \neq j$, если $j \neq 1$). Тогда, убрав город c_i из имеющегося пути, мы получим наилучший путь из c_k в c_j , содержащий $d[k, j]$ городов. И наоборот, взяв любой путь из c_k ($k \in q_i$, $k \neq j$, если $j \neq 1$) в c_j и добавив к нему в качестве первого город c_i , мы получаем путь из c_i в c_j . Именно на этом этапе рассуждений для обеспечения неповторяемости городов в пути и использовано условие $i > j$. Из этих двух утверждений следует третье равенство.

Четвертое соотношение обосновывается аналогичным образом.

Для вывода городов, принадлежащих наилучшему пути, требуется дополнительная структура данных — массив *how*. Элемент *how*[i, j] при $i \geq j$ обозначает номер города, следующего за c_i в наилучшем пути из c_i в c_j , — это то самое k , на котором достигается максимум в третьем (при $i > j$) или четвертом (при $i = j$) равенстве. При $i < j$ значение *how*[i, j] положим равным *how*[j, i], а элемент *how*[1, 1] не определен.

Методическое отступление. Приведа один из возможных способов программной реализации логики решения задачи, можно предложить учащимся выполнить ее «ручную» трассировку для конкретного примера.

```

Procedure Solve;
  Var i,j,k:Integer;
  Begin
    <Элементам массива d присвоить значение 0>;
    {Массивы d, h имеют следующее описание: d, how :
    Array [1..n,1..n] Of Integer}
    d[1,1]:=1;
    For i:=2 To n Do
      For j:=1 To i Do
        If (i<>j) Or (i=n) Then
          For k:=1 To i-1 Do
            If ((k<>j) Or (j=1)) And (d[k,j]<>0) And
              (d[k,j]+1>d[i,j]) Then Begin
              d[i,j]:=d[k,j]+1; d[j,i]:=d[i,j];
              how[i,j]:=k; how[j,i]:=k;
            End;
          If d[n,n]=0 Then WriteLn('NO SOLUTION')
          Else Begin
            WriteLn(d[n,n]-Ord(n>1)); {Для правильной
            обработки случая n=1}
            Way(n,n);
            If n=1 Then WriteLn(name[1]);
          End;
        End;
  End;

```

Процедура же вывода наилучшего пути из города c_i в город c_j (Way) традиционна:

```

Procedure Way(i,j : Integer);
Begin
  If (i=1) And (j=1) Then WriteLn(name[1])
  Else If i>=j Then Begin
    WriteLn(name[i]);
    Way(how[i,j],j);
  End
  Else Begin
    Way(i,how[i,j]);
    WriteLn(name[j]);
  End;
End;

```

Методические рекомендации

Данную задачу в настоящее время (но не в 1993 году!) можно считать традиционной по теме «Динамическое программирование». Целесообразность ее рассмотрения следует из того, что различные вариации этой задачи неоднократно использовались в олимпиадной проблематике.

Самостоятельные исследования

Задача решается с помощью обычного перебора вариантов*. Экспериментальное сравнение времени того и другого решения для различных значений n требует определенных усилий от школьника. Заметим, что полный балл за задачу получили именно участники олимпиады, решившие ее методом динамического программирования.

* Ожолов С. М., Пестов А. А. 100 задач по информатике. Киров: Изд-во ВГПУ, 2000. С. 98–102.

ЗАДАЧИ ПЕРВОГО ТУРА

«Треугольник»

Условие задачи

На рис. 6.1 изображен треугольник из чисел. Напишите программу, которая вычисляет наибольшую сумму чисел, расположенных на пути, начинающемся в верхней точке треугольника и заканчивающемся на основании треугольника.

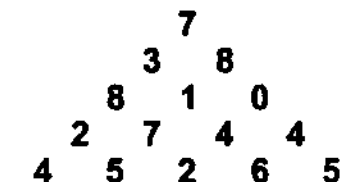


Рис. 6.1. Треугольник из чисел

Каждый шаг на пути может осуществляться вниз по диагонали влево или вниз по диагонали вправо.

Число строк в треугольнике больше 1 и меньше или равно 100.

Треугольник составлен из целых чисел от 0 до 99.

Входные данные. Первым числом во входном файле с именем INPUT.TXT является количество строк в треугольнике. Пример файла исходных данных для приведенного на рис. 6.1 треугольника представлен ниже:

```
5
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

Выходные данные. В выходной файл с именем OUTPUT.TXT записывается только наибольшая сумма в виде целого числа. Ниже приведен файл OUTPUT.TXT для описанных выше исходных данных:

30

Идея решения

Это типичная задача на метод динамического программирования. Пусть $A[i, j]$ обозначает j -е число в i -й строке треугольника, а $S[i, j]$ — наибольшую сумму чисел на пути от вершины треугольника до этого числа. Тогда очевидны соотношения:

$$\begin{cases} S[1,1] = A[1,1], \\ S[i,j] = \max(S[i-1,j], S[i-1,j-1] + A[i,j]) \quad (1 \leq j \leq i \leq n, i > 1). \end{cases}$$

Примем, что числа $S[i, 0]$ и $S[i, i+1]$ ($1 \leq i \leq n$) равны нулю. Ответ задачи — это максимальное число в последней строке треугольника (матрицы S): $\max_{1 \leq j \leq n} S[n, j]$.

Для приведенного в условии задачи примера числа $S[i, j]$ будут следующими:

```

7
10 15
18 16 15
20 25 20 19
24 30 27 26 24

```

Ответ задачи: число 30.

Методические указания

Эта задача даже в 1994 г. носила скорее «утешительный» характер, а в настоящее время может быть использована как простое упражнение при изучении темы «Динамическое программирование».

Самостоятельные исследования

1. В ресторане собираются n посетителей*. Посетитель с номером i подходит во время t_i и имеет p_i рублей. Входная дверь ресторана имеет k состояний открытости. Состояние открытости двери может изменяться на одну единицу за одну единицу времени, т. е. она или открывается на единицу, или остается в том же состоянии. В начальный момент времени дверь закрыта (состояние 0). Посетитель с номером i входит в ресторан только в том случае, если дверь открыта специально для него, т. е. когда состояние открытости двери совпадает с его степенью полноты s_i . Если в момент прихода посетителя состояние двери не совпадает с его степенью полноты, то посетитель уходит и больше не возвращается. Ресторан работает в течение времени t .

Цель: правильно открывая дверь, добиться того, чтобы за время работы ресторана в нем собрались посетители с максимальной суммой денег (предполагается, что они останутся в ресторане).

*

Эту задачу один из авторов в свое время предложил как шутку (по формулировке) для одной из санкт-петербургских олимпиад. Слово «посетитель» там было заменено на слово «гангстер», и задача под этим названием сейчас «гуляет» в разных источниках.

Входные данные. В первой строке находятся значения n , k и t , разделенные пробелами ($1 \leq n \leq 100$, $1 \leq k \leq 100$, $0 \leq t \leq 30000$). Во второй строке находятся времена прихода посетителей $t_1, t_2, \dots, t_n$, разделенные пробелами ($0 \leq t_i \leq t$, для всех $i = 1, 2, \dots, n$). В третьей строке находятся значения $p_1, p_2, \dots, p_n$, разделенные пробелами ($0 \leq p_i \leq 300$, для всех $i = 1, 2, \dots, n$). В четвертой строке находятся значения степени полноты посетителей $s_1, s_2, \dots, s_n$, разделенные пробелами ($1 \leq s_i \leq k$, для всех $i = 1, 2, \dots, n$). Все исходные данные — целые числа.

Выходные данные. Выводится одно число, являющееся максимальным значением суммы денег посетителей. В случае, если нельзя добиться прихода в ресторан ни одного посетителя, выходной файл должен содержать нуль*.

Пример:

```
4 10 20
10 16 8 16
10 11 15 1
10 7 1 8
```

Результат — число 26.

2. Количество треугольников**.

Рассмотрим фигуру, аналогичную показанной на рис. 6.2 (большой равносторонний треугольник, составленный из маленьких равносторонних треугольников). На рис. 6.2 приведена фигура, состоящая из 4 уровней треугольников.

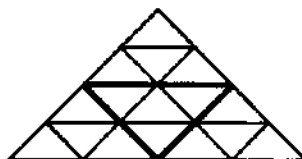


Рис. 6.2. Пример фигуры

Напишите программу, которая будет определять, сколько всего треугольников в заданной фигуре (необходимо учитывать не только «маленькие» треугольники, а вообще все треугольники — в частности треугольник, выделенный жирным, а также вся фигура в целом являются интересующими нас треугольниками).

Входные данные. Одно число n — количество уровней в фигуре ($1 \leq n \leq 100000$).

* Окулов С. М., Пестов А. А. 100 задач по информатике. Киров: Изд-во ВГПУ, 2000. С. 102–104.

** Андреева Е. В., Гурвич В. М., Матюхин В. А. и др. Московские олимпиады по программированию. М.: МЦНМО, 2006. С. 116–117.

Выходные данные. Одно число — количество треугольников в фигуре.

Пример: $n = 4$; ответ — число 27.

«Замок»

Условие задачи

На рис. 6.3 изображен план замка. Написать программу, которая определяет:

- количество комнат в замке;
- площадь наибольшей комнаты.

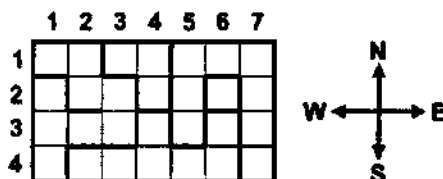


Рис. 6.3. План замка

Какую стену в замке следует удалить, чтобы получить комнату наибольшей площади?

Замок условно разделен на $m \times t$ клеток ($m \leq 50$, $t \leq 50$). Каждая такая клетка может иметь от 0 до 4 стен.

Входные данные. План замка содержится во входном файле с именем INPUT.TXT в виде последовательности чисел, по одному числу, характеризующему каждую клетку. В начале файла расположено число клеток в направлении с севера на юг и число клеток в направлении с запада на восток. В последующих строках каждая клетка описывается числом p ($0 \leq p \leq 15$). Это число является суммой следующих чисел: 1 (если клетка имеет западную стену), 2 (северную), 4 (восточную), 8 (южную). Внутренняя стена считается принадлежащей обеим клеткам. Например, южная стена в клетке (1, 1) также является северной стеной в клетке (2, 1). Замок содержит по крайней мере две комнаты.

Ниже приведен пример файла исходных данных INPUT.TXT для примера, изображенного на рис. 6.3.

```

4
7
11 6 11 6 3 10 6
7 9 6 13 5 15 5
1 10 12 7 13 7 5
13 11 10 8 10 12 13

```

Выходные данные. В выходном файле с именем OUTPUT.TXT должно быть 3 строки. В первой строке содержится число комнат, во второй — площадь наибольшей комнаты (измеряется количеством клеток). Третья строка содержит три числа, определяющих удаляемую стену: номер строки и номер столбца

клетки, содержащей удаляемую стену и положение этой стены в клетке (N — север, W — запад, S — юг, E — восток). Ниже приведен выходной файл OUTPUT.TXT для нашего примера:

5

9

4 1 E (*4 1 E* — один из возможных способов описания удаляемой стены).

Идея решения

Это классическая задача по теме «лабиринт», особенностью которой является своеобразное описание исходных данных, поэтому обсуждение задачи можно начать с небольшого упражнения.

Методическое отступление. Введем следующее описание данных:

```
Type Doors = (W,N,E,S);
      Can = Set Of Doors;
Var A : Array [1..50,1..50] Of Can;
```

По данным из файла INPUT.TXT требуется написать наиболее компактный программный код формирования матрицы A . Для примера из формулировки задачи она имеет вид:

$$A = \begin{bmatrix} [W, N, S] & [N, E] & [W, N, S] & [N, E] & [W, N] & [N, S] & [N, E] \\ [W, \dots, E] & [W, S] & [N, E] & [W, E, S] & [W, E] & [W, \dots, S] & [W, E] \\ [W] & [N, S] & [E, S] & [W, \dots, E] & [W, E, S] & [W, \dots, E] & [W, E] \\ [W, E, S] & [W, N, S] & [N, S] & [S] & [N, S] & [E, S] & [W, E, S] \end{bmatrix}$$

Элемент матрицы $A[i, j]$ при этом имеет тип «множество» и характеризует стены конкретной клетки.

Клетки, принадлежащие одной комнате замка, помечаются «меткой» (матрица num) — целым положительным числом, которое является индексом к элементам массива squ для фиксации площади комнат. При введенных структурах данных:

```
Var num : Array [1..50,1..50] Of Integer;
      squ : Array [1..2500] Of Integer;
      cnt;
```

и принятом описании замка с помощью матрицы A рекурсивный обход очередной найденной комнаты реализуется компактно и наглядно:

```

Procedure Wgn(i,j : Integer);
Begin
  num[i,j]:=cnt;
  squ[cnt]):=squ[cnt]+1;
  If (i>1) And (Not (N In A[i,j])) And (num[i-1,j]=0)
    Then Wgn(i-1,j); {Идем на север}
  If (i<m) And (Not (S In A[i,j])) And (num[i+1,j]=0)
    Then Wgn(i+1,j); {Идем на юг}
  If (j>1) And (Not (W In A[i,j])) And (num[i,j-1]=0)
    Then Wgn(i,j-1); {Идем на запад}
  If (j<t) And (Not (E In A[i,j])) And (num[i,j+1]=0)
    Then Wgn(i,j+1); {Идем на восток}
End;

```

Вызов процедуры Wgn традиционный — мы просматриваем *num* и при нахождении нулевого элемента (новая комната) — num[*i,j*]=0 — увеличиваем на единицу значение *cnt*, а затем начинаем рекурсивный обход.

Поиск комнаты с наибольшей площадью сводится к нахождению максимального элемента в массиве *squ*.

Для решения последнего пункта задачи требуется удалять по очереди все стены (разделяя две клетки) и проверять:

```

Procedure Vr(n1,n2,i,j:Integer; Door : Doors);
{n1 и n2 — номера комнат; i,j — координаты клетки
комнаты с номером n1; Door — направление}
Begin
  If (n1=n2) And (squ[n1]+squ[n2]>smax) Then Begin
    smax:=squ[n1]+squ[n2]; {Нашли лучшее решение,
запоминаем клетку и направление}
    li:=i; lj:=j; LDoor:=Door;
  End;
End;

```

Заметим, что следствием соответствия структур данных решаемой задаче является не менее изящный, чем процедура Wgn, фрагмент логики удаления стены:

```

smax:=-1;
For i:=1 To m Do
  For j:=1 To t Do Begin
    If (i>1) And (N In A[i,j])
      Then Vr(num[i,j],num[i-1,j],i,j,N);
    If (i<m) And (S In A[i,j])
      Then Vr(num[i,j],num[i+1,j],i,j,S);
  End;

```

```

If (j>1) And (W In A[i,j])
Then Vr(num[i,j],num[i,j-1],i,j,W);
If (j<t) And (E In A[i,j])
Then Vr(num[i,j],num[i,j+1],i,j,E);
End;

```

Методические указания

Задача традиционна для темы «Поиск с возвратом». Ее методическая ценность заключается в возможности в очередной раз убедиться в верности утверждения Н. Вирта о том, что программа — это синтез в единое целое управления и структур данных, основанный на соответствии структур данных решаемой задачи.

Самостоятельные исследования

Требуется удалить минимальное количество стен так, чтобы из любой комнаты можно было попасть в любую другую комнату.

«Матрица простых чисел»

Условие задачи

На рис. 6.4 изображена числовая матрица. Каждая строка, каждый столбец и обе диагонали матрицы рассматриваются как 5 цифр, составляющих простое число. Строки читаются слева направо. Столбцы читаются сверху вниз. Обе диагонали читаются слева направо.

Напишите программу, которая на основе исходных данных, расположенных во входном файле с именем INPUT.TXT, находит описанные выше матрицы.

1	1	3	5	1
3	3	2	0	3
3	0	3	2	3
1	4	0	3	3
3	3	3	1	1

Рис. 6.4. Пример матрицы

- Простые числа должны иметь одинаковую сумму цифр (например, 11).
- Цифра в левом верхнем углу матрицы задается заранее.
- Матрица может содержать одинаковые простые числа.
- В случае нескольких возможных вариантов решения выдать все решения.
- Простое число не может начинаться с нуля, например 00003 не является простым пятизначным числом.

Входные данные. Программа читает данные из входного файла INPUT.TXT, в котором расположена сумма цифр в простых числах и заданная цифра в левом верхнем углу матрицы. Файл состоит из двух строк. Для заданных при тестировании файлов всегда существует хотя бы одно решение.

Пример файла исходных данных INPUT.TXT:

```
11
1
```

Выходные данные. Для каждого найденного варианта решения записать в выходной файл с именем OUTPUT.TXT пять строк, каждая из которых содержит пятизначное простое число. Ниже приведен файл OUTPUT.TXT для описанных выше входных данных (пустую строку, разделяющую варианты решения, помещать в выходной файл не обязательно):

```
1 1 3 5 1
1 4 0 3 3
3 0 3 2 3
5 3 2 0 1
1 3 3 1 3
```

```
1 1 3 5 1
3 3 2 0 3
3 0 3 2 3
1 4 0 3 3
3 3 3 1 1
```

```
1 3 3 1 3
1 3 0 4 3
3 2 3 0 3
5 0 2 3 1
1 3 3 3 1
```

Обсуждение задачи

Это задача на умение правильно организовать перебор. Первоначально находятся все пятизначные простые числа с заданной суммой цифр s (назовем такие числа «правильными»). Результаты поиска заносятся в массив `Var r : Array [1..9, 0..9, 0..9, 0..9] Of Boolean`; где элемент $r[a, b, c, d]$, равный `True`, означает, что существует правильное число, первые четыре цифры которого суть a, b, c и d (последняя цифра этого числа находится как $s - a - b - c - d$).

Назовем строки, столбцы и диагонали матрицы «линиями». Всего в ней существует 12 линий. Заполнять матрицу цифрами можно слева направо и сверху вниз, проверяя получающиеся на каждой линии числа на «правильность». Однако этот алгоритм работает медленно. Поэтому следует как можно раньше отсекал «ветви» перебора, не приводящие к решению, для

чего необходимо уменьшить количество ветвлений на верхних уровнях дерева вариантов.

Процесс нахождения матрицы в обсуждаемом решении состоит из 12 шагов, на каждом из которых заполняются оставшиеся пустые клетки на одной из линий. Перебираются все различные варианты заполнения, приводящие к «правильному» числу («правильность» числа проверяется с использованием массива r). Заметим, что цифры в последней строке и в последнем столбце должны быть нечетными, а цифры в первой строке и в первом столбце не могут быть нулями — это позволяет существенно сократить число анализируемых вариантов.

В табл. 6.1 для каждой клетки указан номер шага, на котором в нее записывается цифра, а в табл. 6.2 для каждого шага указана линия, которая заполняется в соответствующий момент. Левая верхняя клетка матрицы помечена нулем, так как цифра в ней известна заранее.

Таблица 6.1

0	1	1	1	1
9	3	9	4	2
8	5	3	7	2
11	4	10	3	2
4	5	6	6	2

Таблица 6.2

№	Линия	№	Линия
1	Строка 1	7	Столбец 4
2	Столбец 5	8	Строка 3
3	Диагональ 1	9	Строка 2
4	Диагональ 2	10	Столбец 3
5	Столбец 2	11	Строка 4
6	Строка 5	12	Столбец 1

На шагах 7, 8, 10 и 11 вычисляется оставшаяся цифра (исходя из суммы s и уже известных цифр) и проверяется «правильность» образовавшегося числа. Заполнять на шаге 12, вообще говоря, уже нечего — требуется лишь проверка числа в первом столбце на его «правильность». Если это так, то получено очередное решение задачи, и необходимо вывести его в выходной файл.

Методические указания

Это задача по традиционной для олимпиад теме «Перебор с возвратом». Ее методическая ценность заключается в развитии навыков отсечения ветвей перебора на ранней стадии.

Самостоятельные исследования

На международной олимпиаде на каждый тест отводилось по 90 секунд. Однако программа, составленная по обсуждаемому алгоритму, на самом сложном тесте работает на компьютере

типа 486DX2 лишь около 0,6 секунды. Набор использовавшихся на олимпиаде тестов приведен в табл. 6.3.

Таблица 6.3

№	Исходные данные	Количество решений
1	11 5	1
2	37 9	3
3	35 5	1
4	13 4	1

Требуется реализовать алгоритм заполнения матрицы по цифрам слева направо и сверху вниз и провести сравнительный анализ времени работы программ на приведенных тестах.

ЗАДАЧИ ВТОРОГО ТУРА

«Часы»

Условие задачи

В матрице размером 3*3 расположены 9 циферблатов с заданным положением стрелок (рис. 6.5). Требуется установить на всех циферблатах время 12 часов.

Возможны 9 различных способов изменения стрелок на циферблатах. Каждый такой способ задается набором циферблатов, стрелки которых поворачиваются на 90 градусов по часовой стрелке. На рис. 6.6 для каждого из способов соответствующие циферблаты выделены серым цветом. Каждый способ определяется номером — числом от 1 до 9 (рис. 6.6).

Напишите программу, которая определяет кратчайшую последовательность шагов, переводящую стрелки всех циферблатов из начальной позиции в позицию 12 часов.

Входные данные. Перевод стрелок из начального состояния в конечное производится последовательностью шагов. За один шаг можно осуществить перевод стрелок на циферблатах одним из указанных способов.

Исходные данные расположены во входном файле с именем INPUT.TXT, в котором задано начальное расположение стрелок на циферблатах. Положе-



Рис. 6.5. Пример положения стрелок на 9 циферблатах

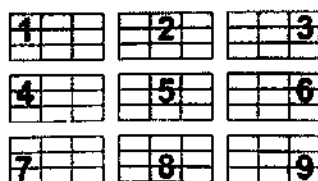


Рис. 6.6. Способы перевода стрелок

ние стрелки на циферблате задается числом от 0 до 3: 0 — 12 часов, 1 — 3 часа, 2 — 6 часов, 3 — 9 часов. Ниже приведен файл INPUT.TXT для примера, изображенного на рис. 6.5:

```
3 3 0
2 2 2
2 1 2
```

Выходные данные. Поместить в выходной файл с именем OUTPUT.TXT искомую последовательность шагов (номеров способов). В случае нескольких решений вывести только один из возможных вариантов ответа. Файл OUTPUT.TXT для нашего примера имеет вид:

```
5 8 4 9
```

Данная последовательность шагов наглядно представлена на рис. 6.7.

Обсуждение задачи

Пронумеруем циферблаты в матрице слева направо и сверху вниз, начиная с 1. Пусть число a_i означает, сколько раз требуется повернуть стрелку i -го циферблата на 90 градусов, чтобы на нем установилось время 12 часов ($0 \leq a_i \leq 3$).

Легко заметить, что порядок шагов в решении не играет никакой роли. Следовательно, решение полностью описывается набором чисел $(s_1, s_2, \dots, s_9)$, где s_i равно количеству применений способа i ($0 \leq s_i \leq 3$). Для нахождения этого набора требуется решить по модулю 4 следующую систему линейных уравнений (см. рис. 6.6):

$$\begin{cases} x_1 + x_2 + x_4 = a_1, \\ x_1 + x_2 + x_3 + x_5 = a_2, \\ x_2 + x_3 + x_6 = a_3, \\ x_1 + x_4 + x_5 + x_7 = a_4, \\ x_1 + x_3 + x_5 + x_7 + x_9 = a_5, \\ x_3 + x_5 + x_6 + x_9 = a_6, \\ x_4 + x_7 + x_8 = a_7, \\ x_5 + x_7 + x_8 + x_9 = a_8, \\ x_6 + x_8 + x_9 = a_9. \end{cases}$$

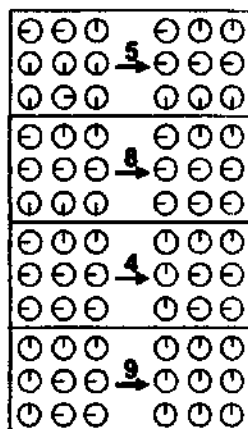


Рис. 6.7.
Последовательность шагов по преобразованию циферблатов

Рассмотрим эту систему уравнений. Для любых чисел a_i она имеет единственное решение, так как определитель матрицы коэффициентов равен 5, т. е. отличен от нуля. Умножим правые части этих уравнений на 5, и пусть $(x_1, x_2, \dots, x_9)$ — решение модифицированной таким образом системы. Это решение — целочисленное (что следует, например, из формул Крамера), причем, положив s_i для всех $i = 1, \dots, 9$ равным $x_i \pmod{4}$, мы получим, очевидно, решение исходной системы по модулю 4.

Будет ли это решение единственным, удовлетворяющим неравенствам $0 \leq s_i \leq 3$ ($1 \leq i \leq 9$)? Мы показали, что для каждого входного набора $(a_1, \dots, a_9)$ существует хотя бы одно такое решение системы $(s_1, \dots, s_9)$, причем различным наборам соответствуют, очевидно, различные решения. Количество различных входных наборов равно 4^9 . Но количество различных решений, удовлетворяющих этим неравенствам, также равно 4^9 . Поэтому для любого набора входных данных существует единственное (с точностью до порядка шагов) оптимальное решение, которое можно найти с помощью описанной процедуры.

Проще, однако, перебрать все возможные значения для s_1, s_2 и s_3 , выражая через них числа $s_4, s_5, \dots, s_9$ с помощью некоторых из уравнений системы (идея В. Баргачева). Если полученный набор чисел $(s_1, \dots, s_9)$ удовлетворяет и оставшимся уравнениям, — значит, найдено решение задачи.

Итак, процедура поиска решения:

```

For s1:=0 To 3 Do
  For s2:=0 To 3 Do
    For s3:=0 To 3 Do Begin
      s4:= (a1-s1-s2) Mod 4;
      s5:= (a2-s1-s2-s3) Mod 4;
      s6:= (a3-s2-s3) Mod 4;
      s7:= (a4-s1-s4-s5) Mod 4;
      s8:= (a7-s4-s7) Mod 4;
      s9:= (a6-s3-s5-s6) Mod 4;
      If ((s1+s3+s5+s7+s9) Mod 4 = a5) And ((s5+s7+s8+s9)
Mod 4 = a8) And ((s6+s8+s9) Mod 4 = a9) Then Exit;
    End;
  
```

Примечание. Оператор Mod языка *Turbo Pascal* при отрицательном первом аргументе работает не совсем корректно: например, $-1 \text{ Mod } 4 = -1$. Необходимо устранить этот недостаток из «наброска» алгоритма.

Методические указания

Данная задача, с одной стороны, требует знания элементов алгебры — умения решать системы уравнений. С другой сторо-

редном шаге окажется, что все отсчеты задействованы. Далее необходимо сравнить его с наилучшим из ранее найденных разбиений.

Однако этот простой алгоритм перебора «не укладывается» во временные ограничения. Необходим поиск эвристик, сокращающих перебор.

Первая эвристика. Часть маршрутов имеет длину 2, т. е. содержит два отсчета. Назовем такие маршруты «большими парами». Оказывается, что для разбиения какого-либо множества отсчетов на «большие пары» можно применять «жадный» алгоритм: взять первый свободный отсчет и подобрать к нему первый подходящий для образования «большой пары»; затем опять взять первый свободный отсчет, проделать с ним то же самое и т. д. Если же на каком-либо шаге не удастся найти подходящий свободный отсчет, то разбиение невозможно.

Тем самым алгоритм простого перебора можно модифицировать. Взяв первый свободный отсчет m ($m \leq 29$), следует отнести его к одному из маршрутов (m, d) длины не менее 3 ($m + 2*d \leq 59$), либо «оставить на потом» (для «больших пар»), увеличивая счетчик «больших пар» $pcnt$ на единицу. Если на каком-то шаге количество оставшихся свободных отсчетов rem равно $2*pcnt$, то разбиваем их (если это возможно) на «большие пары» с помощью «жадного» алгоритма.

Вторая эвристика. Пусть к очередному шагу нами уже построено cnt маршрутов, а первый свободный отсчет — m ($m \leq 29$). Все маршруты, которые выделяются в дальнейшем, имеют длину не более $l = \lceil (60-m)/(m+1) \rceil$ (где $\lceil x \rceil$ означает наименьшее целое число, большее или равное x). Следовательно, если число $cnt + pcnt + \lceil (rem - 2*pcnt)/l \rceil$ больше или равно количеству маршрутов min в наилучшем из найденных разбиений, то эту «ветку» перебора можно не рассматривать. В начале работы программы значение min равно 18, так как известно, что количество маршрутов в наилучшем разбиении не превосходит 17.

С учетом вышесказанного, логика процедуры поиска разбиения может выглядеть следующим образом:

Procedure Разбиение;

Var m, d : Integer;

Begin

If $rem < 2*pcnt$ **Then** **Exit**;

If $rem = 2*pcnt$ **Then** **Begin** Большие_Пары; **Exit** **End**;

$m :=$ <Первый свободный отсчет>;

If $m > 29$ **Then** **Exit**;

If $cnt + pcnt + \lceil (rem - 2*pcnt) / \lceil (60-m)/(m+1) \rceil \rceil \geq min$

Then **Exit**; {Отсечение}

```

cnt:=cnt+1; {Ищем новый маршрут}
For <всех маршрутов (m,d) длины  $\geq 3$ > Do
  If <есть свободные отсчеты m, m+d, ...> Then Begin
    <Добавляем маршрут (m,d) к частичному разбиению>;
    <Помечаем отсчеты как "занятые">;
    Разбиение;
    <Помечаем отсчеты как "свободные">;
  End;
cnt:=cnt-1;
If <существует хотя бы одна большая пара (m,d)>
Then Begin
  pcnt:=pcnt+1; {Оставляем для больших пар}
  Разбиение;
  pcnt:=pcnt-1;
End;
End;

```

Вспомогательную процедуру Большие_Пары при этом можно записать так:

```

Procedure Большие_Пары;
Var t,m,d : Integer;
Begin {В этой точке cnt+pcnt<min}
  For t:=1 To pcnt Do Begin {Выделяем очередную
    большую пару}
    m:=<Первый свободный отсчет>;
    If m>29 Then Exit;
    For <всех маршрутов (m,d) длины 2> Do
      If <есть свободный отсчет m+d> Then Break;
      If <нашли подходящий отсчет m+d>
        Then <Добавляем маршрут (m,d) к частичному
        разбиению>
        Else Exit;
    End;
  min:=cnt+pcnt;
  <Запоминаем построенное разбиение как наилучшее>;
End;

```

Вызов же этих процедур должен осуществляться так:

```

rem:=n; {Осталось отсчетов}
min:=18; {Число маршрутов в наилучшем разбиении}
cnt:=0; {Текущее число маршрутов}
pcnt:=0; {Текущее число больших пар}
Разбиение;

```

Так как несколько отсчетов могут соответствовать одной и той же минуте, необходимо ввести массив, элементы которого бы означали, сколько осталось свободных отсчетов для каждой из минут. Наконец, можно сократить перебор, заметив, что порядок отнесения отсчетов, соответствующих одной минуте, к маршрутам не имеет значения. Так, если один из этих отсчетов был отнесен к маршруту (m, d_1) , то для следующего отсчета мы будем перебирать маршруты (m, d_2) с $d_2 \geq d_1$.

В ряде случаев (на определенных входных данных) приведенные эвристики сокращают время получения решения.

Методические указания

Это задача на традиционный перебор с возвратом. Рассмотренный перечень эвристик не окончателен, его можно расширить. В целом поиск и экспериментальную проверку таких эвристик следует считать достойным проявлением профессионализма в информатике. Материал данной задачи предоставляет широкое поле для этой деятельности.

Самостоятельные исследования

В табл. 6.5 приведены тесты, использовавшиеся при тестировании программ на международной олимпиаде, и ответы к ним.

Таблица 6.5

№	Тест	Количество маршрутов
1	12 2 3 13 19 19 24 35 35 40 46 51 57	3
2	44 0 1 3 5 5 6 7 9 12 13 13 13 15 18 19 21 21 21 24 25 27 29 29 30 31 33 36 37 37 37 39 42 43 45 45 45 48 49 51 53 53 54 55 57	4
3	43 0 1 2 3 5 6 9 10 12 12 13 14 15 17 18 21 22 24 25 26 27 28 29 30 33 34 36 37 38 39 41 42 44 45 46 48 49 50 51 53 54 57 58	5
4	31 0 2 3 5 6 6 8 10 14 19 20 22 24 26 26 29 32 32 38 38 39 42 44 45 50 52 56 56 57 58 58	7
5	40 0 1 2 5 6 9 11 11 15 15 16 17 18 19 21 21 26 26 28 31 33 33 35 36 36 36 36 39 41 43 46 47 51 53 53 54 55 56 57 57	11
6	70 0 0 1 1 2 2 2 3 3 5 6 6 7 10 11 12 13 13 13 14 17 17 19 20 20 21 22 22 23 25 25 26 26 27 27 27 29 30 32 33 34 37 37 38 39 41 41 41 42 43 44 46 46 46 47 47 48 48 49 50 52 52 52 55 55 57 57 59 59	17

Поиск входных данных, для которых рассмотренные эвристики не дают желаемого результата (сокращения времени перебора), является предметом исследования. Эта работа потребует разработки генераторов тестов и продуманного экспериментирования.

«Сектора»

Условие задачи

Задан круг, разделенный на сектора. Даны три числа: k ($0 \leq k \leq 20$), n ($n \leq 6$) и m ($m \leq 20$), где n — количество секторов. В каждый из секторов помещается одно число, не меньшее k . Когда сектора заполнены числами, вы можете получать из них новые числа по следующим правилам:

- взять число из одного сектора;
- взять число, равное сумме двух или более чисел в смежных секторах.

Из этих чисел составляется наибольшая последовательность подряд идущих новых чисел, начинающаяся с числа m : $(m, m+1, m+2, \dots, i)$.

Напишите программу, которая определяет способ расстановки чисел в секторах, максимизирующий длину последовательности.

Пример на рис. 6.8 показывает, как получить все новые числа от 2 до 21 для приведенных чисел в секторах. Серым цветом выделены суммируемые числа.

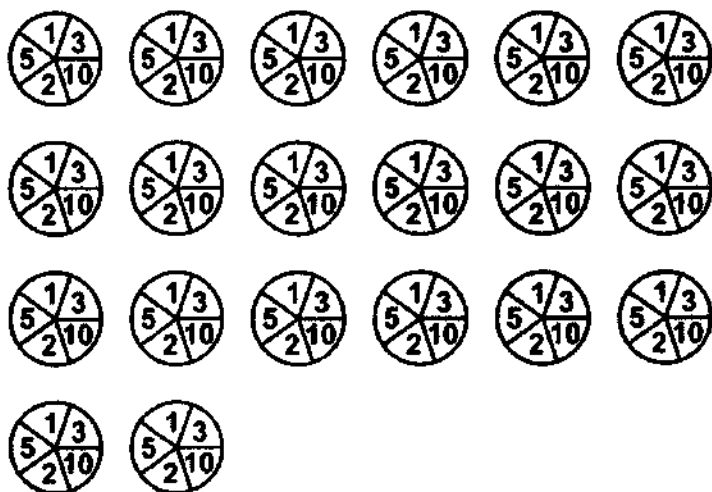


Рис. 6.8. Пример получения последовательности из чисел в секторах

Входные и выходные данные. Исходные данные — числа n , m и k — расположены во входном файле с именем INPUT.TXT. Ниже приведен пример файла исходных данных INPUT.TXT:

```
5
2
1
```

Выходной файл с именем OUTPUT.TXT должен содержать:

- наибольшее число i в неразрывной последовательности, которое может быть получено из чисел в секторах;
- все наборы чисел в секторах, из которых можно получить неразрывную последовательность от m до i . В каждой строке выводится один набор чисел в секторах. Каждый такой набор — список чисел, начинающихся с наименьшего (которое может быть не единственным).

Например, (2 10 3 1 5) не является решением, так как начинается не с наименьшего числа. Обратите внимание, что (1 3 10 2 5) и (1 5 2 10 3) считаются различными решениями и должны быть оба выведены. Ниже приведен файл OUTPUT.TXT для нашего примера:

```
21
1 3 10 2 5
1 5 2 10 3
2 4 9 3 5
2 5 3 9 4
```

Если наименьшее число встретится несколько раз, вывести все возможные комбинации, например: (1 1 2 3), (1 2 3 1), (1 3 2 1) и (1 1 3 2).

Обсуждение задачи

Пусть для определенности минимальное из чисел находится в первом секторе. Очевидно, что это число должно принадлежать интервалу от k до m (если $k > m$, то задача решений не имеет). Введем массив `Var a : Array [1..n] Of Integer;` для хранения текущего состояния круга ($a[i]$ равно числу в i -м секторе).

Задачу можно решать с помощью рекурсивного алгоритма расстановки чисел в секторах: на каждом шаге заполняется очередной сектор, корректируется множество новых чисел, которые можно получить, и осуществляется переход к следующему сектору.

Какие числа допустимо ставить в сектора? В условии задачи сказано, что они обязаны быть не меньше k , но ничего не говорится о верхней границе!

Пусть $n-1$ секторов круга заполнено числами, а какой-либо один из них пуст. Легко заметить, что при этом можно получить не более чем $n(n-1)/2$ различных новых чисел. Обозначим через x первое число из последовательности $m, m+1, \dots$, которое получить нельзя. Тогда в оставшийся пустым сектор не имеет смысла ставить число, большее x . Поскольку x не превосходит $n(n-1)/2+m$, то последнее выражение является верхней границей для чисел в секторах. Если $a[1] < m$, то эта граница уменьшается еще на единицу. В секторы с номерами 2, 3, ..., n помещаются числа, не меньшие $a[1]$, — это будет нижней границей.

Для последнего сектора рассматриваемый диапазон чисел можно еще уменьшить. В качестве нижней границы (если $n > 2$) допустимо взять число $a[2]$, чтобы исключить симметричные варианты, а в качестве верхней — то самое число x , о котором говорилось выше. Если при этом сумма чисел, стоящих в секторах 1, ..., $n-1$, и числа x меньше наилучшего из найденных значений i , то при любом варианте заполнения последнего сектора новое решение не получается.

Методическое отступление. Итак диапазон значений чисел в секторах определен. Возможен вариант «лобового» решения и оценки времени его работы.

Первая эвристика. Пусть $m < 2^*k$. Тогда числа $m, m+1, \dots, 2^*k-1$ не могут быть получены как сумма чисел из нескольких (более одного) секторов. Следовательно (если $2^*k-m \leq n$), все они должны находиться в круге либо (если $2^*k-m > n$) в круге должны находиться числа $m, m+1, \dots, m+n-1$, и только они.

Методическое отступление. Проверка данного факта путем простого подсчета является необходимым условием дальнейшей работы с задачей.

Вторая эвристика. Введем массив `Var sum : Array [1..n, 1..n] Of Byte;`, элемент `sum[i, j]` которого будет равняться сумме чисел в j последовательных секторах, начиная с сектора i (если, конечно, все они уже заполнены). К моменту заполнения очередного сектора p должно быть сформировано текущее множество s новых чисел, которые получаются из чисел в секторах от 1 до $p-1$, и числа `sum[i, j]` ($1 \leq i < p, i+j \leq p$). После заполнения сектора p (при переходе к следующему сектору) необходимо вычислить значения `sum[i, p-i+1]` ($1 \leq i \leq p$) и включить их в множество s . Если этот сектор — последний ($p = n$), то

вычисляются и оставшиеся элементы матрицы zim , при этом они также заносятся в множество z .

После заполнения секторов отыскивается наибольшее число i в неразрывной последовательности, получаемой из чисел в секторах, и сравнивается с максимальным из ранее найденных ($maxi$). Если $i = maxi$, то к списку решений добавляется новое решение. При $i > maxi$ создается новый список решений из найденного способа заполнения секторов и обновляется значение $maxi$.

При выводе результата для каждого из найденных решений с $s[2] < s[n]$ требуется выдать также и симметричное ему решение с $s[2] > s[n]$, полученное путем зеркального отражения относительно первого сектора.

Методические указания

Это задача по теме «Перебор с возвратом», и, как и предыдущая, позволяет не только совершенствовать навыки реализации традиционной схемы, но и «оттачивать» искусство поиска эвристик (которое действительно является искусством).

Самостоятельные исследования

1. Разработайте программу и оцените время ее работы на тестах, приведенных в табл. 6.6.

Таблица 6.6

№	N	m	k	i	Количество решений
1	3	1	1	7	2
2	4	2	2	13	2
3	5	3	1	22	2
4	4	16	1	23	6
5	5	16	12	20	24
6	6	1	1	31	10
7	6	2	2	29	2
8	6	4	4	31	6

2. Увеличьте диапазон значений входных данных. Оцените время работы программы (оно не должно превышать 1 с). Поиск дополнительных эвристик, сокращающих перебор, является предметом исследования.

ЗАДАЧИ ПЕРВОГО ТУРА

«Упаковка прямоугольников»

Условие задачи

Даны четыре прямоугольника. Найдите наименьший покрывающий их (новый) прямоугольник, в который могут поместиться без перекрытий заданные четыре прямоугольника. Под наименьшим мы понимаем прямоугольник наименьшей площади.

Стороны всех четырех прямоугольников должны быть параллельны соответствующим сторонам покрывающего прямоугольника. Рис. 7.1 показывает шесть способов совместного размещения четырех прямоугольников. Эти шесть примеров исчерпывают все возможные структуры, так как любая другая может быть получена из некоторой приведенной поворотами и/или отражениями.

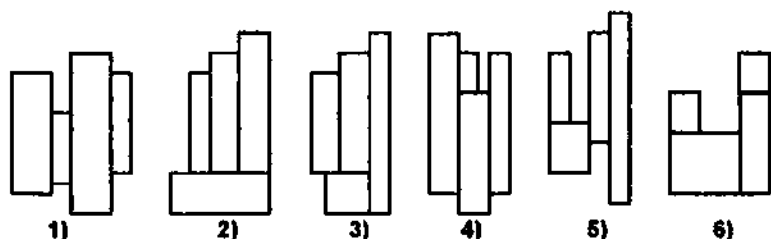


Рис. 7.1. Шесть основных конфигураций для четырех прямоугольников

Могут существовать покрывающие прямоугольники с различными длинами сторон, удовлетворяющие условиям и имеющие минимальную площадь. Вы должны получить все такие покрывающие прямоугольники.

Входные данные. Входной файл INPUT.TXT состоит из четырех строк. Каждая строка описывает один исходный прямоугольник двумя положительными целыми числами — длинами сторон прямоугольника. Каждая сторона прямоугольника имеет длину не менее 1 и не более 50.

Выходные данные. Выходной файл OUTPUT.TXT должен содержать на одну строку больше, чем существует решений. Первая строка должна содержать одно целое число: минимальную площадь покрывающего прямоугольника (подзадача A). Каждая из следующих строк должна содержать одно решение, опи-

санное числами p и q — длинами сторон покрывающего прямоугольника; при этом должно выполняться условие $p \leq q$ (подзадача В). Эти строки должны быть упорядочены по возрастанию p ; все они должны быть различными.

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
1 2	40
2 3	4 10
3 4	5 8
4 5	

Обсуждение задачи

Поясним на рис. 7.2 шесть способов совместного расположения четырех прямоугольников, ранее показанные на рис. 7.1. При этом для большей понятности объяснений дополним изображения прямоугольников на рис. 7.2 их буквенными обозначениями.

Назовем два прямоугольника A и B независимыми по оси Ox (Oy), если их проекции на эту ось не имеют общих внутренних точек. Возможны 16 вариантов ориентации 4 прямоугольников, так как каждый из прямоугольников (если это не квадрат) можно ориентировать двумя способами.

Вне зависимости от ориентации, существуют следующие три основных варианта расположения прямоугольников.

- 1) все 4 прямоугольника попарно независимы по одной из осей (можно считать, что по оси Ox , — см. фигуру 1 на рис. 7.2);
- 2) существуют 3 прямоугольника, попарно независимых по одной из осей (можно считать, что по оси Ox). Обозначим эти три прямоугольника буквами A , B , C , а оставшийся — буквой D . Тогда возможны следующие три случая:
 - а) D не зависит по оси Oy от каждого из других прямоугольников A , B и C (как фигура 2 на рис. 7.2),
 - б) D зависит по оси Oy от одного из других прямоугольников A , B и C (как фигура 3 на рис. 7.2),
 - в) D зависит по оси Oy от двух из прямоугольников A , B , C . В этом случае D располагается над (или под) оставшимся из первых трех прямоугольников. При этом возможны два варианта: оставшийся прямоугольник может быть крайним среди A , B , C (фигура 5 на рис. 7.2) или средним (фигура 4 на рис. 7.2). Однако необходимости различать эти два варианта нет;
- 3) по каждой из осей максимальное количество попарно независимых прямоугольников равно двум (фигура 6 на рис. 7.2).

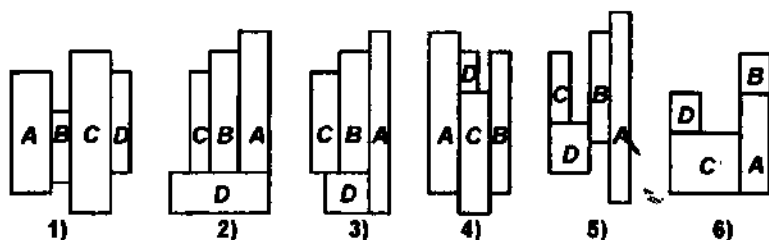


Рис. 7.2. Шесть основных конфигураций для четырех прямоугольников (иллюстрация объяснений)

Методическое отступление. Желательно прорисовать 16 вариантов ориентации 4 прямоугольников и убедиться в полноте предложенной классификации их взаимного расположения.

Разберем для каждого из указанных выше случаев алгоритм поиска ширины и высоты минимального покрывающего прямоугольника P .

В первом случае предположим, что прямоугольники выровнены по нижнему краю. Тогда ширина P должна быть равна сумме ширин всех четырех прямоугольников, а высота P — высоте самого высокого из них.

Во втором случае считаем, что прямоугольники A , B и C расположены справа налево в порядке убывания высоты и выровнены по нижнему краю. При этом:

1) если прямоугольник D расположен снизу от прямоугольников A , B , C , то:

- $\text{ширина}(P) = \text{Max}(\text{ширина}(A) + \text{ширина}(B) + \text{ширина}(C), \text{ширина}(D))$,
- $\text{высота}(P) = \text{высота}(A) + \text{высота}(D)$;

2) если прямоугольник D расположен под B и C , то:

- $\text{ширина}(P) = \text{ширина}(A) + \text{Max}(\text{ширина}(B) + \text{ширина}(C), \text{ширина}(D))$,
- $\text{высота}(P) = \text{Max}(\text{высота}(A), \text{высота}(B) + \text{высота}(D))$;

3) если прямоугольник D расположен под C , то:

- $\text{ширина}(P) = \text{ширина}(A) + \text{ширина}(B) + \text{Max}(\text{ширина}(C), \text{ширина}(D))$,
- $\text{высота}(P) = \text{Max}(\text{высота}(A), \text{высота}(C) + \text{высота}(D))$.

В третьем случае, если четыре заданных прямоугольника расположены без перекрытия внутри большего, то допустимо считать, что они находятся в его углах (либо их можно передвинуть в его углы). Будем считать, что в правом нижнем углу находится самый высокий из прямоугольников — A . Пусть прямоугольник B расположен в правом верхнем, C — в левом

нижнем и D — в левом верхнем углах. Тогда выполняются следующие соотношения:

- $\text{ширина}(P) \geq RW = \text{Max}(\text{ширина}(A) + \text{ширина}(C), \text{ширина}(B) + \text{ширина}(D))$,
- $\text{высота}(P) \geq RH = \text{Max}(\text{высота}(A) + \text{высота}(B), \text{высота}(C) + \text{высота}(D))$.

Если прямоугольники A и D независимы по оси Ox , то

- $\text{ширина}(P) = \text{Max}(RW, \text{ширина}(A) + \text{ширина}(C))$;
- $\text{высота}(P) = RH$,

иначе:

- $\text{ширина}(P) = RW$;
- $\text{высота}(P) = \text{Max}(RH, \text{высота}(A) + \text{высота}(D))$.

Общая схема алгоритма при этом имеет вид:

For <каждого из 16 вариантов ориентации
прямоугольников> **Do Begin**

<Анализ первой схемы расположения прямоугольников>;

<Анализ второй схемы расположения прямоугольников>;

<Анализ третьей схемы расположения прямоугольников>;

End;

Методические указания

В 1960–70-е годы профессиональную пригодность специалиста по информатике оценивали с помощью задач такого типа, где требовалось проанализировать все случаи взаимного расположения двух треугольников (или четырехугольников) на плоскости. Думается, что современный школьник легко прошел бы это тестирование.

Тщательный логический анализ условий задачи — залог ее успешного решения. Знания специальных алгоритмов и методов при этом не требуется, но качество программного кода напрямую зависит от выбранных структур данных.

Самостоятельные исследования

1. На плоскости отмечено $n = 3k$ точек. Будем рассматривать такие варианты построения k невырожденных треугольников с вершинами в этих точках, при которых каждая из заданных точек является вершиной какого-либо треугольника. Точки расположены так, что хотя бы одно построение с указанным свойством существует. Требуется определить вариант, при котором суммарная площадь полученных k треугольников минимальна.

Входные данные. В файле содержатся (в указанном порядке) целое число n ($1 \leq n \leq 30$) и n пар вещественных чисел, задающих координаты точек. Числа разделяются пробелами и/или символами перевода строки.

Выходные данные. Первая строка файла должна содержать минимально возможное значение суммарной площади. В каждой из следующих k строк записывается тройка номеров вершин, образующих очередной треугольник. Номера вершин разделяются пробелами*.

2. Два многоугольника на плоскости заданы координатами своих вершин. Требуется вычислить площадь пересечения этих многоугольников, т. е. сумму площадей тех «кусков», которые образуются при пересечении и принадлежат каждому из них. При этом возможен один из двух случаев:

- многоугольники выпуклые, а координаты их вершин даны в произвольном порядке;
- хотя бы один из многоугольников — невыпуклый, но известно, что у каждого многоугольника имеется не более одного угла, большего 180° , а координаты вершин даны в порядке обхода по часовой стрелке.

Программа по входным данным должна сама определить, какой из этих двух случаев имеет место.

Входные данные. Первая строка входного файла содержит целое число n — количество вершин в первом многоугольнике ($3 \leq n \leq 50$). Во второй строке записаны координаты этих вершин. Третья и четвертая строки таким же образом задают второй многоугольник. Координаты всех вершин — целые числа из диапазона $[-32768, 32767]$.

Выходные данные. Файл содержит искомую площадь не менее чем с 6 верными цифрами после запятой**.

«Торговые скидки»

Условие задачи

В магазине каждый товар имеет цену. Например, цена одного цветка равна 2 ICU (ICU — Informatics Currency Unit), а цена одной вазы равна 5 ICU. Чтобы привлечь больше покупателей, магазин ввел скидки.

Скидка заключается в том, чтобы продавать набор одинаковых или разных товаров по пониженной цене. Примеры: три цветка за 5 ICU вместо 6, или две вазы вместе с одним цветком за 10 ICU вместо 12.

* Беров В. И., Лалунов А. В., Матюхин В. А., Пономарев А. Е. Особенности национальных задач по информатике. Киров: ООО Триана-С, 2000. С. 31–32.

** Там же. С. 94–96.

Напишите программу, вычисляющую наименьшую цену, которую покупатель должен заплатить за заданные покупки. Оптимальное решение должно быть получено посредством скидок. Набор товаров, который требуется купить, нельзя дополнять ничем, даже если бы это снизило общую стоимость набора. Для описанных выше цен и скидок наименьшая цена за три цветка и две вазы равна 14 ICU: две вазы и один цветок продаются по сниженной цене за 10 ICU, и два цветка — по обычной цене за 4 ICU.

Входные данные. Входные данные содержатся в двух файлах: INPUT.TXT и OFFER.TXT. Первый файл описывает покупки («корзину с покупками»). Второй файл описывает скидки. В обоих файлах содержатся только целые числа.

Первая строка файла INPUT.TXT содержит количество b различных видов товара в корзине ($0 \leq b \leq 5$). Каждая из следующих b строк содержит значения s , k и p . Значение s — уникальный код товара ($1 \leq s \leq 999$). Значение k задает, сколько единиц товара находится в корзине ($1 \leq k \leq 5$). Значение p задает обычную (без скидок) цену единицы товара ($1 \leq p \leq 999$). Обратите внимание, что общее количество товаров в корзине может быть не более $5 \cdot 5 = 25$ единиц.

Первая строка файла OFFER.TXT содержит количество s возможных скидок ($0 \leq s \leq 99$). Каждая из следующих s строк описывает одну скидку, определяя набор товаров и общую стоимость набора. Первое число n в такой строке определяет количество различных видов товара в наборе ($1 \leq n \leq 5$). Следующие n пар чисел (s, k) указывают, что k единиц товара с кодом s включены в набор для скидки ($1 \leq k \leq 5, 1 \leq s \leq 999$). Последнее число в строке p определяет уцененную стоимость набора ($1 \leq p \leq 9999$). Стоимость набора меньше суммарной стоимости отдельных единиц товаров в наборе.

Выходные данные. Запишите в выходной файл OUTPUT.TXT одну строку с наименьшей возможной суммарной стоимостью покупок, заданных во входном файле.

Пример ввода и вывода. Ниже по тексту приведены входные и выходной файлы для приведенного выше примера. Код продукта-цветка равен 7, код вазы равен 8.

INPUT.TXT
2
7 3 2
8 2 5

OFFER.TXT
2
1 7 3 5
2 7 1 8 2 10

OUTPUT.TXT
14

Идея решения

Пусть требуется купить k_1 товаров с кодом c_1 , k_2 товаров с кодом c_2 , ..., k_b товаров с кодом c_b ($c_1, \dots, c_b$ — фиксированные числа и $b \leq 5$). Обозначим через $A[k_1, \dots, k_b]$ наименьшую возможную стоимость такого набора. Значение $A[k_1, \dots, k_b]$ можно вычислять динамически: вначале положим $A[0, \dots, 0]$ равным нулю, затем последовательно будем вычислять $A[k_1, \dots, k_b]$ с $k_1 + \dots + k_b = 1, 2, 3$ и т. д. При этом для каждого из анализируемых наборов мы будем пытаться применить все возможные скидки, пользуясь тем, что для наборов с меньшим количеством товаров наименьшая стоимость уже вычислена.

Алгоритм решения задачи:

Begin

<Удалить скидки, в которые входят не нужные нам товары>;

<Установить стоимость пустого набора товаров равной нулю>;

For cnt:=1 **To** <Количество товаров в "корзине"> **Do**

For <Всех наборов с числом товаров cnt> **Do Begin**

 min:=<Стоимость товаров из набора, купленных без скидок>;

For <всех скидок> **Do**

If <в наборе есть все товары из скидки>

Then Begin

 new:=<Стоимость товаров из скидки> +
 <Стоимость оставшихся товаров (уже подсчитана)>;

If new<min **Then** min:=new;

End;

 <Запомнить минимальную стоимость анализируемого набора (min)>;

End;

 <Вывести минимальную стоимость набора>;

End.

Методические указания

Эта задача по теме «Динамическое программирование» имеет скорее сложную, запутанную формулировку, чем сложное решение. «Сжатость» ее обсуждения вызвана тем, что с методической точки зрения данная задача представляет интерес как задание для самостоятельной работы, ибо для аудиторной работы со школьниками она не очень удобна. Лаконичный и изящный программный код здесь создать достаточно трудно, он непосредственно зависит от структур данных, выбранных для

представления информации по задаче, а последняя — избыточна. С другим вариантом разбора задачи можно ознакомиться в книге М. С. Долинского*.

Самостоятельные исследования

Смысл параметра «код товара» понятен — он связывает корзину товаров с системой скидок и позволяет придумать «замысловатые» варианты исходных данных. Исключите из формулировки задачи (сохранив общий ее смысл) параметр «код товара» (999 возможных кодов для пяти видов товаров). Снимите ограничение «не более 5» как на количество различных видов товаров в корзине, так и на значение единиц товара (ведь использовать динамическую схему решения при общем количестве товаров в корзине не более 25 — это все равно, что «стрелять из пушки по воробьям»). В решении исходной задачи используется пятимерный массив, но в данном случае это сделать будет нельзя. Решите новый вариант задачи, разработайте систему тестов для ее проверки и тестирующую программу.

«Печать»

Условие задачи

Клиент и сервер

Имеются два пользователя, у каждого из которых есть компьютер. Компьютеры идентифицируются именами: CLIENT(1) и CLIENT(2). Два компьютера подключены к одному или более принтерам с именами SERVER(1), SERVER(2) и т. д. Печать с обоих компьютеров может производиться только последовательно. Чтобы скоординировать взаимодействие компьютеров с принтером, используется объект SEMAPHORE (семафор).

Объект SEMAPHORE

Каждый принтер имеет относящийся к нему семафор. Семафор находится в одном из двух состояний: S1 или S2. Когда принтер свободен для печати, семафор находится в состоянии S1. Когда принтер печатает, семафор находится в состоянии S2.

Семафор может осуществлять два типа переходов состояний: «S1 $\Rightarrow$ S2» и «S2 $\Rightarrow$ S1». Когда пользователь посылает компьютеру задание для печати, компьютер посылает семафору сообщение «Are_you_open?». Если семафор находится в состоянии S1, то его состояние меняется на S2 и семафор посылает сооб-

* Долинский М. С. Алгоритмизация и программирование на TURBO PASCAL: от простых до олимпиадных задач / М. С. Долинский. СПб.: Питер, 2005. С. 148–153.

щение «Open» тому компьютеру, который прислал сообщение «Are_you_open?». Если семафор находится в состоянии S2, то он возвращает сообщение «Closed». После завершения печати принтер посылает сообщение «Ready» своему семафору. После получения сообщения «Ready» семафор меняет свое состояние на S1.

Тип объекта SEMAPHORE

В документе 1 вы найдете спецификацию типа объекта SEMAPHORE. Спецификация содержит возможные идентификаторы и состояния объекта типа SEMAPHORE, «Список приоритетов», «Диаграмму соединений», «Диаграмму перехода состояний» и «Процедуры приема» для сообщений «Ready» и «Are_you_open?». В «Процедурах приема» описано, как семафор откликается на сообщения.

«Список приоритетов» необходим, поскольку сообщения, получаемые семафором одновременно, могут быть обработаны только последовательно. «Список приоритетов» в документе 1 указывает, например, что каждое сообщение SERVER'a имеет более высокий приоритет, чем сообщение CLIENT'a, а сообщение SERVER(2)'а имеет более высокий приоритет, чем сообщение SERVER(3)'а.

Документ 1. Тип объекта: SEMAPHORE

Возможные идентификаторы: {SEMAPHORE(1), SEMAPHORE(2), ...}

Состояния: {S1, S2}; начальное состояние - S1.

Список приоритетов: SERVER(1), SERVER(2), ..., CLIENT(1), CLIENT(2), ...

Диаграмма соединений



Процедуры приема сообщений

```

procedure Are_you_open?(Client, Semaphore)
begin
    if      State = S1
    then    State ← S2
            Send('Open(Semaphore, Client)')
    else
    if      State = S2
    then    Send('Closed(Semaphore, Client)')
end

procedure Ready(Server, Semaphore)
begin
    State ← S1
end

```

Тип объекта CLIENT

В документе 2 вы найдете спецификацию типа объекта CLIENT. Объект типа CLIENT находится в одном из трех состояний: SA, SB или SC. Клиент находится в состоянии SA, если этот клиент не обращался к серверу и серверы не печатают задания для этого клиента. Клиент находится в состоянии SB, если он хочет получить доступ к серверу, но не может (клиент может получить доступ к серверу только посредством семафора). Клиент находится в состоянии SC, если сервер выполняет задание на печать для данного клиента.

Объект CLIENT может изменять состояние тремя способами: «SA ⇒ SB», «SB ⇒ SC», «SC ⇒ SA». Когда объект CLIENT находится в состоянии SB, он может получить сообщение «Closed» от семафора. После получения этого сообщения клиент ждет в течение периода «Waiting_Period», прежде чем снова послать семафору сообщение «Are_you_open?». Когда семафор посылает клиенту сообщение «Open», этот клиент изменяет свое состояние на SC и одновременно посылает задание на печать серверу с помощью сообщения «S_Job». После завершения работы сервер одновременно посылает два сообщения: сообщение «Ready» своему семафору и сообщение «C_Ready» объекту CLIENT. Затем объект CLIENT меняет свое состояние с SC на SA. Принтеры можно считать идеальными, то есть они завершают каждое задание на печать. Например, никогда не кончится бумага.

Документ 2. Тип объекта: CLIENT

Возможные идентификаторы:

{CLIENT(1), CLIENT(2), CLIENT(3), ...}

Состояния: {SA, SB, SC}; начальное состояние - SA.

Список приоритетов:

CLIENT, SERVER(1), SERVER(2), ..., SEMAPHORE(1),

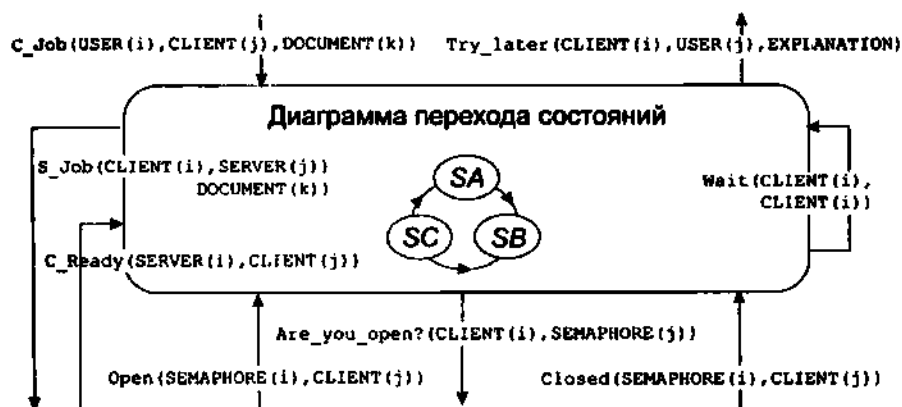
SEMAPHORE(2), ..., USER(1), USER(2), ...

Countdown: {t | t ∈ N}; начальное значение - 0.

Waiting_Period: {t | t ∈ N и t > 0}

Semaphore_Index: {i | i = 1, 2, ..., Number_of_Servers}

Number_of_Servers: {i | i ∈ N и i > 0}

Диаграмма соединений**Процедуры приема сообщений**

```

procedure C_Job(User, Client, Document)
begin
  if State = SA
  then State ← SB
    Send("Are_you_open?(Client, SEMAPHORE(Semaphore_Index))")
  else
    if State = SB
    then Send("Try_later(Client, User, Client_is_busy)")
    else
      if State = SC
      then Send("Try_later(Client, User, All_Servers_are_busy)")
End

```

```

procedure Open(Semaphore, Client)
begin
    if State = SB
    then State  $\leftarrow$  SC
        Send("S_Job(Client, Server, Document)")
End

procedure Closed(Semaphore, Client)
begin
    Countdown  $\leftarrow$  Waiting_Period
    Send("Wait(Client, Client)")
End

procedure Wait(Client, Client)
begin
    Countdown  $\leftarrow$  Countdown - 1
    if Countdown > 0
    then Send("Wait(Client, Client)")
    else Send("Are_you_open?(Client, SEMAPHORE(Semaphore_Index))")
End

procedure C_Ready(Server, Client)
begin
    State  $\leftarrow$  SA
End

```

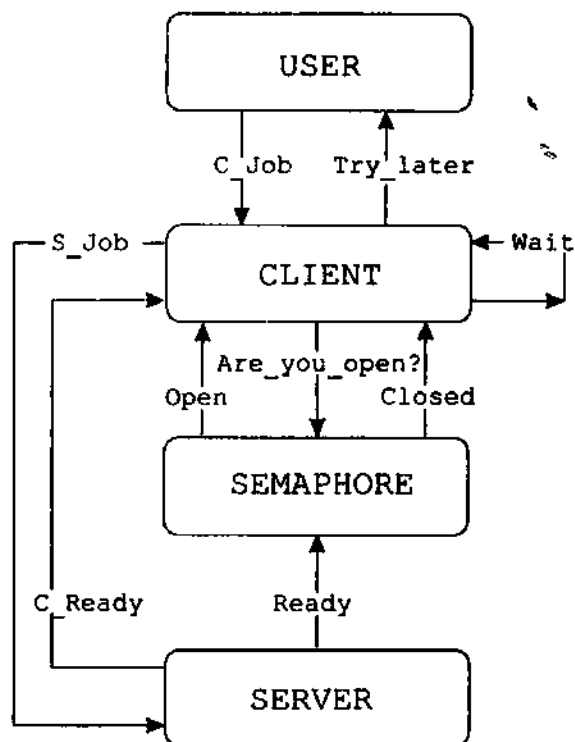
Соединения

В документе 3 вы найдете в «Диаграмме структуры соединений» все типы сообщений, которые могут пересылаться между различными типами объектов. В «Списке сообщений» вы найдете спецификацию каждого типа сообщений. Каждое сообщение имеет идентификатор, имя отправителя, имя получателя и, в некоторых случаях, какое-то содержание.

Когда отправитель в момент времени t посылает сообщение с идентификатором A , то получатель в момент времени $t + 1$ будет обрабатывать сообщение путем выполнения «Процедуры приема» с именем A .

Если несколько отправителей в момент времени t посылают сообщения одному и тому же получателю, то получатель обрабатывает все эти сообщения в момент времени $t + 1$ в том же порядке, в котором отправители расположены в «Списке приоритетов» данного получателя.

Документ 3. Диаграмма структуры соединений



Список сообщений

Идентификатор	Отправитель	Получатель	Содержание
Are_you_open?	CLIENT (i)	SEMAPHORE (j)	—
C_Job	USER (i)	CLIENT (j)	DOCUMENT (k)
C_Ready	SERVER (i)	CLIENT (j)	—
Closed	SEMAPHORE (i)	CLIENT (j)	—
Open	SEMAPHORE (i)	CLIENT (j)	—
Ready	SERVER (i)	SEMAPHORE (j)	—
S_Job	CLIENT (i)	SERVER (j)	DOCUMENT (k)
Try_later	CLIENT (i)	USER (j)	EXPLANATION
Wait	CLIENT (i)	CLIENT (i)	—

Подзадача А

В начальный момент времени (0) локальная сеть определяется следующими объектами:

- объект: CLIENT (1), *Client.State* = SA, *Waiting_Period* = 2, *Number_of_Servers* = 1;

- объект: CLIENT(2), *Client.State* = SA, *Waiting_Period* = 1, *Number_of_Servers* = 1;
- объект: SERVER(1)
- объект: SEMAPHORE(1), *Semaphore.State* = S1.

В этой сети были посланы, помимо других, следующие сообщения:

- в момент времени 1: CLIENT(1) послал сообщение с идентификатором «Are_you_open?»;
- в момент времени 2: CLIENT(2) послал сообщение с идентификатором «Are_you_open?»;
- в момент времени 4: SERVER(1) послал сообщение с идентификатором «Ready»;
- в момент времени 5: CLIENT(1) послал сообщение с идентификатором «Are_you_open?».

В документе 4 для объектов SEMAPHORE(1), CLIENT(1) и CLIENT(2) в виде таблицы до момента времени 6 отображено, какие сообщения объекты посылали, какие сообщения объекты получали, а также в каких состояниях объекты находились, или в какие состояния они переходили.

Вопрос A1

Что случится, если в дополнение к вышесказанному в момент времени 4 объект CLIENT(1) получит сообщение «C_Job»?

Сформулируйте свой ответ (ответ составляйте согласно документу 4).

Вопрос A2

Что случится, если в дополнение к вышесказанному в момент времени 4 объект CLIENT(2) получит сообщение «C_Job» вместо объекта CLIENT(1)?

Сформулируйте свой ответ (ответ составляйте согласно документу 4).

Вопрос A3

Заполните в документе 4 таблицу до момента времени 13 включительно, если известно, что:

- в момент времени 8: SERVER(1) послал сообщение с идентификатором «Ready»;
- в момент времени 10: CLIENT(1) получил сообщение с идентификатором «C_Job»;
- в момент времени 12: SERVER(1) послал сообщение с идентификатором «Ready».

Документ 4

SEMAPHORE (1)

	Полученные сообщения	Сост./переход	Посланные сообщения
<i>time</i>			
0		S1	
1		S1	
2	Are_you_open?	S1 $\Rightarrow$ S2	Open
3	Are_you_open?	S2	Closed
4		S2	
5	Ready	S2 $\Rightarrow$ S1	
6	Are_you_open? Are_you_open?	S1 $\Rightarrow$ S2 S2	Open Closed

CLIENT (1)

	Полученные сообщения	Сост./переход	Count-down	Посланные сообщения
<i>time</i>				
0		SA	0	
1	C_Job	SA $\Rightarrow$ SB	0	Are_you_open?
2		SB	0	
3	Open	SB $\Rightarrow$ SC	0	S_Job
4		SC	0	
5	C_Ready C_Job	SC $\Rightarrow$ SA SA $\Rightarrow$ SB	0 0	Are_you_open?
6				

CLIENT (2)

	Полученные сообщения	Сост./переход	Count-down	Посланные сообщения
<i>time</i>				
0		SA	0	
1		SA	0	
2	C_Job	SA $\Rightarrow$ SB	0	Are_you_open?
3		SB	0	
4	Closed	SB	0 $\Rightarrow$ 1	Wait
5	Wait	SB	1 $\Rightarrow$ 0	Are_you_open?
6				

Подзадача В

Рассмотрим расширенную локальную сеть. Теперь она включает две пары «семафор-принтер»: (SEMAPHORE(1), SEMAPHORE(2), SERVER(1), SERVER(2)). Для каждого объекта CLIENT величина *Number_of_Servers* равна 2. Для того, чтобы использовать оба принтера, необходимо внести изменения в описание объекта CLIENT, которые приведены в документе 2. В документе 5 приведены измененные «Процедуры приема» сообщений «Wait» и «C_Job», а также описание состояний объектов в момент времени 0.

С учетом сказанного, в указанные ниже моменты времени объекты принимают следующие сообщения:

- в момент времени 0: CLIENT(1) получил сообщение «C_Job» от пользователя;
- в момент времени 0: CLIENT(2) получил сообщение «C_Job» от пользователя;
- в момент времени 4: SEMAPHORE(1) получил сообщение «Ready».

Что случится в заданной расширенной сети при указанных изменениях в описании объекта CLIENT? Выберите и отметьте правильный ответ из приведенных в документе шести вариантов.

Документ 5

Измененные «Процедуры приема» C_Job и Wait

```

procedure C_Job(User,Client,Document)
begin
  if State = SA
  then   State ← SB
        for   Semaphore_Index ← 1 step 1 until
Number_of_Servers

  Send("Are_you_open?(Client,SEMAPHORE(Semaphore_Index))")
  else
    if   State = SB
    then   Send("Try_later(Client,User,Client_is_busy)")
    else
      if   State = SC
      then
        Send("Try_later(Client,User,All_Servers_are_busy)")
  End

procedure Wait(Client,Client)
begin
  if   State = SB
  then

```

```

        Countdown ← Countdown - 1
    if      Countdown > 0
    then    Send("Wait(Client,Client)")
    else
    if      Countdown < 0
    then    Countdown ← 0
    else    for Semaphore_Index ← 1 step 1
until      Number_of_Servers

Send("Are_you_open?(Client,SEMAPHORE(Semaphore_Index))")
End

```

В момент времени 0 состояние сети следующее:

- объект: CLIENT(1), *Client.State* = SA, *Waiting_Period* = 2, *Number_of_Servers* = 2;
- объект: CLIENT(2), *Client.State* = SA, *Waiting_Period* = 1, *Number_of_Servers* = 2;
- объект: SEMAPHORE(1), *Semaphore.State* = S1;
- объект: SEMAPHORE(2), *Semaphore.State* = S1.

В следующие моменты времени были получены следующие сообщения:

- момент 0: CLIENT(1) получил сообщение «C_Job» от пользователя;
- момент 0: CLIENT(2) получил сообщение «C_Job» от пользователя;
- момент 4: SEMAPHORE(1) получил сообщение «Ready».

Что случится в расширенной сети согласно изменениям типа объекта CLIENT?

Отметьте правильный ответ.

- (a) Задание CLIENT(1) 'а будет напечатано на SERVER(1) 'е и SERVER(2) 'е.
Задание CLIENT(2) 'а не будет напечатано.
- (b) Задание CLIENT(1) 'а будет напечатано только на SERVER(1) 'е.
Задание CLIENT(2) 'а будет напечатано только на SERVER(2) 'е.
- (c) Задание CLIENT(1) 'а будет напечатано только на SERVER(1) 'е.
Задание CLIENT(2) 'а будет напечатано только на SERVER(1) 'е.
- (d) Задание CLIENT(1) 'а будет напечатано только на SERVER(2) 'е.
Задание CLIENT(2) 'а будет напечатано только на SERVER(2) 'е.
- (e) Задание CLIENT(1) 'а не будет напечатано.
Задание CLIENT(2) 'а будет напечатано на SERVER(1) 'е и SERVER(2) 'е.

Подзадача С

Оказалось, что внесенные в подзадаче В изменения в описание объекта CLIENT являются неэффективными для расширенной сети с более чем одной парой «семафор-принтер».

Подзадача С1

Внести изменения в описание объекта типа CLIENT (см. документ 2) так, чтобы расширенная локальная сеть из подзадачи В с более чем одной парой «семафор-принтер» могла функционировать следующим образом:

- объект CLIENT(*i*) может использовать любые серверы в сети, при этом в каждый момент времени он должен иметь не более одного выполняемого задания. Каждое задание печатается только один раз;
- объект CLIENT(*i*) посылает сообщение с идентификатором «Are_you_open?» нескольким семафорам, пока он не получит сообщение «Open» либо не получит определенное для данного объекта CLIENT пороговое количество ответных сообщений «Closed» (см. документ 3);
- когда объект CLIENT(*i*) получает соответствующее количество ответов «Closed», он ждет в течение периода времени *Waiting_Period*, после чего снова рассылает серии сообщений «Are_you_open?».

Подзадача С2

Измените описание типа объекта CLIENT так, чтобы объект CLIENT(*i*) мог иметь также несколько заданий, выполняемых одновременно разными серверами. Однако количество заданий на печать у объекта CLIENT(*i*) не должно превосходить величины CLIENT(*i*).Job_Maximum.

Идея решения

Подзадача А

Если в момент времени 4 объект CLIENT(1) получит от пользователя USER(1) сообщение «C_Job», то он тут же (согласно процедуре приема сообщения «C_Job» из документа 2) отправит обратно сообщение «Try_later(CLIENT(1),USER(1), All_Servers_are_busy)», поскольку будет в этот момент находиться в состоянии SC.

Если же в момент времени 4 сообщение «C_Job» получит объект CLIENT(2), то он отправит обратно сообщение «Try_later(CLIENT(2), USER(2), Client_is_busy)», так как будет в этот момент находиться в состоянии SB.

Ответ на вопрос А3 приведен в табл. 7.1.

Таблица 7.1

SEMAPHORE (1)

	Полученные сообщения	Сост./переход	Посланные сообщения
<i>time</i>			
0		S1	
1		S1	
2	Are_you_open?	S1 $\Rightarrow$ S2	Open
3	Are_you_open?	S2	Closed
4		S2	
5	Ready	S2 $\Rightarrow$ S1	
6	Are_you_open?	S1 $\Rightarrow$ S2	Open
	Are_you_open?	S2	Closed
7		S2	
8		S2	
9	Ready	S2 $\Rightarrow$ S1	
	Are_you_open?	S1 $\Rightarrow$ S2	Open
10		S2	
11	Are_you_open?	S2	Closed
12		S2	
13	Ready	S2 $\Rightarrow$ S1	

CLIENT (1)

	Полученные сообщения	Сост./переход	Count-down	Посланные сообщения
<i>time</i>				
0		SA	0	
1	C_Job	SA $\Rightarrow$ SB	0	Are_you_open?
2		SB	0	
3	Open	SB $\Rightarrow$ SC	0	S_Job
4		SC	0	
5	C_Ready	SC $\Rightarrow$ SA	0	
	C_Job	SA $\Rightarrow$ SB	0	Are_you_open?
6		SB	0	
7	Open	SB $\Rightarrow$ SC	0	S_Job
8		SC	0	
9	C_Ready	SC $\Rightarrow$ SA	0	
10	C_Job	SA $\Rightarrow$ SB	0	Are_you_open?
11		SB	0	
12	Closed	SB	0 $\Rightarrow$ 2	Wait
13	Wait	SB	2 $\Rightarrow$ 1	Wait

CLIENT (2)

	Полученные сообщения	Сост./переход	Count-down	Посланные сообщения
<i>time</i>				
0		SA	0	
1		SA	0	
2	C_Job	SA $\Rightarrow$ SB	0	Are_you_open?
3		SB	0	
4	Closed	SB	0 $\Rightarrow$ 1	Wait
5	Wait	SB	1 $\Rightarrow$ 0	Are_you_open?
6		SB	0	
7	Closed	SB	0 $\Rightarrow$ 1	Wait
8	Wait	SB	1 $\Rightarrow$ 0	Are_you_open?
9		SB	0	
10	Open	SB $\Rightarrow$ SC	0	S_Job
11		SC	0	
12		SC	0	
13	C_Ready	SC $\Rightarrow$ SA	0	

Подзадача В

Рассмотрим расширенную сеть с несколькими парами «семафор-принтер». Рассылка сообщений «Are_you_open?» одновременно всем семафорам приведет к тому, что все находящиеся в текущий момент в состоянии $S1$ семафоры откликнутся сообщением «Open» и перейдут в состояние $S2$, вывести из которого семафор можно только печатью задания на относящемся к нему принтере. Однако, как легко заметить, любое задание в расширенной сети (с измененными согласно документу 5 процедурами приема) печатается только один раз, поэтому все перешедшие в состояние $S2$ семафоры, кроме одного, просто «зависают» в нем, делая невозможным вывод на соответствующие принтеры.

В нашем примере расширенной сети, поскольку приоритет CLIENT(1)'а выше приоритета CLIENT(2)'а, в момент времени 1 «зависает» SEMAPHORE(2). Поэтому оба задания будут напечатаны на SERVER(1)'е, и правильный вариант ответа — (с).

Подзадача C1

Изменению подлежат процедуры приема сообщений «C_Job», «Closed» и «Wait»:

```
procedure C_Job(User, Client, Document)
begin
  if      State = SA
  then   State ← SB
        Semaphore_Index ← 1
        Send("Are_you_open?(Client,
        SEMAPHORE(Semaphore_Index))")
  else
    if      State = SB
    then   Send("Try_later(Client, User, Client_is_busy)")
    else
      if      State = SC
      then   Send("Try_later(Client, User,
        All_Servers_are_busy)")
  end

procedure Closed(Semaphore, Client)
begin
  Semaphore_Index ← Semaphore_Index + 1
  if Semaphore_Index ≤ Number_of_Servers
  then Send("Are_you_open?(Client,
        SEMAPHORE(Semaphore_Index))")
  else Countdown ← Waiting_Period
        Send("Wait(Client, Client)")
end

procedure Wait(Client, Client)
begin
  Countdown ← Countdown - 1
  if Countdown > 0
  then Send("Wait(Client, Client)")
  else Semaphore_Index ← 1
        Send("Are_you_open?(Client,
        SEMAPHORE(Semaphore_Index))")
end
```

Подзадача C2

Введем для типа объекта CLIENT новую внутреннюю переменную *Job_Count*, значением которой будет являться количество печатаемых для этого CLIENT'а заданий (начальное значение *Job_Count* равно нулю). По сравнению с подзадачей C1 необходимо изменить процедуры приема сообщений «Open» и «C_Ready». По-прежнему состояние SA означает, что CLIENT готов к приему очередного задания ($Job\_Count < Job\_Maximum$),

SB — что *CLIENT* ищет свободный принтер для поступившего на печать задания, а *SC* — что *CLIENT* занят и не может в текущий момент обрабатывать новое задание (*Job_Count = Job_Maximum*).

```
procedure Open(Semaphore, Client)
begin
    Send("S_Job(Client, Server, Document)")
    Job_Count ← Job_Count + 1
    if Job_Count < Job_Maximum
    then State ← SA
    else State ← SC
end
```

```
procedure C_Ready(Server, Client)
begin
    Job_Count ← Job_Count - 1
    if State = SC
    then State ← SA
end
```

Методические указания

Данная задача в «идейном» плане настолько отличается от проблематики международных (и не только) олимпиад по информатике, что в последующие годы эта тема не использовалась вовсе.

ЗАДАЧИ ВТОРОГО ТУРА

«Словесная игра»

Условие задачи

Словесные игры популярны дома и на телевидении. В нашем варианте игры каждая буква имеет цену, и вы должны составить из букв одно или более слов, дающих максимальную суммарную стоимость. До тех пор, пока возможно, вы подбираете все известные вам слова, иногда проверяя правописание и вычисляя их стоимость. Очевидно, это лучше сделать с помощью компьютера.

Даны цены букв (рис. 7.3), список английских слов и набор букв. Найдите в словаре слова или пары слов с наибольшей суммарной стоимостью, которые можно составить из заданного набора букв.

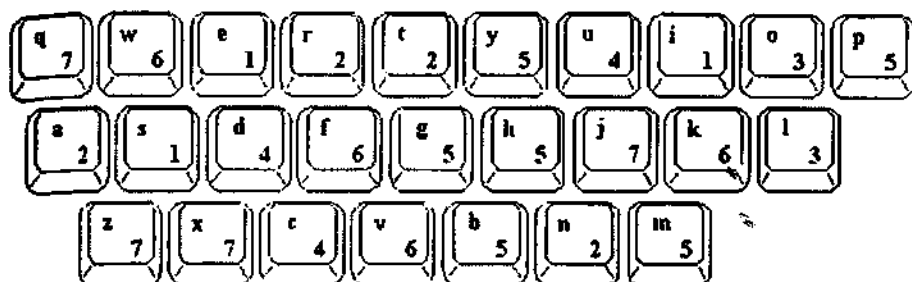


Рис. 7.3. Алфавит и цена букв

Входные данные. Входной файл INPUT.TXT содержит одну строку с маленькими латинскими буквами (допустимые буквы — от «a» до «z»). Строка содержит от трех до семи букв включительно в произвольном порядке.

Файл-«словарь» WORDS.TXT содержит не более 40000 слов. В конце этого файла находится строка с единственным символом «.» (точка). Каждая из остальных строк содержит слово длиной от трех до семи маленьких латинских букв включительно. Слова в файле упорядочены по алфавиту.

Выходные данные. В первую строку выходного файла OUTPUT.TXT ваша программа должна написать наибольшую возможную суммарную стоимость (подзадача A), а в следующих строках — перечислить все допустимые слова и/или пары слов из словаря WORDS.TXT с такой стоимостью (подзадача B). Каждое слово или пару выводите в отдельную строку. Используйте цены букв, заданные на рис. 7.3.

Если из заданных букв можно составить пару, эти слова надо напечатать, разделив пробелом. Не повторяйте пары; например, «rag prom» и «prom rag» — одна пара, она должна быть напечатана один раз.

Пример ввода и вывода.

WORDS.TXT	INPUT.TXT	OUTPUT.TXT
profile	Prmgroa	24
program		program
prom		prom rag
rag		
ram		
rom		

Примечания:

- 1) файл WORDS.TXT не содержит повторяющихся слов;
- 2) буква не может встречаться в выбранном слове или паре слов большее количество раз, чем в файле INPUT.TXT;
- 3) слово может образовывать пару само с собой.

Идея решения

Эта задача логически разбивается на две части. Первая — построение (генерация) слов из заданных букв. Это известная комбинаторная задача о генерации перестановок, несколько усложненная тем, что возможна пара слов. Вторая — поиск слова в словаре. Объем словаря таков, что он размещается в оперативной памяти, тем более что размещение данных словаря в памяти можно сделать по маске — мы берем только те слова, в которых есть буквы из исходной строки. Словарь упорядочен, поэтому метод поиска очевиден: это бинарный поиск. Вычисление суммарной стоимости букв слова, найденного в словаре, и сохранение результата являются уже техническими деталями.

Методические указания

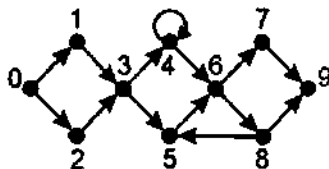
Задачу (при заданных ограничениях и существовании соответствующего обрамления — подготовленного словаря, проверочной программы, тестов) можно использовать как учебную при изучении комбинаторных алгоритмов и методов поиска. Ее методическая ценность заключается в исследовательской составляющей.

Самостоятельные исследования

Задание для самостоятельного исследования, как говорится, «лежит на поверхности». Максимально допустимые семь символов в строке — очень скромная цифра, так же как и объем словаря. Снятие этих ограничений, — а точнее, увеличение их верхних значений, поднимает огромный пласт не решенных и по сей день в полном объеме проблем в информатике.

«Уличная гонка»**Условие задачи**

На рис. 7.4 изображен пример плана улиц для гонки. Вы видите точки, помеченные числами от 0 до n (где $n = 9$), а также стрелки, соединяющие их. Точка 0 является стартовой, точка n — финиш-

Рис. 7.4. План улиц с 10 точками¹

ной. Стрелками представлены улицы с односторонним движением. Участники гонки передвигаются от точки к точке по улицам только в направлении стрелок. В каждой точке участник гонки может выбрать любую из исходящих стрелок.

Назовем план улиц «хорошим», если он обладает следующими свойствами:

- каждая точка плана может быть достигнута со старта;
- финиш может быть достигнут из любой точки плана;
- у финиша нет исходящих стрелок.

Для достижения финиша участник не обязан пройти через все точки. Однако некоторые точки невозможно обойти. Назовем их «неизбежными». В примере такими точками являются точки 0, 3, 6, 9. Для заданного «хорошего» плана ваша программа должна определить множество «неизбежных» точек (за исключением старта и финиша), которые должны посетить все участники (подзадача А).

Предположим, что гонка должна проводиться за два последовательных дня. Для этой цели план должен быть разбит на два «хороших» плана, по одному на каждый день. В первый день стартом является точка 0, а финишем служит некая «точка разбиения». Во второй день старт находится в этой точке разбиения, а финиш — в точке m . Для заданного «хорошего» плана ваша программа должна определить множество всех возможных точек разбиения (подзадача В).

Точка S является точкой разбиения для «хорошего» плана C , если S отличается от старта и финиша C , и план разбивается ею на два «хороших» плана без общих стрелок и с единственной общей точкой S . В примере точкой разбиения является точка 3.

Входные данные. В файле INPUT.TXT описан «хороший» план, содержащий не более 50 точек и не более 100 стрелок. В файле имеется $n+1$ строка. Первые n строк содержат конечные точки стрелок, исходящих, соответственно, из точек от 0 до $n-1$. Каждая из этих строк заканчивается числом -2 . В последней строке содержится число -1 .

Выходные данные. Ваша программа должна записать две строки в файл OUTPUT.TXT. Первая строка должна содержать количество «неизбежных» точек в заданном плане, после чего в той же строке должны следовать номера этих точек в любом порядке (подзадача А). Вторая строка должна содержать количество точек разбиения в заданном плане, за которым в той же строке должны следовать номера этих точек в любом порядке (подзадача В).

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
1 2 -2	2 3 6
3 -2	1 3
3 -2	
5 4 -2	
6 4 -2	
6 -2	
7 8 -2	
9 -2	
5 9 -2	
-1	

Идея решения

Легко заметить, что точка является «неизбежной» тогда и только тогда, когда ее удаление ведет к невозможности добраться от старта до финиша.

Все точки, в которые можно попасть после удаления некоторой заданной точки, находятся с помощью поиска в глубину. Пусть A — множество точек, в которые можно попасть со старта, если удалить точку k ($0 < k < n$), а B — множество всех остальных точек (включая саму точку k). Точка k является «неизбежной» тогда и только тогда, когда финиш (точка n) принадлежит множеству B , и точкой разбиения тогда и только тогда, когда, во-первых, нет стрелок от точек из B к точкам из A и, во-вторых, нет стрелок от точек из A к точкам из B , за исключением стрелок, ведущих в точку k . Стрелки от точки k к ней же самой допускаются, так как их можно включить в план для второго дня гонки. Заметим, что точка разбиения всегда является «неизбежной» точкой.

Алгоритм решения этой задачи тогда можно записать так:

```

Const NMax = 49; {Максимальный номер точки}
Var A : Array [0..NMax,0..NMax] Of Boolean;
{A[i,j]=True означает, что есть стрелка от точки i
к точке j}
    New : Array [0..NMax] Of Boolean; {Массив
для пометок}
    k : Integer; {Номер исследуемой точки}
    Procedure WG(i : Integer); {Поиск в глубину,
минуя точку k}

```

```
Var j : Integer;  
Begin  
  New[i]:=False; {Помечаем текущую точку}  
  For j:=0 To n Do {Ищем еще не помеченные  
точки, в которые можно пойти}  
    If (j<>k) And A[i,j] And New[j] Then WG(j);  
  End;  
  
Var i,j : Integer;  
  Ok : Boolean;  
Begin  
  For k:=1 To n-1 Do Begin {Исследуем по очереди  
все промежуточные точки}  
    For i:=0 To n Do New[i]:=True;  
    WG(0); {Помечаем все точки, в которые можно  
попасть, минуя точку k}  
    If New[n] Then <Точка k неизбежна>;  
{Все точки разбиты на два множества: A (помеченные,  
т. е. New[x]=False) и B (непомеченные, т. е. New[x]=True)}  
    Ok:=True;  
    For i∈A Do  
      For j∈B Do  
        If A[j,i] Or A[i,j] And (j<>k) Then Ok:=False;  
    If Ok Then <Точка k – точка разбиения>;  
  End;  
  <Выводить все неизбежные точки>;  
  <Выводить все точки разбиения>;  
End.
```

Методические указания

Это — задача по теме «Алгоритмы на графах». В ней в завуалированном виде ставится вопрос об анализе связности графа. Данная проблема рассматривается в учебной литературе по дискретной математике*. Вариант разбора приведенной здесь задачи дан в книге М. С. Долинского**.

Самостоятельные исследования

1. Есть n карточек. На каждой из них черными чернилами написан ее уникальный номер — число от 1 до n . Также на каждой карточке красными чернилами написано еще одно це-

* Окулов С. М. Дискретная математика: теория и практика решения задач по информатике. М.: БИНОМ. Лаборатория знаний, 2007.

** Долинский М. С. Алгоритмизация и программирование на TURBO PASCAL: от простых до олимпиадных задач. СПб.: Питер, 2005. С. 195–201.

лое число, лежащее в промежутке от 1 до n (некоторыми одинаковыми «красными» числами могут помечаться несколько карточек).

Необходимо выбрать из заданных n карточек максимальное количество карточек так, чтобы множества «красных» и «черных» чисел на них совпадали.

Пример. Допустим, что $n = 5$ и эти 5 карточек помечены «черными» числами 1, 2, 3, 4, 5 и, соответственно, «красными» числами 3, 3, 2, 4, 2. Тогда «правильными» будут карточки с «черными» номерами 2, 3 и 4, поскольку множество «красных» номеров в данном примере именно такое — 2, 3, 4*.

2. Все команды, участвовавшие в «ACM World Finals», приглашаются на большой банкет, который устраивается после церемонии награждения. Чтобы члены разных команд общались как можно больше, за одним столом не разрешается сидеть двум членам одной и той же команды.

Определите по заданному количеству членов каждой команды (включая участников, тренеров, запасных и гостей) и количеству мест за каждым столом, можно ли рассадить команды так, как описано выше. Если это возможно, то выведите один из возможных вариантов их рассаживания. Если существует несколько вариантов, то допускается любой.

Входные данные. Входной файл может состоять из нескольких тестовых блоков. Первая строка каждого тестового блока содержит два целых числа — $1 \leq m \leq 70$ и $1 \leq n \leq 50$, задающих количества команд и столов, соответственно. Вторая строка каждого тестового блока содержит m чисел; i -е число m_i указывает количество членов команды под номером i (количество членов любой команды не превышает 100 человек). Третья строка содержит n чисел, причем j -е число n_j , где $2 \leq n_j \leq 100$, обозначает количество мест за j -м столом.

Входные данные заканчиваются тестовым блоком, содержащим нули в качестве m и n .

Выходные данные. Для каждого тестового блока выведите строку, содержащую число 1 или 0, указывающее, существует ли корректный способ рассаживания членов команд. В случае положительного ответа выведите m строк, i -я из которых содержит номера столов (от 1 до n) для членов команды номер i **.

* Долинский М. С. Алгоритмизация и программирование на TURBO PASCAL: от простых до олимпиадных задач. СПб.: Питер, 2005. С. 185–191.

** Скиена С. С. Ревилла М. А. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. М.: КУДИЦ-ОБРАЗ, 2005. С. 266–267.

«Провода»

Провода и выключатели

На рис. 7.5 изображен кабель, содержащий три провода, соединяющие концы *A* и *B*. На конце *A* находятся три провода, пронумерованные числами 1, 2 и 3. На конце *B* провода соединены с пронумерованными выключателями: провода 1 и 3 соединяются с выключателем 3, а провод 2 соединяется с выключателем 1.

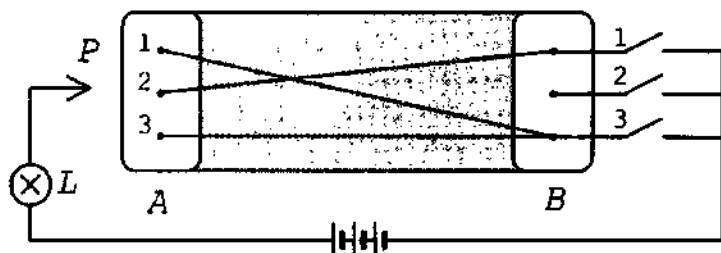


Рис. 7.5. Пример кабеля, содержащего три провода

В общем случае кабель состоит из m проводов ($1 \leq m \leq 90$), пронумерованных на конце *A* числами от 1 до m , а на конце *B* находятся m выключателей, также пронумерованных числами от 1 до m . Каждый провод соединяется ровно с одним выключателем. К каждому выключателю подходит нулевое или большее количество проводов.

Измерения

Напишите программу, которая путем некоторых запросов определяет, какие провода с какими выключателями соединены. Каждый выключатель может находиться в одном из двух состояний: он либо соединяет провод (выключатель включен), либо разрывает (выключатель выключен). В начальный момент времени все выключатели выключены. В процессе измерений провод может быть проверен при помощи щупа *P*: щуп соединяется с тестируемым проводом, и лампочка *L* загорается в том и только в том случае, если этот провод соединен с одним из включенных выключателей на конце.

Программа должна работать в диалоговом режиме, при этом ввод и вывод осуществляются через стандартные устройства (standard input и standard output соответственно), которыми по умолчанию являются клавиатура и экран монитора. В начале программа считывает одну строку, содержащую количество проводов m . Далее она последовательно выводит команды, которые бывают трех видов. Команда задается в отдельной

строке и начинается с имени команды — заглавной буквы латинского алфавита: *T* (*Test a wire* — «проверить провод»), *C* (*Change a switch* — «изменить состояние выключателя») и *D* (*Done* — «конец работы»). За командой *T* записывается номер провода, за командой *C* — номер выключателя, за командой *D* — результат работы программы — список из m номеров выключателей, в котором i -й элемент задает номер выключателя, к которому присоединяется провод с номером i .

После выдачи команд *T* или *C* программа должна считать строку, содержащую ответ на этот запрос. На запрос *T* выдается ответ *Y* (*Yes* — «Да»), если лампочка *L* загорелась, и ответ *N* (*No* — «Нет») — в противном случае. Запрос *C* изменяет состояние соответствующего выключателя на противоположное. На команду *C* выдается ответ *Y* (*Yes* — «Да»), если новое положение выключателя «включен», и ответ *N* (*No* — «Нет»), если новое положение — «выключен». Ответ на команду *C* выдается только в качестве подтверждения. На команду *D* ответ не выдается.

Ваша программа должна выводить последовательность из не более чем 900 команд, последней из которых является команда *D*, содержащая найденную схему соединений.

Пример возможного ввода и вывода программы для изображенного на рис. 7.4 кабеля:

Standard input	Standard output
3	C 3
Y	T 1
Y	T 2
N	T 3
Y	C 3
N	C 2
Y	T 2
N	D 3 1 3

Примечание. Не подключайте к вашей программе модуль CRT!

Идея решения

В основе решения лежит идея бинарного поиска — деления данных на равные части и решения задачи для каждой из частей. Пусть имеется некоторое множество проводов, соединенных с m выключателями с номерами $1, 1 + 1, \dots, r$, причем все выключатели находятся в одном положении («включено» или

«выключено») и для каждого провода требуется определить тот выключатель, с которым он соединен.

Если $m = 1$, то задача очевидна.

Пусть $m > 1$. Изменим состояние первых $[m/2]$ выключателей на противоположное, а затем проверим каждый из проводов. По полученным ответам множество проводов разделяется на два непересекающихся подмножества: из тех проводов, которые соединены с первыми $[m/2]$ выключателями, и тех, которые соединены с $m - [m/2]$ оставшимися. Тем самым исходная задача сведена к двум аналогичным, но более простым (с меньшим числом выключателей).

Описанный алгоритм можно записать в виде рекурсивной процедуры:

```

Procedure Rec(s : WSet; l, r:Integer; on:Boolean);
{s – множество проводов, соединенных с выключателями
l, l+1, ..., r, имеет тип Type WSet = Set Of 1..MMax}
{on = True, если эти выключатели находятся
в положении "включено", и False – в противном
случае}
Var x, i : Integer;
    Y, N : WSet;
    ch : Char;
Begin
    If s=[] Then Exit;
    If l=r Then Begin
        For i:=1 To m Do If i In s Then A[i]:=1;
        {Фиксируем факт о том, что все провода из s
соединены с выключателем l. Массив A имеет тип
A : Array [1..MMax] Of Integer, где MMax –
константа, равная 90}
        Exit;
    End;
    x:=(l+r-1) Div 2;
    For i:=1 To x Do Begin {Меняем состояние
выключателей l, l+1, ..., r на противоположное}
        WriteLn('C',i);
        ReadLn(ch);
    End;
    Y:=[]; N:=[];
    For i:=1 To m Do
        If i In s Then Begin
            WriteLn('T',i);
            ReadLn(ch);
            If ch='Y' Then Y:=Y+[i]

```

Else $N := N + [i];$

End;

{Тестируем все провода из s . Пусть y — множество тех проводов, при проверке которых был получен ответ "Y", n — множество тех проводов, при проверке которых был получен ответ "N">}

If on Then Begin

$\text{Rec}(N, l, x, \text{False});$

$\text{Rec}(Y, x+1, r, \text{True});$

End

Else Begin

$\text{Rec}(Y, l, x, \text{True});$

$\text{Rec}(N, x+1, r, \text{False});$

End;

End;

Вызов этой процедуры должен производиться следующим образом:

$\text{Rec}([1..m], 1, 90, \text{False})$

Примечание. Обсудим появление числа 900 в условии задачи и вопрос о том, «укладывается» ли алгоритм в это количество команд. Пусть условного оператора **If** $S = []$ **Then** **Exit** в процедуре Rec нет. Обозначим через $C(m)$ количество команд C , используемых алгоритмом для m выключателей. Тогда для чисел $C(m)$ верны следующие рекуррентные соотношения:

$$\begin{cases} C(2 * k) = (k + 2 * C(k)); \\ C(2 * k + 1) = k + C(k) + C(k + 1). \end{cases}$$

Учитывая, что $C(1) = 0$, получаем: $C(90) = 275$. Подсчитаем количество команд T , выдаваемых алгоритмом при $m = 90$. Исходная задача размера 90 разбивается на 2 подзадачи размера 45, каждая из которых, в свою очередь, разбивается на подзадачу размера 22 и подзадачу размера 23, и т. д. Этот процесс наглядно изображается в виде дерева (см. рис. 7.6а). Вызов процедуры Rec для m выключателей здесь обозначен как $\text{Rec}(m)$. Если же на каждом из уровней дерева рисовать лишь задачи различных размеров, то мы получим граф, изображенный на рис. 7.6б.

При переходе на каждый следующий уровень дерева (кроме, может быть, последнего) проверяются все 90 проводов. Заметим, что в процессе решения исходной задачи для 90 выключателей на шестом уровне приходится решать 6 подзадач для 2 выключателей (то есть происходит 6 вызовов $\text{Rec}(2)$). Рассмотр-

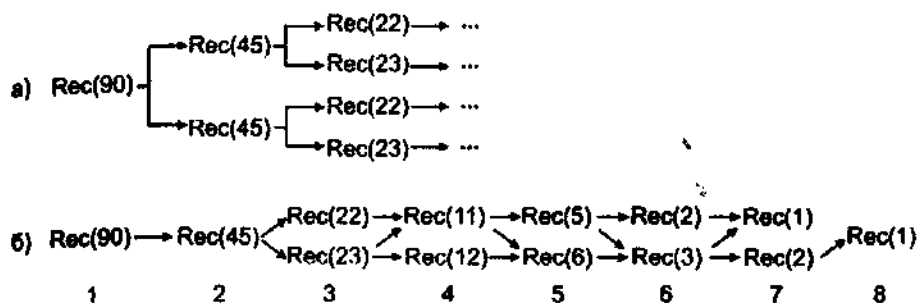


Рис. 7.6. Пример заботы процедуры Rec для 90 переключателей.
Цифрами 1–8 обозначены номера уровней

рим эти вызовы. Если в каком-то из них множество s не пусто, то ни один провод из s не тестируется при переходе на восьмой уровень; следовательно, экономится как минимум одна команда T . В том случае, когда множество s оказалось пустым, происходит немедленный выход из процедуры Rec (вспомним о ее первом операторе) — значит, нет одной команды C . Итак, на каждом из шести вызовов $\text{Rec}(2)$ экономится не менее одной команды. Таким образом, общее количество команд при $m = 90$ не превосходит $(275 + 90 \cdot 7 + 1) - 6 = 900$. Если же $m \leq 89$, то оно не превзойдет $275 + 89 \cdot 7 + 1 = 899$. Тем самым показана корректность приведенного алгоритма.

Методические указания

Эту задачу рекомендуется использовать при изучении темы «Сортировка и поиск». Режим «ручной» трассировки для небольших значений m поможет учащимся осознать суть решения.

Самостоятельные исследования

1. *Старательный библиотекарь.* На длинном столе лежат книги, первоначально расположенные в m стопок. Старательный библиотекарь каждое утро берет самую левую стопку и раскладывает все лежащие в ней книги во все следующие по одной штуке. Если стопок меньше, чем раскладываемых книг, то оставшиеся книги образуют новые стопки. Для примера проследим несколько шагов этого процесса:

- первоначально: 3 7 4 9 2 6;
- после шага 1: 8 5 10 2 6;
- после шага 2: 6 11 3 7 1 1 1 1;
- после шага 3: 12 4 8 2 2 2 1; ...

Определите, сколько книг будет в каждой стопке по истечении n дней*.

2. Дана последовательность целых чисел $a_1, a_2, \dots, a_n$ произвольного знака, по модулю не превосходящих 10000. Неубывающей подпоследовательностью последовательности $a_1, a_2, \dots, a_n$ длины k мы назовем такой набор чисел $a_{j_1}, a_{j_2}, \dots, a_{j_k}$, что $1 \leq j_1 < j_2 < \dots < j_k \leq n$ и $a_{j_1} \leq a_{j_2} \leq \dots \leq a_{j_k}$. Требуется найти количество элементов в самой длинной неубывающей подпоследовательности для заданной последовательности**.

* Воронин А. В., Кузнецов В. А., Корзун Д. Ж. Командные олимпиады по программированию: организация, задачи и решения. Петрозаводск: ПетрГУ. 2001. С. 141.

** Арсак Р. Программирование игр и головоломок. М.: Мир, 1994.

ЗАДАЧИ ПЕРВОГО ТУРА

«Игра»

Условие задачи

Рассмотрим следующую игру между двумя игроками. На игровой доске записана последовательность положительных целых чисел. Игроки ходят по очереди. Ход заключается в том, что игрок выбирает число, расположенное на левом или на правом конце последовательности. Выбранное число стирается с доски. Игра заканчивается, когда на доске не остается чисел. Первый игрок выигрывает, если сумма выбранных им чисел не меньше, чем сумма чисел, выбранных вторым игроком. Второй игрок играет наилучшим образом. Первый игрок всегда ходит первым.

Известно, что если в начале игры на доске записано четное количество чисел, то у первого игрока есть выигрышная стратегия. Требуется написать программу, которая реализует выигрышную стратегию первого игрока.

Ответы второго игрока выдает имеющийся в вашем распоряжении модуль Play. В модуле реализованы 3 процедуры: StartGame, MyMove, YourMove. В самом начале ваша программа должна инициализировать игру, вызвав процедуру StartGame (без параметров). Когда ваша программа делает очередной ход, она выполняет инструкцию MyMove (L) или MyMove (R) в зависимости от того, какое число она выбирает: «L» — левое или «R» — правое. Чтобы получить ответ второго игрока, ваша программа должна выполнить инструкцию YourMove (C), где C — переменная символьного типа (в C/C++ вы должны записать YourMove (&C)). Процедура YourMove записывает ответ второго игрока в заданную переменную C («L», если выбирается левое число; «R» — если правое).

Входные данные. Первая строка файла исходных данных с именем INPUT.TXT содержит длину начальной последовательности n (n — четное и $2 \leq n \leq 100$). Далее следуют n строк, в каждой из которых содержится одно число. Эти числа задают начальные значения, записанные на доске в порядке слева направо. Числа на доске не превышают 200.

Выходные данные. Когда игра заканчивается, ваша программа должна записать результат игры в файл с именем

OUTPUT.TXT. Файл содержит два числа, записанные в одной строке. Первое число задает сумму чисел, выбранных первым игроком, второе число — сумму чисел, выбранных вторым игроком. Ваша программа должна сыграть игру с модулем Play, а вывод должен соответствовать сыгранной игре.

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
6	15 14
4	
7	
2	
9	
5	
2	

Обсуждение задачи

Аддитивность задачи (а значит, и динамическая схема ее решения) здесь явно просматривается. Пусть последовательность чисел хранится в массиве x . Введем матрицу $d[1..n, 1..n]$, где элемент $d[i, j]$ обозначает максимальную сумму выигрыша на отрезке $[i, j]$. Очевидно, что $d[i, i] = x[i]$. Затем сформируем элементы матрицы, соответствующие длине «два», «три» и т. д., пока не будет заполнена вся «верхняя» часть матрицы. Схема получения элемента $d[i, j]$ приведена на рис. 8.1.

Обозначение $sum(i, j)$ на рис. 8.1 — это ни что иное, как сумма элементов на отрезке $[i, j]$. Главное заключается в том, что элементы $d[i-1, j]$ и $d[i, j-1]$ уже подсчитаны и без пересчета (аддитивно) входят в подсчет $d[i, j]$.

Ответ задачи — максимальный выигрыш — равен элементу $d[1, n]$. Одновременно с заполнением матрицы d мы заполняем матрицу p , в которой фиксируем признаки хода (с левого или правого конца), дающего максимальный выигрыш.

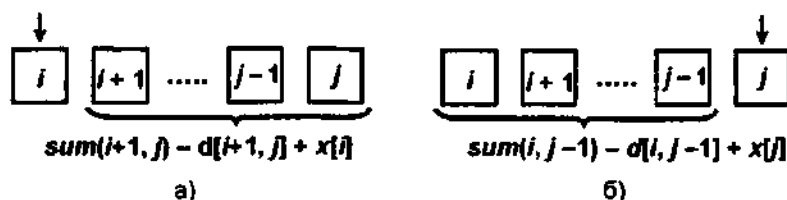


Рис. 8.1. Схема формирования элемента $d[i, j]$

Обратимся к примеру из формулировки задачи. Для удобства с помощью простой конструкции:

```
w[0]:=0;
For i:=1 To n Do w[i]:=w[i-1]+x[i];
```

зафиксируем в массиве w суммы элементов отрезков $[1..i]$. Возьмем для примера $w = (4, 11, 13, 22, 27, 29)$.

Пусть требуется определить максимальный выигрыш на отрезке $[i, j]$. Если будет взят левый элемент, то сумма выигрыша составит $w[j]-w[i]+x[i]-d[i+1, j]$. Значение $(w[j]-w[i])$ — это сумма элементов отрезка $[i+1, j]$, $d[i+1, j]$ — максимальный выигрыш второго игрока, а он играет наилучшим образом, и $x[i]$ — вклад в выигрыш первого игрока при этом ходе. Аналогично подсчитывается значение максимального выигрыша и при взятии правого элемента отрезка $[i, j]$. Максимальный элемент из этих двух значений мы запоминаем в матрице d , а в матрице p сохраняется признак правого или левого хода.

Для примера из формулировки задачи имеем:

$$d = \begin{bmatrix} 4 & 7 & 6 & 16 & 11 & 18 \\ -1 & 7 & 7 & 11 & 16 & 14 \\ -1 & -1 & 2 & 9 & 7 & 11 \\ -1 & -1 & -1 & 9 & 9 & 11 \\ -1 & -1 & -1 & -1 & 5 & 5 \\ -1 & -1 & -1 & -1 & -1 & 2 \end{bmatrix} \quad \text{и} \quad p = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Методическое отступление. Используя «набросок» процедуры формирования d и p , целесообразно в «ручном» режиме «просчитать» игру для конкретных примеров. При этом следует обратить внимание на очередность заполнения элементов матрицы d . Это можно считать контрольным вопросом на понимание рекурсивной реализации логики.

```
Procedure Calc(i,j:Integer);
Var s1,s2:Integer;
Begin
  If d[i,j]=-1 Then
    If i=j Then d[i,j]:=x[i] {Заполняем элемент
на диагонали матрицы d}
    Else Begin
      Calc(i+1,j);
      Calc(i,j-1);
```

```

    s1: = w[j]-w[i]+x[i]-d[i+1,j]; {Вычисляем сумму
выигрыша в случае взятия числа с левого конца}
    s2: = w[j-1]-w[i-1]+x[j]-d[i,j-1]; {Вычисляем
сумму выигрыша в случае взятия числа с правого конца}
    If s1>s2 Then Begin d[i,j]:=s1; p[i,j]:=0;
End {Запоминаем. В матрице p значение "нуль"
означает признак "L", единица — "R"}
    Else Begin d[i,j]:=s2; p[i,j]:=1; End
End
End;

```

Методическое отступление. Мы обсудили один из вариантов решения. А затем можно «ошарашить» учащихся вопросом: есть ли более простой и изящный способ решения?

Если задача проверена для нескольких наборов входных данных, то можно установить некую закономерность. Подсчитаем суммы элементов исходной последовательности, находящихся на нечетных и четных местах. Для примера из формулировки задачи они равны: $4 + 2 + 5 = 11$ и $7 + 9 + 2 = 16$. Если первым ходом мы возьмем число, расположенное на четном месте, то противник должен будет брать число, находящееся на нечетном месте. Затем мы опять берем число на четном месте и т. д. Выигрыш при этом нам обеспечен, но не максимальный (как в разобранный стратегии), — однако это и не требовалось по условию задачи.

Уместен и еще один дополнительный вопрос: по какой стратегии игры получен результат 15:14 для примера из формулировки задачи?

Методические указания

Эта задача является типичной при изучении темы «Динамическое программирование». Ее более простые варианты решения разнообразят занятие и придают ему более «живой» характер.

Самостоятельные исследования

1. Снятие ограничения на четность n приводит к небольшому мини-исследованию по поиску выигрышной стратегии и достойному проигрышу при ее отсутствии.
2. Разработка модуля поддержки игры со стороны компьютера повысит привлекательность работы с задачей.
3. Возможен вариант построения полного дерева игры и реализация некой схемы перебора всех вариантов. Оцените время работы для различных значений n .

«Детали»

Условие задачи

На фабрике функционирует производственная линия. Обработка каждой детали требует двух операций: сначала выполняется операция «А», а затем — операция «В». Для выполнения каждой из операций существует определенное количество станков.

Таблица 8.1

Содержимое входного контейнера:	□ □ □ □ □ □ □ □ □ □
Станки для операции «А»:	A1 A2
Содержимое промежуточного контейнера:	■ ■ ■ ■
Станки для операции «В»:	B1 B2 B3
Содержимое выходного контейнера:	◆ ◆ ◆ ◆ ◆ ◆ ◆ ◆ ◆ ◆

В табл. 8.1 приведен пример организации производственной линии, которая работает следующим образом. Станок, выполняющий операцию «А», берет деталь из входного контейнера, исполняет операцию «А» и помещает деталь в промежуточный контейнер. Станок, выполняющий операцию «В», берет деталь из промежуточного контейнера, исполняет операцию «В» и помещает деталь в выходной контейнер. Все станки работают параллельно и независимо друг от друга. Вместимость каждого из контейнеров не ограничена. Станки имеют различные временные характеристики обработки деталей, каждый станок имеет свое время обработки.

Напишите программу, которая определяет минимальный период времени, необходимый для выполнения операции «А» над всеми n деталями при условии, что они становятся доступными в момент времени 0 (подзадача А). Необходимо также вычислить минимальный период времени, необходимый для выполнения обеих операций «А» и «В» над всеми n деталями (подзадача В).

Входные данные. Файл INPUT.TXT состоит из пяти строк и содержит набор целых положительных чисел. В первой строке содержится число n — количество деталей ($1 \leq n \leq 1000$). Во второй строке находится число M_1 — количество станков, вы-

полняющих операцию «А» ($1 \leq M_1 \leq 30$). В третьей строке содержатся M_1 целых чисел, определяющих время обработки детали соответствующими станками, выполняющими операцию «А». В четвертой и пятой строках находятся соответственно число M_2 — количество станков, выполняющих операцию «В» ($1 \leq M_2 \leq 30$), и M_2 целых чисел, определяющих время обработки детали соответствующими станками. Время обработки детали на каждом станке включает время, необходимое для перемещения детали из контейнера к станку и со станка в контейнер. Время обработки для каждого станка задается целым числом от 1 до 20.

Выходные данные. Ваша программа должна сформировать две строки в файле OUTPUT.TXT. Первая строка должна содержать одно положительное целое число — решение подзадачи А. Вторая строка должна содержать решение подзадачи В.

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
5	3
2	5
1 1	
3	
3 1 4	

Обсуждение задачи

Рассмотрим выполнение операции «А».

Время обработки деталей станками записано в массиве a (его тип — `TVect=Array[0..NMax] Of Integer`, где $NMax$ — константа, описывающая максимальное количество станков). В элементе $a[0]$ мы будем хранить количество станков, выполняющих операцию «А».

Используем идею бинарного поиска для нахождения времени завершения станками первой операции. Зададим нижнюю и верхнюю границы интервала времени завершения операции, по которым определяется (методом половинного деления) время завершения q . Зная время завершения операции q , можно подсчитать количество деталей, обрабатываемых каждым станком, и общее количество деталей t (как сумму), обрабатываемых всеми станками. Если количество обрабатываемых деталей меньше заданного (n), то следует изменять нижнюю границу поиска, в противном случае — верхнюю границу.

$l:=0$; $r:=SMax$; {Начальный интервал поиска, $SMax$ — константа, определяющая максимальное значение границы}

While $(l+1)<r$ **Do Begin**

$q:=(l+r) \text{ Div } 2$; {Время завершения есть половина суммы значений границ}

$t:=0$; {Подсчитываем количество деталей, обрабатываемых станками для операции "A"}

For $i:=1$ **To** $a[0]$ **Do** $t:=t+q \text{ Div } a[i]$; {Количество деталей, обрабатываемых станком с номером i }

If $t<n$ **Then** $l:=q$ **Else** $r:=q$ {Изменяем время завершения выполнения операции}

End;

$Soll:=r$; {Ответ подзадачи A}

Еще раз подчеркнем суть идеи решения: здесь задается (подбирается, в данном случае на основе бинарной схемы) время завершения выполнения операции «A» и проверяется, можно ли уложиться в это время. Сформулируем вопрос: как подобрать минимальное время выполнения операции «B» и как использовать эту же идею для второй части задачи?

Зная время r , для каждого станка подсчитывается время поступления в промежуточный контейнер деталей, обработанных этим станком. Зафиксируем их в массиве x , а после выполнения расчетов для всех станков отсортируем его элементы в порядке неубывания.

$x[0]:=0$; {Количество деталей}

For $i:=1$ **To** $a[0]$ **Do** {Цикл по станкам}

For $j:=1$ **To** $r \text{ Div } a[i]$ **Do Begin** {Цикл по деталям, обработанным станком с номером i }

$x[0]:=x[0]+1$; {Увеличиваем количество деталей}

$x[x[0]]:=j*a[i]$; {Подсчитываем время завершения обработки детали с номером j }

End;

<сортировка элементов массива x >;

$x[0]:=n$; {Количество деталей равно n , предыдущая обработка может дать на единицу больше}

Итак, нам известно время поступления деталей в промежуточный контейнер. Попробуем подобрать время завершения выполнения операции «B», по-прежнему используя идею бинарного поиска. Время завершения работы станков фиксируем в массиве y . Первоначально значения элементов равны q , а затем для каждой детали (начиная с последней) отыскивается

станок (первый возможный — элемент «жадного алгоритма»), на котором она может быть обработана. Если такого станка нет, то следует увеличить нижнюю границу поиска; иначе из времени завершения работы станка надо вычесть время обработки детали и перейти к следующей детали. Когда для всех деталей найдены станки, нужно уменьшить верхнюю границу поиска. Процесс обработки заканчивается стандартным образом для бинарного поиска.

```

l:=0; r:=SMax;
While (l+1)<r Do Begin
  q:=(l+r) Div 2;
  For i:=1 To b[0] Do y[i]:=q; {q — время
    завершения операции "В", фиксируем его для
    каждого станка (b[0] — количество станков)}
  ok:=True; {Считаем, что детали могут быть
    обработаны за это время}
  For i:=x[0] DownTo 1 Do Begin {Цикл по деталям
    (в массиве x время поступления в промежуточный
    контейнер), находящимся в промежуточном контейнере.
    Начинаем "укладку" с последней детали, и в этом
    состоит "изюминка" решения}
    j:=1; {Находим станок, на котором могла бы
    быть обработана деталь с номером i}
    While (j<=b[0]) And ((x[i]+b[j])>y[j]) Do j:=j+1;
    {Если сумма времени поступления детали
    в промежуточный контейнер и времени обработки
    детали на станке с номером j превышает время
    завершения работы станка с номером j (точнее —
    времени освобождения станка j), то деталь i
    не может быть обработана на станке j}
    If j<=b[0] Then y[j]:=y[j]-b[j] {Найден
    станок, который может обработать деталь}
    Else Begin
      ok:=False; {Деталь с номером i не может
      быть обработана ни одним станком}
      Break
    End
  End;
End;
If Not ok Then l:=q Else r:=q {Если
не укладываемся с обработкой во время q,
то увеличиваем нижнюю границу, иначе —
уменьшаем верхнюю}
End;
Sol2:=r; {Ответ подзадачи "В"}

```

Методическое отступление. Объяснив логику решения, целесообразно продемонстрировать ее на конкретном примере. Пусть это будет пример из формулировки задачи. Время поступления деталей в промежуточный контейнер зафиксировано в массиве $x = (5, 1, 1, 2, 2, 3)$. Первый элемент — количество деталей. Значение $l = 0$, $r = 20$, тогда $q = 10$. За 10 единиц времени обрабатываются все детали. Массив y из первоначального состояния $(10, 10, 10)$ преобразован в $(1, 8, 10)$. Уменьшаем значение $r = 10$, $q = 5$. Дальнейший ход обработки (преобразование массива y) приведен в табл. 8.2.

Таблица 8.2

l	r	$y[1]$	$y[2]$	$y[3]$	Номер детали	Примечание
0	10	5	5	5		
		5	4	5	5	5-я деталь обрабатывается вторым станком
		2	4	5	4	4-я деталь обрабатывается первым станком
		2	3	5	3	3-я деталь обрабатывается вторым станком
		2	2	5	2	2-я деталь обрабатывается вторым станком
		2	1	5	1	1-я деталь обрабатывается вторым станком
0	5	2	2	2	5	ok = False, 5-я деталь не может быть обработана
2	5	3	3	3	5	ok = False, 5-я деталь не может быть обработана
3	5	4	4	4		
		4	3	4	5	5-я деталь обрабатывается вторым станком
		4	2	4	4	4-я деталь обрабатывается вторым станком
					3	ok = False, 3-я деталь не может быть обработана
4	5					По логике бинарного поиска обработка завершена

Методические указания

Тема поиска, в частности, метод бинарного поиска, является фундаментальной в информатике, поэтому содержательные задачи (а данная задача является именно таковой) необходимы для ее изучения.

Методическая ценность задачи повышается благодаря идее, заложенной в ее решении, — время обработки ищется не по распределению деталей по станкам, а наоборот, задается время и определяется, можно ли выполнить обработку деталей.

Самостоятельные исследования

1. Дайте логическое обоснование того, что алгоритм действительно определяет минимальное время.
2. Приводит ли к изменению алгоритма нахождения минимального времени обработки введение в производственную линию станков типа C , D и так далее?

«Межшкольная сеть»

Условие задачи

Некоторые школы связаны компьютерной сетью. Между школами заключены соглашения: каждая школа имеет список школ-получателей, которым она рассылает программное обеспечение всякий раз, получив новое бесплатное программное обеспечение (извне сети или из другой школы). При этом, если школа B есть в списке получателей школы A , то школа A может и не быть в списке получателей школы B .

Требуется написать программу, определяющую минимальное количество школ, которым надо передать по экземпляру нового программного обеспечения, чтобы распространить его по всем школам сети в соответствии с соглашениями (подзадача A).

Кроме того, надо обеспечить возможность рассылки нового программного обеспечения из любой школы по всем остальным школам. Для этого можно расширять списки получателей некоторых школ, добавляя в них новые школы. Требуется найти минимальное суммарное количество расширений списков, при которых программное обеспечение из любой школы достигло бы всех остальных школ (подзадача B). Одно расширение означает добавление одной новой школы-получателя в список получателей одной из школ.

Входные данные. Первая строка файла INPUT.TXT содержит целое число n — количество школ в сети ($2 \leq n \leq 100$). Школы нумеруются первыми n положительными целыми числами. Каждая из следующих n строк задает список получателей.

Строка $i + 1$ содержит номера получателей i -й школы. Каждый такой список заканчивается нулем. Пустой список содержит только нуль.

Выходные данные. Ваша программа должна записать две строки в файл OUTPUT.TXT. Его первая строка должна содержать одно положительное целое число — решение подзадачи А. Вторая строка должна содержать решение подзадачи В.

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
5	1
2 4 3 0	2
4 5 0	
0	
0	
1 0	

Обсуждение задачи

Для решения задачи требуется знание фундаментального математического понятия «отношение» и, соответственно, понятия «замкнутость отношения». Конспективно рассмотрим их.

Бинарным отношением R двух множеств A и B называют подмножество прямого произведения A и B . Прямое (декартово) произведение двух множеств A и B есть множество упорядоченных пар, в котором первый элемент каждой пары принадлежит A , а второй принадлежит B . Если $A = B$, то говорят, что R есть отношение на множестве A .

Отношение R обладает обычно рядом свойств. Так, оно обладает свойством *транзитивности*, если для любых элементов a, b, c , принадлежащих A , из $a R b$ и $b R c$ следует, что $a R c$. Словами это проговаривается так: если a и b находятся в отношении R и b и c находятся в отношении R , то пара a и c находится в отношении R .

Неформально говоря, *замкнутость отношения относительно некоторого свойства* означает, что многократное его применение не выводит нас за определенные границы. Задачу же нахождения по транзитивному отношению R отношения R^* , обладающего свойством *замкнутости*, называют *задачей замыкания*, а отношение R^* — *транзитивным замыканием R* .

Бинарное отношение R в памяти компьютера обычно описывается квадратной матрицей, размерность которой равна мощности (n) множества A (пусть это будет матрица $s[1..n, 1..n]$).

Элемент $s[i, j]$ равен значению True, если элементы множества a_i и a_j находятся в отношении R .

Для нахождения R^* используется алгоритм С. Уоршалла (1962 г.). Его запись в виде процедуры имеет вид:

```

Procedure Lock; {По отношению  $R$  строится  $R'$ }
  Var  $k, i, j$ :Integer; { $s$  — глобальная переменная}
  Begin
    For  $k:=1$  To  $n$  Do
      For  $i:=1$  To  $n$  Do
        For  $j:=1$  To  $n$  Do
           $s[i, j] := s[i, j]$  Or ( $s[i, k]$  And  $s[k, j]$ );
  End;

```

Пример. Пусть отношение R на множестве A из пяти элементов описывается матрицей:

$$s = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix},$$

где 1 означает True, а 0 — False. Тогда его транзитивное замыкание R^* описывается матрицей:

$$s = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Методическое отступление. Построение транзитивного замыкания отношения R с помощью алгоритма С. Уоршалла является хорошим упражнением на осознание сути задачи замыкания.

Вернемся к нашей задаче. Множество школ и сеть между ними описываются *ориентированным графом*, в котором вершина графа соответствует школе, а дуга (*ориентированное ребро*) — связи между школами. Бинарное отношение R — это ничто иное как наличие связи между школами. Оно не является симметричным (т. е. из $a R b$ не следует, что $b R a$), но является транзитивным. Если есть связь между школами a и b (сообще-

ние может быть передано из школы с номером a в школу с номером b) и между школами b и c , то сообщение может быть передано из школы a в школу c .

В терминологии теории графов это звучит так: если вершина b достижима (существует путь — последовательность дуг) из вершины a и вершина c достижима из вершины b , то вершина c достижима из вершины a .

Зададим себе вопрос: чему соответствует, что дает транзитивное замыкание отношения достижимости в ориентированном графе? На рис. 8.2 дан граф, соответствующий примеру из формулировки задачи; матрицы же для R и R^* рассмотрены выше по тексту.

Методическое отступление. Лучше, если бы школьник самостоятельно выполнил постановку задачи, требующей дальнейшего решения, — найти минимальное количество таких строк матрицы s , что единицы из них «покрывают» все столбцы матрицы. Очевидно, что это значение является ответом первой подзадачи.

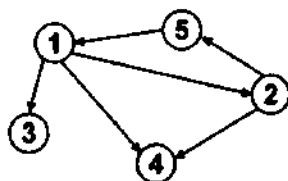


Рис. 8.2. Граф для примера из формулировки задачи

В теории графов решаемая подзадача «проговаривается» так. Пусть $G = (V, E)$ — ориентированный граф, а B — такое подмножество его вершин, что любая вершина из $V \setminus B$ достижима из какой-либо вершины, принадлежащей B . Если к тому же множество B минимально относительно включения среди подмножеств вершин с описанным свойством, то оно называется *базой ориентированного графа*^{*} G . Другими словами, после нахождения транзитивного замыкания G (а мы уже умеем его находить) требуется найти мощность базы G .

Для решения задачи нахождения множества вершин ориентированного графа, являющихся его базой, можно применить вариант «жадной» логики в виде:

Procedure FormSet(s :TGraph; **Var** a :TSet);

{В разделе описания типов программы следует задать:

Type

TGraph=Array[1..100,1..100] Of Boolean;

TPr=Array[1..100] Of Boolean;

TSet=Set Of Byte;. В переменной a мы находим множество вершин графа, составляющих его базу. Матрица s описывает отношение R^*)

* Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. Лекции по теории графов. М.: Наука, 1990. С. 293.

```

Var i,j:Byte;
      p:TPr;
Begin
  For i:=1 To n Do p[i]:=True; {Помечаем все
  вершины графа как непросмотренные}
  a:=[]; {Искомое множество пустое}
  For i:=1 To n Do
    If p[i] Then Begin {Вершина с номером i не
    представлена в базе}
      For j:=1 To n Do
        If (j In a) And s[i,j] Then a:=a+[j];
      {Исключаем из a все вершины j, смежные с i}
      For j:=1 To n Do
        If s[i,j] Then p[j]:=False; {Все вершины,
        смежные с i исключаем из дальнейшего рассмотрения}
        a:=a+[i]; {Включаем вершину i в базу}
      End;
    End;
End;

```

Для подсчета количества элементов в множестве используется простая функция:

```

Function Count(a:TSet):Integer;
Var i, r:Byte;
Begin
  r:=0;
  For i:=1 To n Do
    If i In a Then r:=r+1;
  Count:=r;
End;

```

«Сборка» решения первой подзадачи из приведенных «кирпичиков» очевидна:

```

Begin
  <ввод и инициализация данных>;
  Lock;
  FormSet(s,a);
  WriteLn(Count(a));
End.

```

Для решения второй подзадачи необходимо осознание понятия *сильной связности ориентированного графа* G . Говорят, что пара вершин v_1 и v_2 ориентированного графа G сильно связана, если существует путь (последовательность дуг) из v_1 в v_2 и

из v_2 в v_1 . Если это условие выполняется для всех пар вершин графа G , то он сильно связан.

Построив транзитивное замыкание и найдя базу графа, мы как бы исследовали связность в одну сторону. Условие второй подзадачи — «чтобы программное обеспечение из любой школы достигало бы всех остальных школ» — требует добавить в граф G минимальное количество таких дуг, чтобы G был сильно связан. При этом в нашей задаче не требуется находить сами дуги, а только их минимальное количество.

Единицы в строке с номером i матрицы s говорят о том, какие вершины графа достижимы из i (в какие школы из школы с номером i может быть передано сообщение).

Методическое отступление. Целесообразно обсудить вопрос о виде матрицы s после нахождения транзитивного замыкания для сильно связанного графа G .

Транспонируем матрицу s и зададим себе вопрос: что означает элемент транспонированной матрицы $s'[i, j]$? Для примера из формулировки задачи она имеет вид:

$$s = \begin{Bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{Bmatrix}.$$

Элемент $s'[i, j]$ говорит о том, можно ли из вершины с номером j достигнуть вершины с номером i (т. е. существует ли соответствующая последовательность дуг).

Найдем базу для графа, описываемого матрицей s' . Это требует вызова процедуры `FormSet( $s'$ ,  $b$ )`, где b — переменная множественного типа. Если окажется, что множества a и b состоят из одного элемента, а он в этом случае равен единице, то граф сильно связан, и добавление дуг не требуется. При невыполнении этого условия требуется подсчитать количество элементов в множествах a и b , тогда максимальное из этих чисел является ответом задачи. Таким образом, после транспонирования матрицы s и нахождения базы соответствующего графа логику решения задачи необходимо дополнить одним оператором:

```
If (a={1}) And (b={1}) Then WriteLn(0)
Else WriteLn(Max(Count(a), Count(b)))
```

Методические указания

Эта задача, с одной стороны, просто и жизненно сформулирована, а с другой — требует знания основ теории графов. Без этих знаний школьнику сделать для себя ряд мини-открытий в процессе ее решения достаточно сложно, но при их наличии необходимо только применить эти знания на практике. Если в школе решается проблема подготовки школьника к будущей профессиональной деятельности в области информатики, то адаптированный для этого уровня курс по алгоритмам на графах* следует вводить, и данная задача (как учебная) может быть использована в этой работе.

Самостоятельные исследования

1. «Соберите» полный текст работоспособного решения задачи из фрагментов, приведенных в ее обсуждении. Для заданного значения n (например, 8) приведите примеры графов, для которых ответом второй подзадачи будут числа от 0 до n . Количество дуг в графах должно быть минимальным. Проверьте результаты расчетов с помощью разработанной программы.
2. Определите не только минимальное количество дуг, необходимых для преобразования графа в сильно связный, но и найдите один из вариантов соединения вершин графа, решающий задачу.

ЗАДАЧИ ВТОРОГО ТУРА

«Сортировка последовательности»

Условие задачи

Сортировка является одной из наиболее часто встречающихся вычислительных задач. Рассмотрим специальную задачу сортировки, где сортируемые записи могут иметь не более трех различных значений ключа. Это бывает, например, тогда, когда мы хотим упорядочить медалистов олимпиады в соответствии с рангами медалей, то есть так, чтобы вначале шли золотые медалисты, затем серебряные и, наконец, бронзовые.

В данной задаче в качестве значений ключей используются три числа: 1, 2 и 3. Требуется отсортировать последовательность в порядке неубывания. Сортировка осуществляется с помощью *попарных перестановок* элементов. При попарной пере-

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004.

становке, определяемой номерами позиций p и q , производится обмен местами двух элементов, находящихся в позициях p и q .

Напишите программу, которая по заданной последовательности значений ключей находит минимальное количество операций попарной перестановки, необходимых для того, чтобы отсортировать заданную последовательность (подзадача А). Кроме того, постройте последовательность операций попарной перестановки, реализующую соответствующую сортировку (подзадача В).

Входные данные. Первая строка файла INPUT.TXT содержит количество ключей n ($1 \leq n \leq 1000$). Каждая из последующих n строк содержит значение ключа.

Выходные данные. Первая строка выходного файла OUTPUT.TXT должна содержать минимальное количество операций попарной перестановки l , необходимое для сортировки заданной последовательности (подзадача А). Следующие l строк должны содержать соответствующую последовательность операций попарной перестановки в порядке их выполнения (подзадача В). Каждая строка содержит одну операцию попарной перестановки, заданную двумя числами p и q — позициями двух переставляемых элементов. Позиция задается целым числом от 1 до n .

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
9	4
2	1 3
2	4 7
1	9 2
3	5 9
3	
3	
2	
3	
1	

Обсуждение задачи

Задача, несмотря на всю простоту ее формулировки, как и многие другие задачи по сортировке, обладает определенной привлекательностью из-за ее «коварства». Если снять условие минимальности количества операций попарной перестановки, то она решается за линейное время $O(n)$ с помощью следующих

простых процедур. (Оператор, полностью выделенный жирным подчеркнутым шрифтом, можно убрать, — логика по-прежнему останется работоспособной, но не дающей минимального количества операций Swap «во всех случаях жизни».)

Procedure Swap(*i, j:Integer*);

Var *q:Integer*;

Begin

r:=r+1; {Количество перестановок}

s[r,1]:=i; *s[r,2]:=j*; {Запоминаем перестановку}

q:=x[i]; *x[i]:=x[j]*; *x[j]:=q*; {Выполняем

перестановку}

End;

Procedure Solve;

Var *i, j, k:Integer*;

Begin

i:=0; *k:=0*; *j:=n*; *r:=0*;

While *k<>j* **Do**

If *x[k+1]=2* **Then** *k:=k+1*

Else

If *x[j]=3* **Then** *j:=j-1*

Else

If *x[k+1]>2* **Then** **Begin** Swap(*k+1, j*); *j:=j-1*; **End**

Else **Begin** Swap(*i+1, k+1*); *i:=i+1*; *k:=k+1*; **End**;

End;

Методическое отступление. Следует найти и охарактеризовать те наборы входных данных, при которых логика выдает не минимальное количество операций Swap.

Снимем ограничение на линейность времени сортировки (оно и не требуется по условию задачи). В этом случае ее можно решить следующим образом. В массиве *w* подсчитаем конечные позиции — 1, 2 и 3 в отсортированном массиве. Фрагмент реализации:

For *i:=0* **To** 3 **Do** *w[i]:=0*; {Первоначальное присвоение}

For *i:=1* **To** *n* **Do** *w[x[i]]:=w[x[i]]+1*; {Подсчет количества 1, 2 и 3}

For *i:=1* **To** 3 **Do** *w[i]:=w[i-1]+w[i]*;
{Формирование конечных позиций}

После этого, зная номер элемента *i* в массиве *x*, мы с помощью простой функции Tplace можем определить, какое значение — 1, 2 или 3 — должно быть на этом месте:

```

Function Tplace(i:Integer):Integer;
Var q:Integer;
Begin
  q:=1;
  While i>w[q] Do q:=q+1;
  Tplace:=q;
End;

```

Логика сортировки сводится к просмотру элементов массива x и поиску первого элемента, находящегося «не на своем месте» (значение i). После того, как он найден, в «остатке» массива x находится второй «кандидат» для операции Swap. При этом мы можем использовать информацию из массива w о том, с какого места в результате должны быть записаны числа $x[i]$. Временная сложность логики — $O(n^2)$.

```

Procedure Solve;
Var i,j,k:Integer;
Begin
  <Формирование массива w>;
  i:=1;
  While i<=n Do Begin
    k:=Tplace(i); {Определяем число, которое
    должно быть записано на месте i}
    If k<>x[i] Then Begin {Совпадения нет,
    требуется операция Swap. Один "кандидат" для
    операции есть, он находится на месте i}
      j:=w[x[i]-1]+1; {Ищем второго "кандидата"
    среди тех, на место которых должен быть записан
    элемент x[i]. Этот "кандидат" записывается
    на свое окончательное место в массиве x}
      While (j<=w[x[i]]) And (x[j]<>k) Do j:=j+1;
      If j<=w[x[i]] Then Swap(i,j) {Нашли "кандидата",
    оба операнда Swap "идут" на свои места}
      Else Begin
        j:=i+1; {"Кандидата" нет. Находим первый
    элемент со следующей позиции i+1, который можно
    записать на место i и он находится не на своем
    месте (а он гарантированно есть!)}
        While (Tplace(j)=x[j]) Or (x[j]<>k) Do
          j:=j+1; {Элемент находится на своем месте.
    Бессмысленно выполнять операцию Swap для случая,
    когда x[i]=x[j]}
        Swap(i,j);
      End;
    End;
  End;

```

```

End;
i:=i+1;
End;
End;

```

Однако вернемся к первоначальному желанию — выполнить сортировку за линейное время и за минимальное количество операций Swap. Используя идею создания массива w , сделать набросок логики несложно:

```

Procedure Solve;
  Var i,j,k:Integer;
Begin
  <Формирование массива w>;
  i:=0;k:=w[1];j:=w[2];r:=0;
  While (i<w[1]) Or (k<w[2]) Or (j<w[3]) Do
    If x[i+1]=1 Then i:=i+1
      Else If x[k+1]=2 Then k:=k+1
        Else If x[j+1]=3 Then j:=j+1
          Else <"Черный ящик" А — все
три элемента находятся "не своем" месте>;
End;

```

Но затем требуется пунктуальность в анализе оставшихся случаев. Представление в табличном виде (табл. 8.3), позволяет избежать возможных неточностей.

Таблица 8.3

Номер варианта	$x[i+1]$	$x[k+1]$	$x[j+1]$
1	2	1	1
2	2	1	2
3	2	3	1
4	2	3	2
5	3	1	1
6	3	1	2
7	3	3	1
8	3	3	2

После этой работы остается только реализация «черного ящика» А:

```

If (x[i+1]=2) Then {Варианты 1, 2, 3, 4}
  If x[k+1]=1 Then Begin {Варианты 1, 2}
    Swap(i+1,k+1);
    k:=k+1;i:=i+1;
  End
  Else If x[j+1]=1 Then Begin {Вариант 3}
    Swap(i+1,j+1);
    Swap(k+1,j+1);
    i:=i+1;j:=j+1; k:=k+1;
  End
  Else Begin {Вариант 4}
    Swap(k+1,j+1);
    k:=k+1;j:=j+1;
  End
  Else If x[j+1]=1 Then Begin {Варианты 5, 7}
    Swap(i+1,j+1);
    i:=i+1;j:=j+1;
  End
  Else If x[k+1]=3 Then Begin {Вариант 8}
    Swap(k+1,j+1);
    k:=k+1;j:=j+1;
  End
  Else Begin {Вариант 6}
    Swap(i+1,k+1);
    Swap(k+1,j+1);
    j:=j+1;i:=i+1; k:=k+1;
  End;

```

Методическое отступление. «Обилие» операторов **If ... Then ... Else** несколько «утомляет». Предложить учащимся высказать свои идеи по оптимизации кода «черного ящика» А будет вполне уместно.

Методические указания

Это задача по теме «Алгоритмы сортировки». Если исключить из объяснений таблицу 8.3, то вероятность совершения школьником ошибки при программной реализации достаточно высока. Поиск при этом тех исходных данных, для которых его логика не будет работать, позволяет в очередной раз доказать необходимость тщательного анализа задачи.

Самостоятельные исследования

Решите задачу для значений ключей 1, 2, 3 и 4. Количество операций Swap должно быть минимально, а время работы должно линейно зависеть от значения n .

«Самый длинный префикс»

Условие задачи

Структура некоторых биологических объектов представляется последовательностью их составляющих. Эти составляющие обозначаются заглавными буквами. Биологи интересуются разложением длинной последовательности в более короткие последовательности. Эти короткие последовательности называются *примитивами*. Говорят, что последовательность S может быть образована из данного множества примитивов P , если существуют n примитивов $p_1, \dots, p_n$ в P , таких, что их конкатенация (сцепление) $p_1 \dots p_n$ равняется S . При конкатенации примитивы $p_1, \dots, p_n$ записываются последовательно без разделительных пробелов. Некоторые примитивы могут встречаться в конкатенации более одного раза, и не обязательно все примитивы должны быть использованы.

Например, последовательность **ABABACABAAB** может быть образована из множества примитивов $\{A, AB, BA, CA, BBC\}$.

Первые K символов строки S будем называть *префиксом* строки S длины K . Напишите программу, которая для заданных множества примитивов P и последовательности T определяет длину максимального префикса последовательности T , который может быть образован из множества примитивов P .

Входные данные. Исходные данные расположены в двух файлах. Файл INPUT.TXT описывает множество примитивов P , а файл DATA.TXT содержит последовательность T , требующую проверки. Первая строка INPUT.TXT содержит число n — количество примитивов в множестве P ($1 \leq n \leq 100$). Каждый примитив задан в двух последовательных строках. Первая из них содержит длину L примитива ($1 \leq L \leq 20$), а вторая — строку из заглавных букв (от «A» до «Z») длины L . Все n примитивов различны. Каждая строка файла DATA.TXT содержит одну заглавную букву в первой позиции. Файл DATA.TXT заканчивается строкой с символом «.» (точка) в первой позиции. Длина последовательности не меньше 1 и не больше 500 000.

Выходные данные. Записать в первую строку файла OUTPUT.TXT длину максимального префикса последовательности T , который может быть образован из множества P .

Пример входа и выхода.

INPUT.TXT	DATA.TXT	OUTPUT.TXT
5	A	11
1	B	
A	A	
2	B	
AB	A	
3	C	
BBC	A	
2	B	
CA	A	
2	A	
BA	B	
	C	
	B	
	.	

Идея решения

Ограничения на длину примитивов и их количество позволяют ввести двумерные массивы p и q типа $TStr = \text{Array}[1..NMax, 0..LMax]$ Of Boolean ($NMax = 100$, $LMax = 20$ — описываются в разделе констант) для фиксации «активных» (задействованных в сравнении) примитивов как при обработке предыдущего символа последовательности p , так и текущего — q .

В массиве h мы фиксируем возможность образования из множества примитивов префикса каждой длины. Так, если $h[k] = \text{True}$, то существует возможность получения префикса длины k . При удачном сравнении с последним символом примитива i с k -м символом последовательности достаточно проанализировать элемент $h[k - (\text{длина примитива } i) + 1]$. Если он равен True, то может быть получен префикс и длины k .

Таким образом, задача решается «в лоб». При чтении каждого символа последовательности просматриваются все примитивы. Если есть совпадение активного примитива, и его символ — не последний, то этот факт фиксируется в массиве q (примитив остается «активным»). При положительном сравнении с его первым символом примитив переводится в разряд «активных». При сравнении с последним символом примитива анализируется возможность получения префикса с длиной, равной номеру текущего символа исходной последовательности.

«Магические квадратики»

Условие задачи

После невероятного успеха головоломки «Кубик Рубика» господин Рубик придумал ее плоский вариант — «Магические квадратики». Головоломка состоит из 8 связанных между собой одинаковых квадратиков. Ее начальная конфигурация имеет вид:

1	2	3	4
8	7	6	5

Эти квадратики раскрашены в 8 различных цветов. Цвета определяются целыми числами от 1 до 8. Конфигурация головоломки определяется последовательностью цветов квадратиков, заданных по часовой стрелке, начиная с левого верхнего квадрата и кончая левым нижним. Например, конфигурация, приведенная выше, определяется последовательностью (1, 2, 3, 4, 5, 6, 7, 8). Эта конфигурация всегда является начальной.

Существуют три базовых преобразования головоломки. Эти преобразования обозначаются буквами *A*, *B* и *C*:

- *A* — перестановка верхнего и нижнего рядов;
- *B* — циклический сдвиг вправо на один квадрат;
- *C* — поворот по часовой стрелке четырех средних квадрати-ков на 90 градусов.

Известно, что с использованием этих преобразований можно получить любую заданную конечную конфигурацию.

Реализации базовых преобразований для начальной конфигурации приведены на рис. 8.3. Числа, расположенные вне квадрати-ков, определяют позиции квадрати-ков. Если квадратик в позиции *p* содержит число *i*, это означает, что в результате соответствующего преобразования квадратик, который в начальной конфигурации находился в позиции *i*, перемещается в позицию *p*.

Напишите программу, которая находит последовательность действий, преобразующую начальную конфигурацию в заданную конечную конфигурацию (подзадача *A*).

Два дополнительных балла за каждый тест будут присуждаться в том случае, если длина последовательности преобразований не превышает 300 (подзадача *B*).

A

1	2	3	4
8	7	6	5
1	2	3	4

8 7 6 5

B

1	2	3	4
4	1	2	3
5	8	7	6

8 7 6 5

C

1	2	3	4
1	7	2	4
8	6	3	5

8 7 6 5

Рис. 8.3. Базовые преобразования

Входные данные. Файл исходных данных INPUT.TXT содержит 8 положительных целых чисел, находящихся в первой строке файла. Эти числа задают конечную конфигурацию головоломки.

Выходные данные. В первой строке файла OUTPUT.TXT должно содержаться число L — длина последовательности преобразований. Следующие L строк должны содержать найденную последовательность преобразований. В каждой строке в первой позиции должна содержаться одна из букв A, B или C, задающая соответствующее преобразование.

Дополнительные средства. В ваше распоряжение предоставляется программа MTOOL.EXE (в директории задачи), которая позволяет смоделировать работу головоломки. Программа MTOOL.EXE имеет следующий формат вызова: `mtool input.txt output.txt`, где форматы файлов `input.txt` и `output.txt` соответствуют описанным требованиям. С помощью этой программы вы сможете экспериментировать с конечной конфигурацией и с последовательностью преобразований.

Пример ввода и вывода.

INPUT.TXT	OUTPUT.TXT
2 6 8 4 5 7 3 1	7
	B
	C
	A
	B
	C
	C
	B

Обсуждение задачи

Состояние «магического квадрата» — это перестановка из восьми чисел. Количество различных состояний равно 40320, или $8!$ (факториал числа 8). Преобразования A, B и C задают

правила перехода от текущей перестановки к следующим, т. е. определяют правила перехода в пространстве состояний задачи.

Идейной основой решения является *поиск в ширину*. Он обеспечивает наименьшее количество преобразований при достижении конечного состояния (а оно задано!) из начального. При этом мы предполагаем, что оперативной памяти достаточно для хранения описания пространства состояний. Если же это не так, то следует использовать указатели и память для динамических структур данных.

Методическое отступление. Для реализации задачи необходимо уметь выполнять преобразования: по перестановке получать ее номер в лексикографическом порядке и из номера формировать перестановку (обратное преобразование). Этот материал является классикой комбинаторных алгоритмов*, так же как и работа с очередью (структурой данных) при «поиске в ширину»**.

Пусть у нас есть функция `GetNum` — вычисления по перестановке ее номера и процедура `GetPerm` — определения по номеру перестановки ее значения в некотором массиве s . Рассмотрим пример из формулировки задачи. Есть начальная перестановка (1 2 3 4 5 6 7 8) с номером 1. С помощью преобразования A переходим к перестановке (8 7 6 5 4 3 2 1) с номером 40320; с помощью преобразования B — (4 1 2 3 6 7 8 5) с номером 15130; с помощью преобразования C — (1 7 2 4 5 3 6 8) с номером 3631. Номера перестановок мы будем хранить в очереди, а пространство состояний задачи опишем с помощью двух массивов d и p . На рис. 8.4 представлено его описание после обработки переходов из начального состояния. При этом в массиве d хранится количество преобразований, а в массиве p (в первой его части) — состояние, из которого мы пришли в текущее, и (во второй части) тип преобразования. Тогда элемент массива p можно описать так:

	d		p
1	0		
	...		...
3631	1		A
	...		...
15130	1		B
	...		...
40320	1		C

Рис. 8.4. Пространство состояний задачи после обработки из начального состояния

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатории знаний, 2004. С. 31–39.

** Там же. С. 143–144.

```
prEl=Record  
  num:Integer;  
  vect:Char  
End;
```

В этом случае p есть `Array[1..NMax] Of prEl`, где $NMax$ равно 40320, а вывод последовательности преобразований при сформированном массиве p реализуется с помощью простой процедуры:

```
Procedure PrintWay(State:Integer);  
Begin  
  If State<>1 Then Begin  
    PrintWay(p[State].num);  
    WriteLn(p[State].vect)  
  End  
End;
```

Первый вызов — `PrintWay(BState)`, где $BState = 8586$, т. е. номеру, соответствующему конечной перестановке (2 6 8 4 5 7 3 1). Количество преобразований по достижении результата равно $d[BState]$.

Логика добавления элемента в массивы d и p имеет вид:

```
Procedure AddEl(s:TPerm;w:Char); {a — номер  
текущего состояния; s — перестановка, в которую  
мы переходим из состояния a с помощью  
преобразования w. TPerm — тип массива,  
используемого для хранения перестановки}  
Var c:Integer;  
Begin  
  c:=GetNum(s); {Получаем номер перестановки}  
  If d[c]=-1 Then Begin {В состояние c пришли  
первый раз}  
    d[c]:=d[a]+1; {Запись в массивы d и p}  
    With p[c] Do Begin  
      num:=a;  
      vect:=w  
    End;  
    <Добавляем состояние c в очередь как  
подлежащее обработке>;  
  End  
End;
```

ЗАДАЧИ ПЕРВОГО ТУРА

«Марсоход»

Условие задачи

В будущей экспедиции на поверхность Марса приземлится спускаемый аппарат. В аппарате находятся марсоходы, которые будут двигаться к возвращаемому модулю, приземлившись недалеко. На пути к нему марсоходы собирают образцы горных пород. Образец может быть подобран только один раз тем марсоходом, который первым посетил этот участок местности. После этого образец не может быть взят снова с этого участка, но другие марсоходы могут проходить по этому участку. Марсоходы не могут двигаться по неровным участкам местности.

Марсоход устроен так, что он может двигаться только на юг или на восток по пути от спускаемого аппарата к возвращаемому модулю. На одном участке одновременно может находиться несколько марсоходов. На рис. 9.1 схематично показано расположение горных пород и препятствий на поверхности Марса.

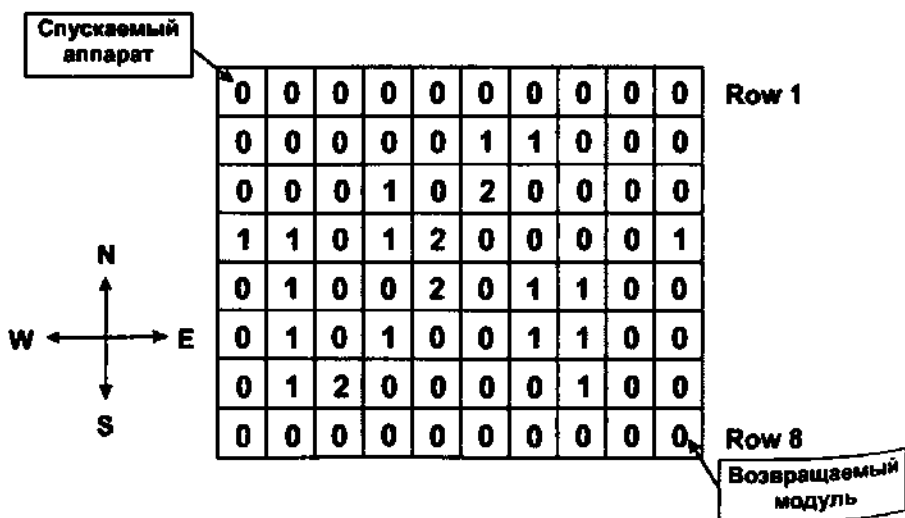


Рис. 9.1. Пример расположения горных пород и препятствий на поверхности Марса

Внимание! Если марсоход не сможет достигнуть возвращаемого модуля, все собранные им образцы теряются навсегда.

Задача. Определить индивидуальные траектории движения марсоходов, чтобы максимизировать свои баллы, зависящие от количества собранных и доставленных образцов и количества марсоходов, достигших возвращаемого модуля.

Входные данные. Поверхность планеты между местом посадки спускаемого аппарата и местонахождением возвращаемого модуля представляется в виде прямоугольной сетки размера p на q . Место посадки находится в позиции $(1, 1)$, а место возвращаемого модуля — в позиции (p, q) . Тип участка местности определяется следующим образом:

- ровный участок — 0;
- неровный участок — 1;
- участок с образцом — 2.

Структура входного файла:

Количество Марсоходов

p

q

$(x_1y_1) (x_2y_1) (x_3y_1) \dots (x_{p-1}y_1) (x_py_1)$

$(x_1y_2) (x_2y_2) (x_3y_2) \dots (x_{p-1}y_2) (x_py_2)$

$(x_1y_3) (x_2y_3) (x_3y_3) \dots (x_{p-1}y_3) (x_py_3)$

...

$(x_1y_{q-1}) (x_2y_{q-1}) (x_3y_{q-1}) \dots (x_{p-1}y_{q-1}) (x_py_{q-1})$

$(x_1y_q) (x_2y_q) (x_3y_q) \dots (x_{p-1}y_q) (x_py_q)$,

где p и q — размеры сетки. Количество Марсоходов — целые числа, меньшие 1000. Каждая из q строк дает описание участка. p и q не превосходят 128.

Пример входных данных.

MARS.DAT	Объяснение
2	Количество марсоходов
10	Размер p
8	Размер q
0 0 0 0 0 0 0 0 0 0	Строка 1
0 0 0 0 0 1 1 0 0 0	Строка 2
0 0 0 1 0 2 0 0 0 0	Строка 3
1 1 0 1 2 0 0 0 0 1	Строка 4
0 1 0 0 2 0 1 1 0 0	Строка 5
0 1 0 1 0 0 1 1 0 0	Строка 6
0 1 2 0 0 0 0 1 0 0	Строка 7
0 0 0 0 0 0 0 0 0 0	Строка 8

Выходные данные. Выходной файл представляется в виде последовательности строк, определяющих движение марсоходов к местоположению возвращаемого модуля. Каждая строка содержит номер марсохода и числа 0 или 1, где 0 означает движение на юг, а 1 — на восток.

Пример выходных данных.

MARS.OUT	Объяснение
1 1	Марсоход 1 идет на восток
1 0	Марсоход 1 идет на юг
2 1	Марсоход 2 идет на восток
2 0	Марсоход 2 идет на юг
1 1	и так далее...
1 1	
2 0	
2 1	
2 0	
2 0	
2 0	
2 0	
1 1	
1 0	
1 0	
1 0	
1 0	
1 0	
1 0	
2 0	
2 1	
1 1	
1 1	
1 1	
1 1	
1 1	
2 1	
2 1	
2 1	
2 1	
2 1	
2 1	
2 1	

Два марсохода с тремя образцами достигают возвращаемого модуля и получают 5 баллов из 5 возможных. Итоговая оценка — 100%.

Баллы. Баллы подсчитываются на основе количества собранных и доставленных образцов с учетом количества марсоходов, достигнувших или не достигнувших возвращаемого модуля. За решение с неправильным ходом дается 0 баллов. Неправильным считается ход, когда марсоход движется по неровному участку или когда выходит за пределы сетки. Баллы = (количество собранных и доставленных до модуля образцов + количество достигнувших модуля марсоходов / количество не достигнувших модуля марсоходов) как % максимально возможного балла решения. Можно получить максимальную оценку 100% и минимальную — 0%.

Обсуждение задачи

Очевидно, что «жадный» алгоритм — когда мы выбираем путь в лабиринте для первого марсохода с максимальным количеством образцов, затем следующий путь — для второго марсохода (по этому же критерию) и т. д. — дает приближенный результат. Найти контрпример уже для двух-трех марсоходов не составляет труда.

Размерность задачи подсказывает, что реализация полного перебора — поиска всего множества путей и выбора из них подмножества с мощностью, равной количеству марсоходов, и максимальным суммарным весом — бесперспективный путь.

Динамические схемы нарушаются условием о том, что марсоходы могут проходить по одной клетке.

Что же нам остается? Это первая задача на международных олимпиадах и, пожалуй, единственная (до 2006 года включительно), в которой требуется знание алгоритмов *поиска максимального потока в сетях*. Отсюда следует, что обсуждению этой задачи должно предшествовать изучение данной темы. Приведем для полноты изложения материал (в сокращенном виде) по данной теме, взятый из книги одного из авторов*.

Постановка задачи и основные понятия. Дан ориентированный граф (сеть) $G = (V, E)$, вершина s графа (источник) и вершина t (сток). Каждой дуге (i, j) приписана некоторая *пропускная способность* $c(i, j) \geq 0$ (без потери общности считаем ее целочисленной величиной), определяющая максимальное значение потока, который может протекать по данной дуге. *Поток в сети* называют целочисленную функцию $f(i, j)$, заданную на множестве дуг E и обладающую следующими свойствами:

* Окулов С. М. Дискретная математика: теория и практика решения задач по информатике. М.: БИНОМ. Лаборатория знаний, 2007.

Теорема*. Следующие три условия эквивалентны:

- поток из s в t максимален,
- не существует увеличивающей цепи для f ,
- $W(f) = c(A, V \setminus A)$ для некоторого $A \subseteq V$, такого, что $s \in A, t \notin A$.

Обозначим максимальный поток в сети как F . Его начальное значение — нулевое, а структурой данных для описания F является матрица того же типа, что и матрица C , в которой определены пропускные способности дуг.

В алгоритме Л. Форда–Д. Фалкерсона «механизм» построения увеличивающей цепи строго не оговаривается. В частности, может использоваться поиск в глубину. Алгоритм является псевдополиномиальным и имеет оценку $O(nmU)$, где $U = \max(c(i, j))$.

В 1969 г. К. Эдмондс и Р. Карп предложили в качестве увеличивающей цепи выбирать цепь наименьшей длины (без учета пропускной способности дуг этой цепи), при этом все дуги имеют единичную длину. На практике этот выбор реализуется с помощью поиска в ширину. Алгоритм имеет полиномиальную оценку $O(nm^2)$. Рассмотрим его.

Пример. На рис. 9.2 показана сеть, источник — вершина 1, сток — вершина 6, в круглых скобках для дуг указаны их пропускные способности. Минимальный разрез — дуги (1, 2) и (3, 4); следовательно, согласно теореме, максимальный поток равен 4. Однако нахождение распределения максимального потока по дугам — по-прежнему вопрос открытый.

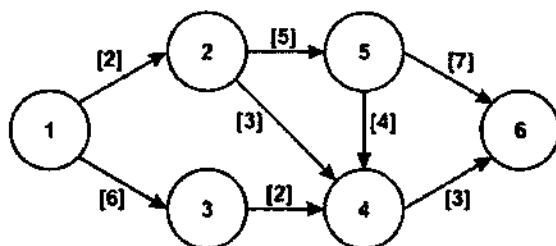


Рис. 9.2. Пример сети для иллюстрации «техники меток» при построении максимального потока

Первая итерация — рис. 9.3. Присвоим вершине 1 метку $[1, \infty]$.

Первый шаг. Рассмотрим дуги, началом которых является вершина 1 — дуги (1, 2) и (1, 3). Вершины 2 и 3 не помечены, поэтому присваиваем им метки, для 2-й — $[1, 2]$ и 3-й — $[1, 6]$.

* Липский В. Комбинаторика для программистов. М.: Мир, 1988.

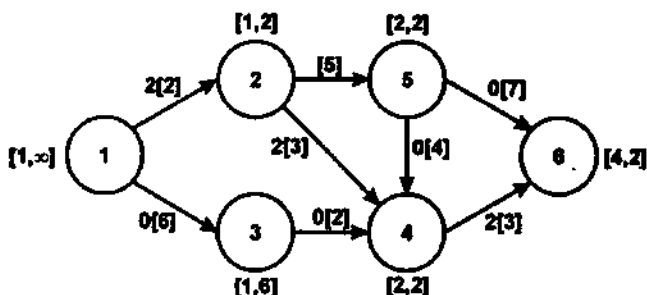


Рис. 9.3. Первая итерация метода построения максимального потока

Что представляют собой эти метки? Первая цифра — номер вершины, из которой идет поток, вторая цифра — численное значение потока, который можно передать по этой дуге.

Второй шаг. Выберем помеченную, но еще не просмотренную вершину. Первой в соответствующей структуре данных записана вершина 2. Рассмотрим дуги, для которых она является началом, — это дуги (2,4) и (2,5). Вершины 4 и 5 не помечены. Присвоим им метки — [2,2] и [2,2]. Итак, на втором шаге вершина 2 просмотрена, вершины 3, 4, 5 помечены, но не просмотрены, а остальные вершины не помечены.

Третий шаг. Выбираем вершину 3. Рассмотрим дугу (3,4). Вершина 4 помечена. Переходим к следующей вершине — четвертой; соответствующая дуга — (4,6). Вершина 6 не помечена. Присваиваем ей метку [4,2]. Мы достигли вершины-стока, тем самым найдя путь (последовательность дуг), поток по которому можно увеличить. Информация об этом пути содержит метки вершин. В данном случае путь, или увеличивающаяся цепочка, имеет вид: $1 \rightarrow 2 \rightarrow 4 \rightarrow 6$. Максимально возможный поток, который можно передать по дугам этого пути, определяется второй цифрой метки вершины стока, т. е. 2. Поток в сети стал равным 2. Заметим, что это построение цепочки — ни что иное как модифицированный просмотр в ширину вершин графа.

Вторая итерация — рис. 9.4.

Первый шаг. Присвоим вершине 1 метку [1,∞]. Рассмотрим дуги, началом которых является помеченная вершина 1. Это дуги (1,2) и (1,3). Вершина 2 не может быть помечена, так как пропускная способность дуги (1,2) исчерпана. Вершине 3 присваиваем метку [1,6].

Второй шаг. Выберем помеченную, но не просмотренную вершину. Это вершина 3. Повторяем действия. В результате вершина 4 получает метку [3,2].

Третий шаг. Выбираем вершину 4: только она помечена и не просмотрена. Вершине 6 присваиваем метку [4,1]. Почему толь-

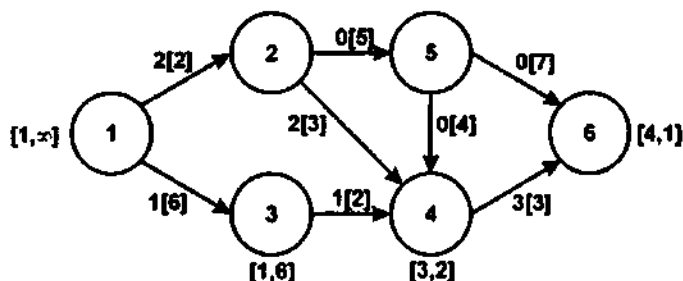


Рис. 9.4. Вторая итерация метода построения максимального потока

ко одна единица потока? Потому что на предыдущей итерации израсходованы две единицы пропускной способности данной дуги, осталась только одна. Вершина-сток достигнута. Найдена увеличивающая поток цепочка, это $1 \rightarrow 3 \rightarrow 4 \rightarrow 6$, по которой можно «протащить» единичный поток. Результирующий поток в сети равен 3.

Третья итерация — рис. 9.5. Вершине 1 присваиваем метку $[1, \infty]$.

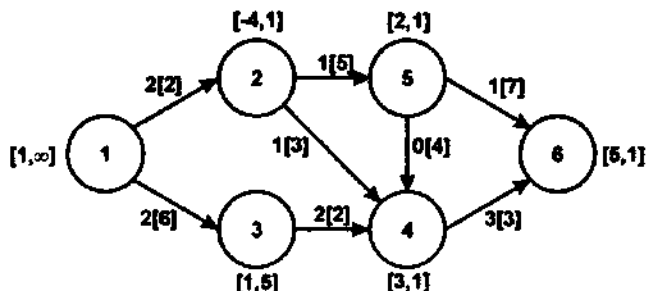


Рис. 9.5. Третья итерация метода построения максимального потока

Первый шаг. Результат — метка $[1, 5]$ у вершины 3.

Второй шаг — метка $[3, 1]$ у вершины 4.

Третий шаг. Пропускная способность дуги $(4, 6)$ израсходована полностью. Однако есть *обратная* дуга $(2, 4)$, по которой передается *поток, не равный нулю* (обратите внимание на текст, выделенный курсивом, — это «изюминка» метода!). Попробуем перераспределить поток. Нам необходимо передать из вершины 4 поток, равный единице (зафиксирован в метке вершины). Задержим единицу потока в вершине 2, т. е. вернем единицу потока из вершины 4 в вершину 2. Эту особенность зафиксируем в метке вершины 2 знаком «минус» (технический прием, упрощающий реализацию) — $[-4, 1]$. Тогда единицу потока из вершины 4 мы передадим по сети вместо той, которая задержана в

вершине 2, а единицу потока из вершины 2 попытаемся «протолкнуть» по сети, используя другие дуги. Итак, вершина 4 просмотрена, вершина 2 помечена, вершины 5 и 6 не помечены.

Четвертый и пятый шаги очевидны. Передаем единицу потока из вершины 2 в вершину 6 через вершину 5. Вершина-сток достигнута, найдена цепочка $1 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 5 \rightarrow 6$, по которой можно передать поток, равный единице. При этом по прямым дугам поток увеличивается на единицу, по обратным — уменьшается. Суммарный поток в сети — 4 единицы.

Четвертая итерация — рис. 9.6. Вершине 1 присваиваем метку $[1, \infty]$.

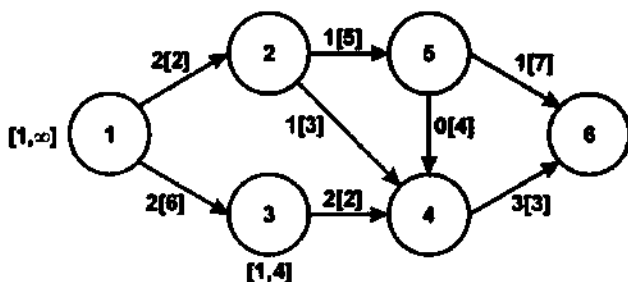


Рис. 9.6. Четвертая итерация метода построения максимального потока

Первый шаг. Помечаем вершину 3 — $[1, 4]$.

Второй шаг. Рассматриваем помеченную, но не просмотренную вершину 3. Одна дуга — $(3, 4)$. Вершину 4 мы пометить не можем — пропускная способность дуги исчерпана. Помеченных вершин больше нет, и вершина-сток не достигнута. Увеличивающую поток цепочку мы построить не можем. Значит, нами найден максимальный поток в сети; можно заканчивать работу.

Итак, в чем заключается суть алгоритма?

1. На каждой итерации вершины сети могут находиться в одном из трех состояний: «вершине присвоена метка, и она просмотрена»; «вершине присвоена метка, и она не просмотрена», т. е. не все смежные с ней вершины обработаны; «вершина не имеет метки».

2. На каждой итерации мы выбираем помеченную, но не просмотренную вершину v и пытаемся найти вершину u , смежную с v , которую можно пометить. Помеченные вершины, достижимые из вершины-источника, образуют множество вершин сети G . Если среди этих вершин окажется вершина-сток, то это означает успешный результат поиска цепочки, увеличивающей поток; при неизменности этого множества работа заканчивается — поток изменить нельзя.

Введение в метод блокирующих потоков, или алгоритм Е. А. Диница. Основная идея данного метода — алгоритм состоит из фаз, на которых поток увеличивается сразу вдоль всех кратчайших цепей определенной длины. Для этого на i -й фазе строится вспомогательная ациклическая (бесконтурная) сеть. Эта сеть содержит все увеличивающие цепи, длина которых не превышает k_i , где k_i — длина кратчайшего пути из s в t на i -ой итерации. Величину k_i называют *длиной вспомогательной сети*.

Рассмотрим работу алгоритма на i -й итерации.

Шаг 1. Построение вспомогательной сети

С помощью поиска в ширину мы движемся из источника сети в сток по допустимым дугам. Первоначально в очередь помещается исток. Затем, пока очередь не пуста, из нее извлекается очередная вершина u . Для этой вершины просматриваются все вершины, в которые ведут допустимые дуги из u . Если текущая вершина v не была ранее просмотрена, то она помещается в очередь, а соответствующая дуга (u, v) добавляется во вспомогательную сеть S_k . Если вершина v уже просмотрена, то она не помещается в очередь. Однако дуга (u, v) добавляется во вспомогательную сеть S_k в том случае, когда длина пути от истока до текущей вершины v больше (максимум на 1) длины пути от истока до вершины u , из которой в данный момент просматривается вершина v . Длина кратчайшего пути от истока до вершины v хранится как метка, значение которой присваивается вершине при записи ее в очередь. Это значение больше на единицу значения метки вершины u . Дуга $e = (u, v)$ добавляется с пропускной способностью $c_k(e) = c(e) - f(e)$, если дуга e согласованна, и $c_k(e) = f(e)$, если дуга e несогласованна.

Пусть сток сети достигается и имеет метку k . Тогда значение k становится «фиксированной» длиной вспомогательной сети на данной итерации. Поиск в ширину продолжается до тех пор, пока очередь не пуста. Однако вершины с меткой, большей значения k , в очередь не записываются. Таким образом, вспомогательная сеть является подсетью исходной сети, содержащей все кратчайшие пути из истока в сток длины k . Если t не достигнут при поиске в ширину, то работа алгоритма завершается.

Шаг 2. Поиск блокирующего потока

Блокирующим потоком в сети называется поток, при котором любая цепь из истока в сток содержит полностью насыщенную дугу (насыщенная дуга блокирует дальнейшее увеличение величины потока по данной увеличивающей цепи). Блокирующий поток во вспомогательной сети является максимальным. В построенной вспомогательной сети длины k находится блоки-

рующий поток (каждая увеличивающая цепь имеет длину k , ибо так строилась вспомогательная сеть). Найденный поток переносится в исходную сеть и осуществляется переход на шаг 1 (следующая итерация).

Для построения блокирующего потока используется поиск в глубину. Пусть найден j -й путь из s в t . «Пустим» по этому пути поток f_j . Это значит, что как минимум одна дуга вспомогательной сети является насыщенной. Удалим все насыщенные дуги. В результате могут образоваться «тупики»: вершины, не имеющие выходных дуг (кроме стока); вершины, не имеющие входных дуг (кроме источника); изолированные вершины. Удалим из вспомогательной сети данные вершины со всеми инцидентными им дугами, что, в свою очередь, может привести к образованию новых «тупиков». Процесс удаления производится до тех пор, пока во вспомогательной сети не останется ни одного «тупика». Изменим пропускные способности оставшихся дуг по формуле $c_k(e) = c_k(e) - f_j(e)$. Поиск потоков продолжается до тех пор, пока вспомогательная сеть не окажется пустой. Найденный поток $f = \sum f_j$.

После завершения работы алгоритма исходная сеть будет содержать максимальный поток.

Алгоритм Е. А. Диница имеет временную оценку $O(n^2m)$. Итерации алгоритма начинаются с построения вспомогательной сети, строящейся обходом в ширину, до момента появления в обходе вершины-стока. Так как любой путь, не проходящий дважды через какую-либо вершину, имеет длину не более n , то именно эта величина ограничивает число вспомогательных сетей в алгоритме Е. А. Диница. Количество вершин во вспомогательной сети ограничено величиной n , а количество дуг — не более m . При нахождении увеличивающей цепи как минимум одна из дуг становится насыщенной и удаляется из вспомогательной сети. Другими словами, существует не более m путей из истока в сток. Каждый путь строится за время $O(n)$, а общее время построения потока во вспомогательной сети пропорционально $O(nm)$, что дает нам неформальную оценку времени работы алгоритма.

Пример. На рис. 9.7 показана сеть для анализа алгоритма Е. А. Диница.

Первая итерация алгоритма начинается с построения вспомогательной сети (рис. 9.8). Исток принадлежит к нулевому слою (слой показывает длину кратчайшего пути по количеству дуг от истока до данной вершины). Просматриваем все вершины, связанные с истоком непосредственно, и записываем их номера в очередь. Это вершины 2, 3 и 6 (t). Отнесем их к первому

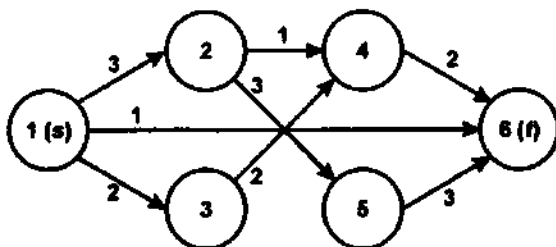


Рис. 9.7. Пример графа для анализа алгоритма

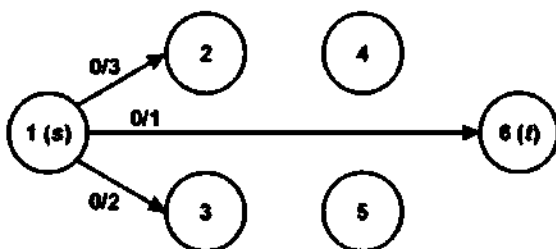


Рис. 9.8. Первая итерация алгоритма

слою. Сток оказался просмотренным ($k = 1$), длина кратчайшего пути $p: s \rightarrow t$ равна 1. Добавить дуги во вспомогательную сеть с сохранением условия $k = 1$ нет возможности. Переходим к построению блокирующего потока во вспомогательной сети.

Используя поиск в глубину, находим увеличивающие цепи (рис. 9.9). Первоначально из вершины 1 (s) просматриваем вершину 2. Так как вершина 2 является тупиковой, то ее следует удалить из вспомогательной сети вместе с инцидентными дугами. То же произойдет и с вершиной 3. Существует единственная увеличивающая цепь в данной вспомогательной сети — (1,6). После первой итерации величина потока равна 1.

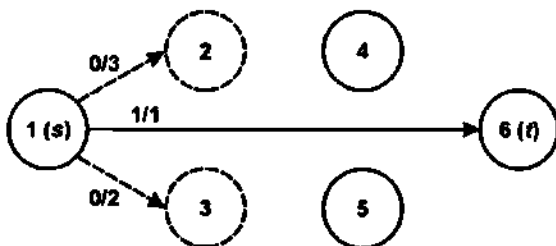


Рис. 9.9. Поиск увеличивающих цепей на первой итерации алгоритма

Вторая итерация. Вспомогательная сеть имеет четыре слоя 0...3 (рис. 9.10). Исток принадлежит нулевому слою. Из него просматриваем и записываем в очередь вершины 2 и 3. Они принадлежат первому слою. Во вспомогательную сеть добавляются соответствующие дуги (1,2) и (1,3). Из очереди извлекаем вершину 2. Просматривая ее исходящие дуги, в очередь записываем вершины 4 и 5. Они принадлежат второму слою. Дуги (2,4) и (2,5) добавляются во вспомогательную сеть. Из очереди извлекаем вершину 3. Единственная достижимая из нее вершина — 4, которая уже была просмотрена из вершины 2. Но так как вершины 3 и 4 принадлежат разным слоям, то дуга (3,4) добавляется во вспомогательную сеть. Из очереди извлекаем вершину 4. Единственная достижимая из нее вершина — 6 (сток). Сток принадлежит третьему слою. Так как сток просмотрен, в очередь никакие вершины больше не записываются. Просматриваем вершины, оставшиеся в очереди. Из вершины 5 достигим сток. Сток уже просмотрен, он он принадлежит другому слою, поэтому дуга (5,6) переносится во вспомогательную сеть. Очередь пуста, построение вспомогательной сети завершено. Любая увеличивающая цепь в ней имеет длину 3.

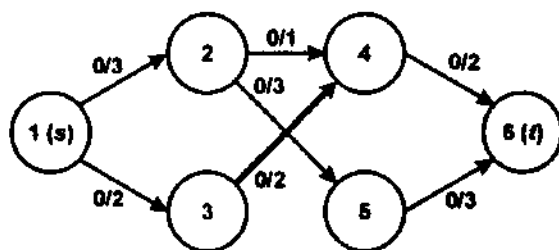


Рис. 9.10. Вспомогательная сеть на второй итерации алгоритма

В построенной вспомогательной сети находим увеличивающие цепи:

- (1,2), (2,4), (4,6), величина потока 1, насыщенная дуга (2,4);
- (1,2), (2,5), (5,6), величина потока 2, насыщенная дуга (1,2);
- (1,3), (3,4), (4,6), величина потока 1, насыщенная дуга (4,6).

На рис. 9.11 пунктиром выделены дуги, насыщенные после пропускания блокирующего потока. Величина потока после второй итерации равна 5.

На третьей итерации (рис. 9.12) вспомогательная сеть имеет 6 уровней (0...5). Существует единственная увеличивающая цепь, каждая дуга которой становится насыщенной после построения потока. Насыщение дуги (2,4) происходит в обратном

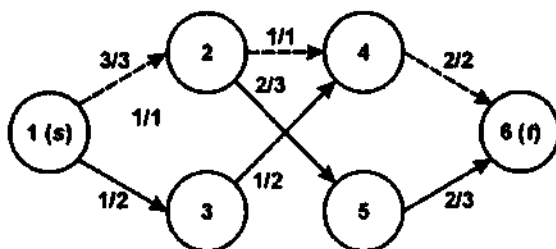


Рис. 9.11. Поток и насыщенные дуги на второй итерации алгоритма

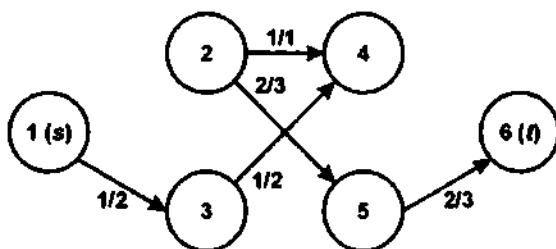


Рис. 9.12. Третья итерация алгоритма

направлении, так как она в данном случае является несогласованной.

В итоге максимальный поток в исходной сети имеет величину 6 (рис. 9.13).

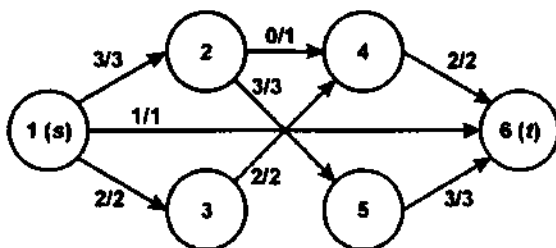


Рис. 9.13. Результирующий поток и его распределение по дугам

Модификация алгоритма Е. А. Диница. Рассмотрим один из наиболее эффективных алгоритмов построения максимального потока в сети, предложенный в 1978 году В. Малхотри, П. Кумаром и С. Махешвари. Этот алгоритм является модификацией метода блокирующего потока, предложенного Е. А. Диницем. На каждой итерации на базе остаточной сети строится вспомогательная сеть тем же способом, что и в методе Е. А. Диница. Затем во вспомогательной сети находится блокирующий поток. Найденный поток суммируется с текущим потоком, а

остаточная сеть изменяется, и выполняется следующая итерация. После каждой итерации длина кратчайшего пути в остаточной сети от источника к стоку увеличивается. Следовательно, общее количество итераций не превышает количества вершин в сети. Алгоритм В. Малхотри, П. Кумара и С. Махешвари позволяет построить блокирующий поток во вспомогательной сети за время $O(n^2)$, а не за $O(nm)$, тем самым уменьшив временную сложность алгоритма до $O(n^3)$.

Рассмотрим подробнее алгоритм построения блокирующего потока. Потенциалом $P(v)$ вершины v сети назовем максимальное количество потока, которое может быть пропущено через вершину. Очевидно, что потенциал является минимумом из суммарной пропускной способности входящих в вершину дуг $P_i(v)$ и суммарной пропускной способности исходящих дуг $P_o(v)$. Для источника и стока потенциал равен $P_i(s)$ и $P_o(t)$, соответственно. Для каждой вершины вычисляется ее потенциал. Ясно, что через вершины с нулевым потенциалом поток проходить не может. Следовательно, их можно удалить из вспомогательной сети. Удалим эти вершины и дуги, им инцидентные, обновив требуемым образом потенциалы вершин, смежных с удаленными. Если в результате появятся новые вершины с нулевым потенциалом, рекурсивно удалим и их. В результате во вспомогательной сети останутся только вершины с ненулевым потенциалом.

После этого приступим к построению блокирующего потока. Пусть вершина v принадлежит k -му слою. Протолкнем p единиц потока из вершины с минимальным потенциалом в смежные с ней вершины по исходящим дугам с ненулевой остаточной пропускной способностью. Попутно будем переносить проталкиваемый поток в исходную сеть, а также корректировать потенциалы вершин, отправляющих и принимающих избыток потока. В результате весь (ввиду минимальности потенциала вершины v) проталкиваемый поток соберется в вершинах $(k+1)$ -го слоя. Повторим процесс отправки потока из вершин $(k+1)$ -го слоя, содержащих избыток потока, в смежные им вершины $(k+2)$ -го слоя. И так до тех пор, пока весь поток не соберется в последнем слое. Заметим, что в этом слое содержится только поток, ибо все остальные вершины, ранее ему принадлежащие, были удалены из вспомогательной сети, как вершины имеющие нулевой потенциал. Следовательно, весь поток величины p , отправленный из вершины с минимальным потенциалом, полностью соберется в стоке. Далее вновь, начиная с вершины v , осуществляется подвод потока уже по входящим дугам. В результате на первом шаге недостаток потока переадресуется к узлам $(k-1)$ -го слоя, затем $(k-2)$ -го. И так до тех пор, пока весь поток величины p , отправленный из вершины с минимальным потен-

циалом, не соберется в истоке. Таким образом, поток и во вспомогательной, и в основной сети увеличится на величину p .

В процессе увеличения потока могли появиться вершины с нулевым потенциалом. Более того, одна такая вершина есть наверняка — это вершина v , ибо ее потенциал используется полностью. Удалим из сети Е. А. Диница вершины, имеющие нулевой потенциал. Если после данного шага множество вершин сети Е. А. Диница все еще не пусто, то вновь перейдем к выбору вершины с минимальным значением потенциала, иначе вернемся к построению вспомогательной сети относительно нового (увеличенного на величину p) потока в основной сети.

Рассмотрим работу алгоритма на примере (рис. 9.7). Аналогично, как и в методе Е. А. Диница, строится вспомогательная сеть.

На первой итерации алгоритма сеть состоит из 2 уровней (рис. 9.14). Во вспомогательную сеть вошли 4 вершины — 1, 2, 3, 6. Подсчитаем потенциал каждой вершины. Сумма пропускных способностей:

- исходящих дуг истока — равна 6, потенциал равен 6;
- входящих дуг вершины 2 — равна 3, исходящих — 0, потенциал равен $\min(3, 0) = 0$;
- входящих дуг вершины 3 — равна 2, исходящих — 0, потенциал равен $\min(2, 0) = 0$;
- входящих дуг стока — равна 1, его потенциал равен 5.

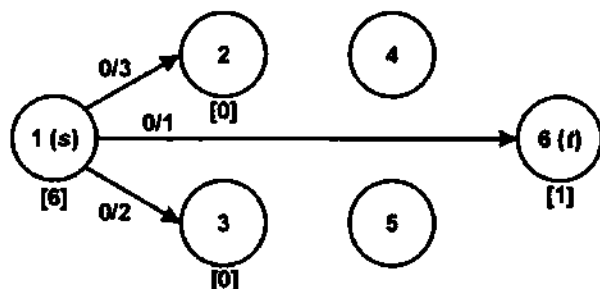


Рис. 9.14. Вспомогательная сеть на первой итерации алгоритма
В. Малхотри, П. Куммера и С. Махешвари

После удаления вершин с нулевыми потенциалами и корректировки потенциалов оставшихся вершин получаем сеть, изображенную на рис. 9.15. В сети осталось две вершины с минимальным потенциалом 1. Это означает, что по данной сети можно гарантированно пропустить поток величины 1. После корректировки потенциалов вершин и удаления вершин с нулевыми потенциалами сеть не содержит ни одной вершины. Переходим к следующей итерации.

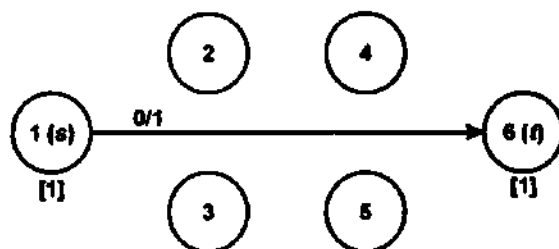


Рис. 9.15. Результирующая сеть на первой итерации алгоритма В. Малхотри, П. Куммера и С. Махешвари

На второй итерации вспомогательная сеть будет иметь 4 уровня (0...3). Любая увеличивающая цепь в ней имеет длину 3 (рис. 9.16).

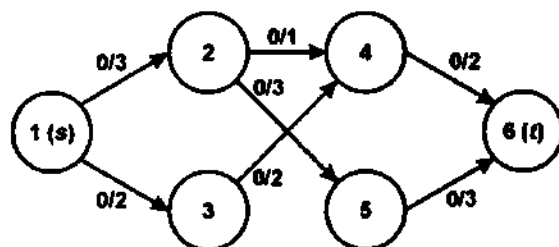


Рис. 9.16. Вспомогательная сеть на второй итерации алгоритма

Подсчитаем потенциал всех вершин этой сети (табл. 9.1).

Таблица 9.1

Вершины	Общая пропускная способность		Потенциал вершины
	входящих дуг	исходящих дуг	
1 (s)	~	5	5
2	3	4	3
3	2	2	2
4	3	2	2
5	3	3	3
6 (t)	5	~	5

Выберем вершину с минимальным потенциалом 2 (вершина с номером 3). Необходимо продвинуть поток величины 2 из вершины 3 к истоку по входящим дугам и к стоку по исходящим дугам. Это возможно, так как остальные вершины имеют потенциал не меньше 2. Поток величины 2 проходит через вер-

шину 3 по следующему пути: (1,3), (3,4), (4,6) — в общем случае это могла быть и не последовательная (с ветвлениями) увеличивающая цепь.

Заново корректируем потенциалы вершин и удаляем вершины с нулевым потенциалом. На рис. 9.17 пунктиром показаны вершины и дуги, удаленные после выполнения процедуры. В табл. 9.2 приведены новые значения потенциалов вершин.

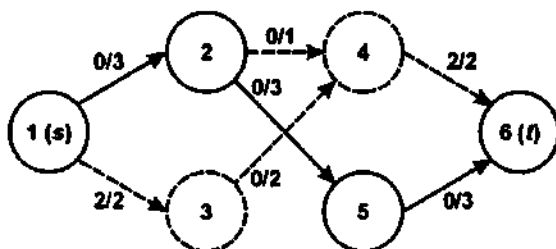


Рис. 9.17. Удаляемая часть сети выделена пунктиром

Таблица 9.2

№ вершины	Пропускная способность		Потенциал вершины
	входящих дуг	исходящих дуг	
1 (s)	~	3	3
2	3	3	3
5	3	3	3
6 (t)	3	~	3

Выберем вершину с минимальным потенциалом 3. Это будет вершина с номером 1 — исток. Теперь необходимо продвинуть поток величины 2 от истока к стоку. Увеличивающая цепь (1,2), (2,5), (5,6) величины 3 закончит построение блокирующего потока на второй итерации, так как все дуги вспомогательной сети станут насыщенными, и при корректировке потенциалов в ней не останется вершин.

При попытке выполнить третью итерацию алгоритм завершит работу, поскольку в остаточной сети исток больше не имеет исходящих дуг с положительными пропускными способностями, т. е. уже невозможно построить вспомогательную сеть, содержащую хотя бы один путь от истока к стоку. Результирующий поток совпадает с тем, что приведен на рис. 9.13.

Из рассмотренного примера мы видим, что, в отличие от алгоритма Е. А. Диница, в данном алгоритме для нахождения максимального потока использованы две итерации, при этом построено всего 3 увеличивающие цепи, а не 4.

Методическое отступление. После проведения как минимум трех — четырех занятий по теме «Потоки в сетях» появляются предпосылки для обсуждения задачи о марсоходах. Однако, к сожалению, в «чистом» виде она не «ложится» ни в один из рассмотренных алгоритмов.

Идея решения

Требуется построить поток мощности r (количество марсоходов) минимальной суммарной стоимости.

Пусть есть два марсохода и простейший вариант поверхности:

$$\begin{pmatrix} 0 & 1 \\ 2 & 2 \\ 2 & 0 \end{pmatrix}$$

Пронумеруем клетки:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

А затем построим сеть (ориентированный граф) так, как это показано на рис. 9.18. Каждой клетке здесь соответствует две вершины сети (условно назовем их «входящей» и «выходящей»): — первая указывает номер клетки, а вторая — номер на рис. 9.18 (размер местности) больше. Если это клетка с препятствием, то ей соответствуют две изолированные вершины (на рис. 9.18 — вершины с номерами 2 и 8). Если клетка не содержит образца, то соответствующие ей вершины соединяются двумя дугами — прямой и обратной (вершины с номерами 1 и 7, 6 и 12). Первая дуга имеет бесконечную пропускную способность, вторая — нулевую. Аналогично соединяются между собой выходящие и входящие вершины разных клеток, но обязательно тех, между которыми есть связь при ходе марсохода (вершины 9 и 4, 9 и 5, 11 и 6), а также вершины для клеток с образцами: вершины 3 и 9 — дугами 9 и 10, вершины 4 и 10 — дугами 15 и 16, вершины 5 и 11 — дугами 21 и 22. Кроме того, последний тип вершин соединяется двумя дополнительными дугами. Первый тип дуги имеет единичную пропускную способность и стоимость, равную -1 (дуги 11, 17, 23). Второго типа дуги имеет нулевую пропускную способность и единичную стоимость (дуги 12, 18, 24). Все остальные ранее описанные дуги имеют нулевую стоимость. Целью введения обратных дуг является реализация возможности перераспределения потока, используемой в технологии потоковых алгоритмов.

Примечание. Нумерация как вершин сети, так и дуг может быть другой. Это зависит от конкретной программной реализации.

Описание сети в памяти компьютера может быть различным. Один из возможных вариантов — через список дуг:

TEdge=Record

Vx, Vy, Flow, Cost, Capac, Rever: Integer {Vx, Vy — номера вершин; Flow — поток по дуге; Cost — стоимость; Capac — пропускная способность дуги; Rever — номер обратной дуги}

End;

TEdgeSet=Array[0..EMax] Of TEdge; {EMax — максимальное количество дуг}

В разделе переменных определим массив (E) типа TEdgeSet. Например, для сети на рис. 9.18 — это массив $E[9] = (3, 9, 0, 0, Inf, 10)$, $E[10] = (9, 3, 0, 0, 0, 9)$, $E[11] = (3, 9, 0, -1, 1, 12)$, $E[12] = (9, 3, 0, 1, 0, 11)$, где *Inf* — константа, обозначающая «бесконечность».

Для удобства организации циклов по номерам вершин сети и перехода к дугам целесообразно ввести массив множеств (G), элемент $G[i]$ которого описывает номера дуг, исходящих из вершины *i*. Для рассматриваемого примера имеем: $G[1] = [3]$, $G[2] = []$, $G[3] = [2, 9, 11]$, $G[4] = [8, 15, 17]$, $G[5] = [6, 21, 23]$, $G[6] = [14, 20, 25]$, $G[7] = [1, 4]$, $G[8] = []$, $G[9] = [5, 7, 10, 12]$, $G[10] = [13, 16, 18]$, $G[11] = [19, 22, 24]$, $G[12] = [26]$.

Методическое отступление. Реализация логики построения сети может быть вынесена на самостоятельную работу или дана в качестве примера для разбора кода:

Procedure AddEdge(X, Y, ECapac, ECost, ERever: Integer);

Begin

ENum:=ENum+1; {ENum — количество дуг}

With E[ENum] Do Begin

Vx:=X; {Начало дуги}

Vy:=Y; {Конец дуги}

Capac:=ECapac; {Пропускная способность}

Rever:=ERever; {Номер обратной дуги}

Cost:=ECost; {Стоимость}

Flow:=0; {Поток}

End;

<Добавить элемент в $G[X]$ — $G[X]:=G[X]+[ENum]>;$

End;

```
Function VertB(i,j:Integer):Integer;  
Begin  
  VertB:=(i-1)*m+j; {m - количество столбцов}  
End;  
Function VertA(i,j:Integer):Integer;  
Begin  
  VertA:=n*m+VertB(i,j); {n - количество строк}  
End;  
Procedure BuiltNet;  
  Var i,j:Integer;  
  Begin  
    <Ввод r,m,n - количество марсоходов, размеры поля>;  
    <Ввод карты местности - Map>;  
    ENum:=0;  
    For i:=1 To n Do  
      For j:=1 To m Do  
        If Map[i,j]<>Wall Then Begin {Если это не  
препятствие, то описываем клетку}  
          If i<n Then {Вертикальная связь}  
            If Map[i+1,j]<>Wall Then Begin  
              AddEdge(VertA(i,j),VertB(i+1,j),Inf,0,ENum+2);  
              AddEdge(VertB(i+1,j),VertA(i,j),0,0,ENum)  
            End;  
            If j<m Then {Горизонтальная связь}  
              If Map[i,j+1]<>Wall Then Begin  
                AddEdge(VertA(i,j),VertB(i,j+1),Inf,0,ENum+2);  
                AddEdge(VertB(i,j+1),VertA(i,j),0,0,ENum)  
              End;  
              AddEdge(VertB(i,j),VertA(i,j),Inf,0,ENum+2);  
              AddEdge(VertA(i,j),VertB(i,j),0,0,ENum);  
              If Map[i,j]=Prize Then Begin {Клетка с  
образцом - две дополнительные дуги}  
                AddEdge(VertB(i,j),VertA(i,j),1,-1,ENum+2);  
                AddEdge(VertA(i,j),VertB(i,j),0,1,ENum)  
              End  
            End;  
          End;  
          SVert:=VertB(1,1); {Номер первой вершины - источник}  
          TVert:=VertA(n,m); {Номер последней вершины - сток}  
        End;  
      End;  
    End;  
  End;
```

Итак, из вершины 1 в вершину 12 необходимо «прогнать» поток мощности 2 (количество марсоходов), и его суммарная стоимость должна быть минимальной. Это основная идея. Для ее реализации подсчитаем потенциалы вершин сети. Они «говорят» о том, какая минимальная стоимость может быть «достав-

лена» в вершину. Естественно, что подсчет идет по всем увеличивающим цепям (в терминологии потоковых алгоритмов), заканчивающимся в этой вершине. Формирование цепей выполняется с учетом пропускных способностей дуг и потока, который можно по ним пропустить. Прохождение через клетку с образцом или, что то же самое, по дуге (i, j) с отрицательной стоимостью -1 уменьшает значение потенциала вершины j , но главное — сумма разности потенциалов вершин в любой цепи от источника к стоку должна быть неотрицательной.

Что нам это дает? Потенциалы просчитаны. Строим цепь — по аналогии с построением увеличивающей цепи в алгоритме Эдмонса—Карпа или кратчайшего пути в алгоритме Дейкстры (сумма разности потенциалов неотрицательна!). Затем находим остаточную сеть, изменяя при этом потенциалы вершин, и продолжаем процесс до тех пор, пока не будут найдены маршруты для всех марсоходов или построение цепи окажется невозможным.

Обратимся к деталям. Первый момент — подсчет потенциалов. Для хранения потенциалов введем массив Φ (его тип: $\text{TIntVect} = \text{Array} [0..V\text{Max}] \text{ Of Integer}$, где $V\text{Max}$ — максимальное количество вершин в сети). Первоначальные значения элементов массива — нулевые. Следующий фрагмент логики решает задачу (анализируя, проведем аналогию с алгоритмом Флойда — поиска кратчайших путей между всеми парами вершин графа или построения транзитивного замыкания в ориентированном графе):

```

For k:=1 To TVert Do {Количество итераций}
  For i:=1 To TVert Do {Цикл по вершинам сети}
    If <мощность множества  $G[i]$  больше нуля> Then
      For <по номерам элементов множества  $G[i]$ > Do
        {Значением элемента  $G[i]$  является номер дуги  $(q)$ ,
        выходящей из  $i$ }
          If  $E[q].\text{Capac} > 0$  Then {Пропускная
            способность дуги  $(i, E[q].Vy)$  больше нуля}
             $\Phi[E[q].Vy] := \text{Min}(\Phi[E[q].Vy], \Phi[i] + E[q].\text{Cost})$ ;
            {Пересчет потенциала}

```

Подсчитаем потенциалы вершин для рассматриваемого примера (начальные значения равны нулю).

На первой итерации ($k = 1$) первоначально изменяются потенциалы вершин 9, 10, 11 с нуля на -1 , а затем (за счет дуг 5, 7, 13) — вершин 5, 4, 6 (именно в такой очередности!). Массив Φ равен $(0, 0, 0, -1, -1, -1, 0, 0, -1, -1, -1, 0)$.

На второй итерации ($k = 2$) изменения претерпевают потенциалы вершин 10, 11 (с -1 на -2), 12 (с 0 на -1) и 6 (с -1 на -2).

На третьей итерации ($k = 3$) потенциал вершины 12 уменьшится на единицу. На следующих итерациях потенциалы вершин не изменятся. Окончательный вид Φ_i : (0, 0, 0, -1, -1, -2, 0, 0, -1, -2, -2, -2).

Методическое отступление. Следует освоить «ручное» построение сети для конкретных примеров и подсчет значений потенциалов вершин. Логика подсчета легко улучшается, например, путем исключения лишних итераций.

Второй момент. Предположим, что «увеличивающая» цепь построена, т. е. в какой-то структуре данных запомнены номера дуг от источника к стоку. Пусть это будет массив $Prev$. В рассматриваемом примере для первого марсохода значения элементов массива $Prev$ равны (0, 0, 1, 7, 5, 19, 3, 0, 11, 17, 23, 25). Идем из вершины 12 (рис. 9.18) по дуге 25 и попадаем в вершину 6, затем из 6-й вершины по дуге 19 переходим в 11-ю вершину, из 11-й по дуге 23 в 5-ю, из 5-й в 9-ю (дуга 5). Затем из 9-й вершины по дуге 11 переходим в 3-ю и т. д., пока не достигнем вершины-источника. Необходимо увеличить на этих дугах поток на единицу, а на обратных (для этой итерации) — на минус единицу. На следующих итерациях поток по дугам 11 и 23 не пойдет: его значение уже равно пропускной способности, однако из вершины 3 в 9 и 5 в 11 следующая цепь может быть продолжена по дугам типа 9 и 21 (рис. 9.18) — марсоходы могут проходить по одним и тем же клеткам. Кроме того, потенциалы вершин должны быть изменены на значение стоимости, которая передается с помощью построенной цепи, а для этого необходим еще один массив Val .

Измененную сеть (потоки на дугах и потенциалы вершин) назовем остаточной. Логика ее формирования имеет вид:

```

If Val[TVert]=Inf Then Exit {Блокирующий поток
не построен — сток не достигнут}
Else Begin {Строим остаточную сеть}
  For i:=1 To TVert Do {Пересчитываем потенциалы.
Начальное значение элементов массива Val равно Inf}
    If Val[i]<>Inf Then  $\Phi[i] := \Phi[i] + Val[i]$ 
    Else  $\Phi[i] := \Phi[i] + Val[TVert]$ ;
    t:=TVert; {Начинаем со стока и "едем" по
построенной цепи в обратном порядке до источника}
    While t>SVert Do Begin
      E[Prev[t]].Flow:=E[Prev[t]].Flow+1; {Изменяем
значение потока}

```

```

      E[E[Prev[t]].Rever].Flow:=-E[Prev[t]].Flow;
{Rever - обратная дуга. По обратной дуге "идет"
поток с противоположным знаком}
      t:=E[Prev[t]].Vx; {Переадресация к предыдущей
вершине цепи}
    End;
  End;
End;

```

Третий момент. При построении цепи из источника (*SVert*) в сток (*TVert*) требуется структура данных (назовем ее *Heap*) для хранения очередности обрабатываемых номеров вершин (обрабатывается всегда первый элемент из *Heap* - *Heap*[1]).

Обратимся к примеру. При построении первой цепи (в соответствии с нумерацией вершин и дуг на рис. 9.18) содержимое *Heap* изменяется следующим образом:

$$|1| \Rightarrow |7| \Rightarrow |3| \Rightarrow |9| \Rightarrow \begin{vmatrix} 5 \\ 4 \end{vmatrix} \Rightarrow \begin{vmatrix} 4 \\ 11 \end{vmatrix} \Rightarrow \begin{vmatrix} 11 \\ 10 \end{vmatrix} \Rightarrow \begin{vmatrix} 10 \\ 6 \end{vmatrix} \Rightarrow |6| \Rightarrow |12|.$$

Поэтому и будет построена цепь из вершин (1, 7, 3, 9, 5, 11, 6, 12), а не (1, 7, 3, 9, 4, 10, 6, 12). До вершины 6 уже по первой цепи потоком будет «дотянута» минимальная стоимость (она равна нулю) и оценка *Val*[6], а соответственно и *Prev*[6] не будет изменена при анализе на первой итерации (первый марсоход) второго способа достижения. На второй итерации в рассматриваемом примере строится цепь, описываемая массивами *Prev* = (0, 0, 1, 7, 5, 13, 3, 0, 9, 17, 20, 25) и *Val* = (0, *Inf*, 0, 1, 1, 1, 0, *Inf*, 1, 1, 1, 1).

Методическое отступление. На конкретных примерах целесообразно построить сеть (так, как это сделано на рис. 9.18) и в «ручном» режиме проверить логику построения потока заданной величины и минимальной стоимости с помощью приведенной ниже логики. Разработка примера, в котором при построении потока заданной мощности будут задействованы и обратные дуги, «украсит» занятие.

Procedure MaxFlowMinCost;

Var i,j,k,t:Integer;

Begin

<Подсчет потенциалов вершин>;

For k:=1 **To** r **Do** **Begin** {Цикл по марсоходам}

For i:=1 **To** TVert **Do** *Val*[i]:=*Inf*; {Бесконечная стоимость}

Val[*SVert*]:=0; {Стоимость источника}

<Подготовка к работе *Heap*. В *Heap*[0] хранится количество запомненных элементов>;

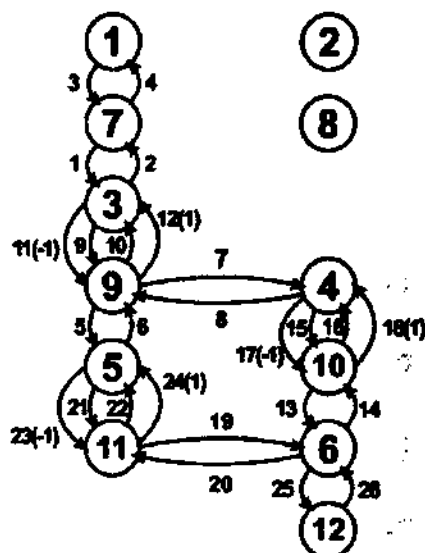


Рис. 9.18.

Repeat

If (Heap[0]>0) **And** (Val[Heap[1]]<>Inf)

Then t:=Heap[1] **Else** t:=0;

If t<>0 **Then** **Begin**

<Удалить первый элемент из Heap>;

If <мощность множества G[t] больше нуля>

{Значением элемента G[t] является номер дуги
выходящей из t}

For <по номерам (i) элементов множества G[t]> **Do**

If (E[q].Сарас-E[q].Flow)>0 **Then** **Begin**

{По дуге можно пропустить поток. Значение пропускной
способности дуги больше значения потока на ней}

j:=E[q].Vy; {Дуга q идет из вершины

t в вершину j}

If Val[j]>(Val[t]+E[q].Cost+Phi[t]-Phi[j])

{Ключевой оператор, стоимость "дотягиваемая"
поток до вершины j, может быть уменьшена}

Then **Begin**

Val[j]:=Val[t]+E[q].Cost+Phi[t]-Phi[j];

Prev[j]:=q;

<Добавить вершину j в Heap>;

End;

End;

End;

Until t=0;

```

    <Построение остаточной сети>;
  End;
End;

```

Единственный не рассмотренный пока вопрос — вывод результата. Очевидное замечание заключается в том, что это нельзя делать в логике нахождения потока при построении цепи, ибо поток может на каждой следующей итерации перераспределяться по дугам (в частности, и по обратным, несогласованным). Это отдельный фрагмент общей логики, исходными данными для которого является построенный поток, и его разработку можно предложить в качестве задания для самостоятельной работы.

Методические указания

Даже если алгоритмы по теме «Потоки в сетях» известны школьнику, данную задачу с трудом можно отнести к сложным: она **ОЧЕНЬ** сложная! Не умея же решать задачи на потоки в сетях, школьнику придется довольствоваться лишь частичным решением (как, вероятно, и было на этой олимпиаде).

Самостоятельные исследования

1. Сильно ли упрощается задача, если на каждую клетку, имеющую образец породы, может «ступать нога» только одного марсохода?
2. Образцы пород имеют разную стоимость. Требуется найти маршруты, обеспечивающие сбор образцов с максимальной суммарной стоимостью.

Игра «гекс»

Условие задачи

Цель игры для первого игрока — соединить своими фишками столбец с номером 1 со столбцом с номером n .

Правила «гекса». «Гекс» — стратегическая игра двух игроков на ромбовидной доске, составленной из $n \times n$ шестиугольников, как показано на рис. 9.19 для $n = 6$.

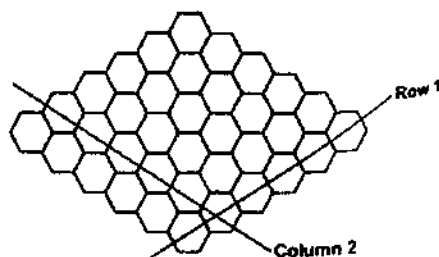


Рис. 9.19. Пример доски для игры «гекс»

1. Два игрока — это ваша программа и оценивающая библиотека.
2. Ваша программа всегда ходит первой.
3. Игроки по очереди ставят свои фишки на клетки доски.
4. Фишку можно ставить на любую свободную клетку доски.
5. Два шестиугольника *смежны*, если у них есть общая сторона (ребро).
6. Фишки одного игрока (как для участника, так и для оценивающей библиотеки), стоящие на смежных шестиугольниках, *связаны*.
7. Связанность транзитивна (и коммутативна), то есть если фишка *hex1* связана с фишкой *hex2* и фишка *hex2* связана с фишкой *hex3*, то *hex3* связана с *hex1*, а *hex1* — с *hex3*.

Задача:

- Требуется написать программу, играющую в «гекс».
- Цель первого игрока (вашей программы) — связать своими фишками столбец (*column*) с номером 1 со столбцом с номером *n*.
- Другой игрок (оценивающая программа) пытается связать своими фишками строку (*row*) с номером 1 со строкой с номером *n*.
- Если ваша программа играет оптимально, то она *всегда* побеждает.

Ввод и вывод

Ваша программа *не должна* читать и писать никаких файлов. Ваша программа *не должна* вводить данные с клавиатуры и выводить данные на экран. Она будет получать данные как значения функций из оценивающей библиотеки (*hex library*). Библиотека *сама* создаст файл HEX.OUT; вы должны игнорировать его содержимое.

В начале игры ваша программа получает игровую доску, на которой, может быть, уже размещены фишки, причем таким образом, как это могло получиться в ходе игры; в этой позиции первый игрок может достичь выигрыша. Чтобы определить размеры доски и положение фишек на ней, ваша программа может использовать функции оценивающей библиотеки GetMax («ДатьМаксимум») и LookAtBoard («ПосмотретьНаДоску»).

В начале игры на доске находится одинаковое количество фишек вашей программы и оценивающей библиотеки.

Ограничения:

1. Размер доски всегда находится в пределах от 1 до 20 включительно.
2. Ваша программа должна сделать до 200 ходов до завершения игры. Вся игра должна завершиться за 40 секунд. Гаранти-

руется, что оценивающая библиотека затратит на свои ходы в сумме не более 20 секунд.

Библиотека. Вы должны присоединить к своей программе библиотеку с названием `HexLib`. Примеры библиотек для каждого языка программирования, показывающие, как работает библиотека, находятся в каталоге задачи. Эти файлы называются `TESTHEX.CPP`, `TESTHEX.C`, `TESTHEX.PAS` и `TESTHEX.BAS`. Если вы используете *QuickBasic*, то должны присоединить библиотеку так: `QB /L HEXLIB`. Библиотека `HexLib` содержит следующие функции и процедуры (соответственно для *Pascal*, *C/C++* и *Basic*):

```
function LookAtBoard (row, column: integer): integer;
{ПосмотретьНаДоску(НомерСтроки, НомерСтолбца)}
int LookAtBoard (int row, int column);
declare function LookAtBoard cdecl (byval x as integer, byval y as integer)
```

Эта функция принимает такие значения:

- -1, если $row < 1$, или $row > n$, или $column < 1$, или $column > n$;
- 0, если на заданном поле нет никакой фишки;
- 1, если на этом поле стоит фишка вашей программы (1-го игрока);
- 2, если на этом поле стоит фишка оценивающей библиотеки (2-го игрока).

```
procedure PutHex (row, column: integer);
{ПоставитьФишку(НомерСтроки, НомерСтолбца)}
void PutHex (int row, int column);
declare sub PutHex cdecl (byval x as integer, byval y as integer)
```

Эта процедура ставит фишку вашей программы в поле с заданным номером строки и заданным номером столбца, если это поле не занято.

```
function GameIsOver: integer; {ИграОкончена}
int GameIsOver (void);
declare function GameIsOver cdecl ()
```

Эта функция принимает одно из следующих целочисленных значений:

- 0 — игра еще не окончена;
- 1 — все поля доски заняты фишками;

- 2 — ваша программа победила;
- 3 — оценивающая библиотека победила.

```
procedure MakeLibMove; {БиблиотекеХодить}  
void MakeLibMove(void);  
declare sub MakeLibMove cdecl ()
```

Эта процедура позволяет оценивающей библиотеке определить ее следующий ход и помещает ее фишку на доску. Изменение позиции на доске можно определить с помощью LookAtBoard и других функций.

```
function GetRow: integer; { ДатьСтроку }  
int GetRow (void);  
declare function GetRow cdecl ()
```

Эта функция принимает значение номера строки последней фишки, которую разместила оценивающая библиотека, или -1, если библиотека фишку не ставила. Эта функция при последовательных вызовах выдает одну и ту же величину, пока ваша программа в очередной раз не вызовет процедуру MakeLibMove.

```
function GetColumn: integer; {ДатьСтолбец}  
int GetColumn (void);  
declare function GetColumn cdecl ()
```

Эта функция принимает значение номера столбца последней фишки, которую разместила оценивающая библиотека, или -1, если библиотека фишку не ставила. Эта функция при последовательных вызовах выдает одну и ту же величину, пока ваша программа в очередной раз не вызовет процедуру MakeLibMove.

```
function GetMax: integer; {ДатьМаксимум}  
int GetMax (void);  
declare function GetMax cdecl ()
```

Значение этой функции всегда равно n — размеру доски.

Оценивание:

- если ваша программа выиграла игру, ей ставится полный балл за данный тест;
- если ваша программа проиграла игру, ей ставится 20% от полного балла за этот тест;
- если ваша программа прекращает работу, не доиграв до конца, или не укладывается в отведенное время, она получает 0 за этот тест.

Идея решения

Перебор всех вариантов игры и выбор одного из них с помощью некой оценивающей функции — это, вероятно, «тупиковая» ветвь решения, не выдерживающая критики по временным характеристикам.

Рассмотрим вариант, позволяющий достойно выйти из ситуации и не набрать за задачу (в реальное время тура) нулевое количество баллов.

Опишем поле игры с помощью графа. На рис. 9.20 показан граф для поля 4×4 . В нем введены две дополнительные вершины: s (источник) и t (сток). Вершины, кроме обычных номеров, имеют метки: 0 — вершина свободна, 1 — занята нашей фишкой, -1 — занята фишкой противника. Ребрам графа приписаны веса:

- от источника s до вершин, соответствующих первому столбцу, значение 0;
- от вершин последнего столбца до стока t — значение 0;
- между вершинами с метками 1 (обе вершины) — значение 0;
- между вершинами с метками 0-0 и 1-0, значение 0;
- всем ребрам, связанным с вершинами, имеющими метку -1 , — значение бесконечность (Inf).

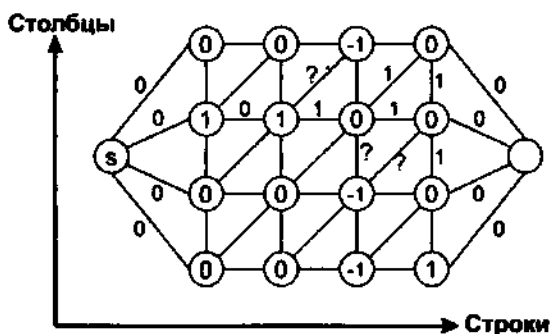


Рис. 9.20. Поле игры, представленное в виде графа для случая 4×4

Что нам дает такое представление? С помощью алгоритма Дейкстры за полиномиальное время находится кратчайший путь между s и t , т. е. мы получаем множество клеток (S_1), на которые можно поставить фишку (они имеют нулевое значение метки), и их количество ($first$). Но мы можем предположить, что и противник стремится выиграть за минимальное количество ходов. Построим аналогичный граф для противника: при этом изменяются связи s (с вершинами из первой строки), связи t (с вершинами из последней строки), а также веса ребер. Та-

ним образом, игра описывается с помощью двух графов. Найдем кратчайший путь для противника — множество клеток (S_2) и их количество (*second*).

Мы получили некую информацию для принятия решения. Далее возможны различные стратегии борьбы с противником. Если *first* или (*и*) *second* равны единице, то выбор хода очевиден. Иначе — одна из возможных стратегий:

- 1) если $first \leq second$ — из пересечения множеств S_1 и S_2 берем клетку, имеющую наибольшее количество соседей с меткой -1 ; так, на рис. 9.20 будет выбрана вершина, соответствующая клетке (3, 3);
- 2) если $first > second$ — в S_2 находим клетку, постановка фишки в которую максимально увеличивает кратчайший путь противника.

По первому правилу мы не только делаем ход, но и пытаемся максимально ухудшить положение противника. Возможно улучшение этого правила: например, если нет соседних клеток, занятых противником, то следует отдавать предпочтение клеткам, имеющим наибольшее количество не занятых игроками клеток.

Второе правило также допускает улучшения. Например, если есть несколько подходящих клеток, то следует выбирать их в соответствии с модификацией первого правила.

Методические указания

Для работы с этой задачей требуется библиотека. При ее наличии целесообразно организовать соревнование по поиску наилучшей стратегии игры. При хорошо структурированном программном коде внесение изменений для реализации найденных стратегий становится чисто техническим вопросом, поэтому вполне допустимо предоставить школьникам не только доступ к библиотеке, но и шаблон программы.

Самостоятельные исследования

Найдите контрпример для рассмотренной стратегии игры, т. е. такое расположение фишек на поле игры, при которой данная стратегия приведет к проигрышу.

«Прожорливая Шонгололо»

Условие задачи

Шонгололо — зулусское название многоножки. Она имеет длинное гладкое черное тело с большим количеством ног.



Шонгололо проедает съедобный «фрукт», который для данной задачи может рассматриваться как прямоугольный параллелепипед с целочисленными размерами l (длина), w (ширина) и h (высота).

Задание. Требуется написать программу, которая максимизирует число блоков единичного размера, съеденных Шонгололо без нарушения заданных ограничений. Программа должна выводить действия, которые Шонгололо совершает в процессе поедания блоков.

Шонгололо начинает процесс поедания за пределами фрукта. Первый блок, который должна съесть Шонгололо, имеет координаты 1,1,1, и после этого она передвигается в этот блок. Она останавливается, когда больше нет блоков, которые можно съесть, и блоков, в которые можно передвинуться.

Ограничения:

- Шонгололо занимает в процессе поедания в точности один пустой блок;
- За один раз Шонгололо может съесть только один заполненный блок;
- Шонгололо не может передвигаться в позицию, где она уже ранее была (то есть двигаться назад или пересекать свой путь);
- Шонгололо не может передвинуться в еще не съеденный блок или за пределы фрукта;
- Шонгололо может передвигаться в блоки или поедать блоки, имеющие общую грань с тем блоком, где находится Шонгололо сейчас. Она может поедать только те блоки, которые не имеют общих граней с пустыми (уже съеденными) блоками, не считая блока, в котором она находится.

Входные данные. Входными данными для программы являются три числа (целые), которые означают длину (l), ширину (w) и высоту (h) фрукта. Каждое из трех целых чисел l , w , h должно быть записано в отдельной строке. Числа находятся в интервале от 1 до 32 (включительно).

Пример входных данных.

TOXIC.DAT	Объяснение
2	Длина равна 2
3	Ширина равна 3
2	Высота равна 2

Выходные данные. Выходной файл должен содержать строки, которые начинаются либо с буквы «Е» (поедать), либо с буквы «М» (передвигаться). За буквой следуют три целых числа, представляющих координаты блока, который Шонгололо съест или в который перейдет. Для входных данных, приведенных выше, выходной файл будет иметь следующий вид.

Пример выходного файла (представленное здесь решение не обязательно оптимальное):

TOXIC.OUT	Объяснение
E 1 1 1	Съеден блок 1 1 1
M 1 1 1	Перемещение в блок 1 1 1
E 2 1 1	Съеден блок 2 1 1
E 1 1 2	Съеден блок 1 1 2
E 1 2 1	Съеден блок 1 2 1
M 1 2 1	Перемещение в блок 1 2 1
E 1 3 1	Съеден блок 1 3 1
M 1 3 1	Перемещение в блок 1 3 1
E 2 3 1	Съеден блок 2 3 1
E 1 3 2	Съеден блок 1 3 2
M 1 3 2	Перемещение в блок 1 3 2

Подсчет баллов:

Если Шонгололо нарушает заданные ограничения, то ваше решение получает 0 баллов;

общее количество баллов есть процентное отношение числа съеденных блоков в вашем решении к числу блоков в решении жюри;

итоговая оценка не может превышать 100%.

Решения

Судя по схеме подсчета баллов, задача не имеет точного алгоритма решения. Поэтому следует искать *эвристики*.

Первый шаг — упрощение задачи. Пусть у нас не трехмерная структура — параллелепипед, а обычный плоский прямоугольник. Экспериментальное заполнение прямоугольника по правилам Шонгололо показывает, что наиболее работоспособным методом является «змейка» с «пожиранием» соседних элементов, если это возможно.

На рис. 9.21 приведено заполнение прямоугольника 8*4. «Змейка» строится по горизонтали и имеет два витка. Аналогично строится «змейка» и в вертикальном направлении. Одна из клеток на рис. 9.21а выделена кружком. По логике построе-

ния «змейки», она в этой клетке должна поворачивать вверх, но справа, через одно поле, находится граница, поэтому следует сделать еще шаг вправо и затем выполнять поворот. В этом случае «Шонгололо» съедает на одну клетку больше (рис. 9.21а), чем при стандартном построении «змейки» (рис. 9.21б).

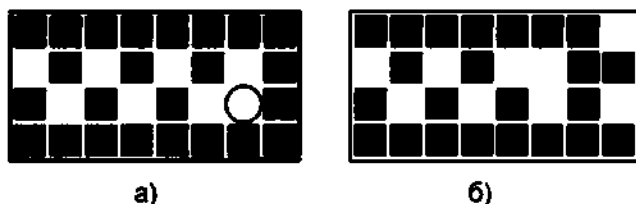


Рис. 9.21. «Плоский фрукт». Реализация заполнения по принципу «змейки» (черные квадратики — «съеденные клетки»)

Чем отличается трехмерный случай — случай параллелепипеда? Количеством направлений: здесь их шесть. В каждом направлении следует построить «змейку» и найти количество «съеденных клеток». После выбора направления следует повторно построить «змейку», но уже с выводом результата.

Общая логика имеет вид:

Procedure Solve;

Var i, sk, count: **Integer**; {Остальные переменные — глобальные}

Begin

ReadLn(tx,ty,tz); {Размеры параллелепипеда}
count:=0;

For i:=0 **To** 5 **Do Begin** {Выбор направления}

Case i **Of**

0 : **Begin** sizex:=tx; sizey:=ty; sizez:=tz; **End**;

1 : **Begin** sizex:=tx; sizey:=tz; sizez:=ty; **End**;

2 : **Begin** sizex:=ty; sizey:=tx; sizez:=tz; **End**;

3 : **Begin** sizex:=tz; sizey:=tx; sizez:=ty; **End**;

4 : **Begin** sizex:=tz; sizey:=ty; sizez:=tx; **End**;

5 : **Begin** sizex:=ty; sizey:=tz; sizez:=tx; **End**;

End;

<построить «змейку» в направлении i
и подсчитать количество «съеденных» клеток —
значение переменной sk>;

If sk>count **Then Begin**

count:=sk;

idir:=i; {Запомнить направление}

End;

<построить "змейку" в направлении *idir* с одновременным выводом результата>;

End;

End;

Задача построения «змейки» носит чисто технический характер (на технику программирования) и может быть предложена школьникам для самостоятельной работы.

Методические указания

Это задача скорее на технику программирования. Целесообразно ее исследовать и реализовать для двумерного случая, а затем дать трехмерный вариант как обобщение.

Самостоятельные исследования

Действительно ли предложенный способ поедания фрукта дает наилучший результат?

ЗАДАЧИ ВТОРОГО ТУРА

«Распознавание символов»

Условие задачи

Задача состоит в написании программы для распознавания символов.

Уточнение. Образ каждого символа состоит из 20 строк по 20 цифр в каждой. Значение цифр — 0 или 1. На рис. 9.22 показан способ представления образов символов в файле.

В файле FONT.DAT хранятся образы 27 символов в следующем порядке: $\square$ abcdefghijklmnopqrstuvwxyz, где $\square$ соответствует символу пробела.

Файл IMAGE.DAT содержит один или несколько образов символов, возможно, искаженных. Образ символа может иметь следующие искажения:

- максимум одна строка может быть продублирована, при этом дубль размещается за исходной строкой;
- максимум одна строка может быть пропущена;
- некоторые 0 могут быть заменены на 1;
- некоторые 1 могут быть заменены на 0.

Ни один образ символа не может быть искажен так, что одновременно одна строка продублирована, а одна пропущена.

Никакой образ символа не может быть искажен более чем на 30% заменами 0 и 1 по сравнению с его исходным представлением.

В случае дублирования строки она и ее дубль могут после этого исказиться, причем по-разному.

Задача. Написать программу для распознавания последовательности из одного или более символов, представленных в файле IMAGE.DAT, используя образы символов, задаваемые в файле FONT.DAT.

Распознайте образ символа, выбрав такой исходный образ, для получения которого требуется минимальное общее количество изменений 0 и 1 в искаженном образе, учитывая наиболее предпочтительное предположение о возможном дублировании или пропуске строки. В случае дублирования строки подсчитывайте искажения только в менее искаженной из двух результирующих строк. Все образы символов, используемые при тестировании, хорошо написанная программа сможет распознать.

Для каждого набора тестовых данных имеется только одно оптимальное решение.

Правильное решение будет использовать в точности все данные, представленные во входном файле IMAGE.DAT.

Входные данные. Оба входных файла начинаются с целого числа n ($19 \leq n \leq 1200$), которое задает количество следующих далее строк, как показано ниже:

```
n
(цифра1) (цифра2) (цифра3) ... (цифра20)
(цифра1) (цифра2) (цифра3) ... (цифра20)
```

Каждая строка содержит ровно 20 цифр без пробелов.

Файл FONT.DAT описывает образы символов и всегда содержит ровно 541 строку. Файл FONT.DAT может отличаться для различных тестов.

Выходные данные. Ваша программа должна создать файл IMAGE.OUT, который содержит строку распознанных символов в формате ASCII без разделителей. Если ваша программа не распознает отдельного символа, она должна вывести символ «?» в соответствующей позиции.

Внимание. Приведенный выше выходной формат не соответствует общим выходным форматам, изложенным в правилах проведения олимпиады.

Подсчет баллов. Баллы вычисляются в соответствии с процентом корректно распознанных образов символов.

Начало файла FONT.DAT
(пробел и символ «а»)

Пример файла IMAGE.DAT с искаженным символом «я»

[illegible]

Рис. 9.22. Пример входных файлов

Выходной файл

IMAGE.OUT	Объяснение
a	Распознан символ «a»

Рис. 9.23. Пример выходного файла

Идея решения

Предположим, что условия пропуска и добавления отдельных строк нет, — есть только искажения 1 и 0. В этом случае первый вариант решения очевиден. Считываем 20 строк из файла IMAGE.DAT, после чего осуществляем сравнение с данными по каждому из 27 образцов, находя при этом минимальное несовпадение, которое и будет определять распознанный символ.

В чем состоит усложнение задачи в случае пропуска или дублирования строк? В том, что мы не знаем количество строк файла IMAGE.DAT, соответствующих очередному символу.

Рассмотрим распознавание первого символа. Считываем 21 строку и выполняем последовательную обработку при предположении, что:

- символу соответствует 20 строк (действия описаны в предыдущем абзаце);
- символу соответствует 19 строк (из строк, описывающих каждый образец, последовательно убираем строки от 1 до 20 и сравниваем с описанием символа, подсчитывая при этом количество несовпадений);
- символу соответствует 21 строка (последовательно дублируем каждую из строк от 1 до 20 в образце и сравниваем с описанием символа, подсчитывая при этом количество несовпадений).

Находя при этом минимальное значение количества несовпадений, считаем, что мы не только распознали символ, но и определили количество строк, ему соответствующих. В зависимости от результата мы считываем определенное количество следующих строк файла IMAGE.DAT так, чтобы с учетом оставшихся их было 21, и продолжаем обработку.

Методические указания

Эта задача в данной постановке — скорее на технику программирования: на работу с файлами и массивами.

Самостоятельные исследования

Если на каких-либо шагах обработки значение минимального элемента (количество несовпадений) будет не единственным

Конечно, описанная выше идея решения при ее реализации даст, вероятно, почти полный балл на олимпиаде, но есть некая незавершенность в задаче, не позволяющая получить удовлетворения от ее решения.

Условие задачи

Карту можно представить в виде **блн**, состоящей из 1000×1000 ячеек. Местоположение города определяется одной ячейкой на карте. Названия городов должны быть расположены на карте в прямоугольниках, состоящих из ячеек. Такие прямоугольники называются *указателями* (рис. 9.24).

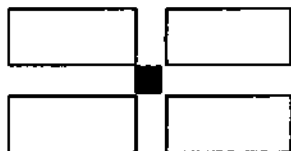


Рис. 9.24. Город с четырьмя возможными позициями его указателей

Расположение указателей должно удовлетворять следующим ограничениям:

- а) указатель города должен находиться в одной из четырех позиций относительно позиции города, как показано на рис. 9.24;
- б) указатели не должны перекрывать друг друга;
- в) указатели не должны перекрывать позиции городов;
- г) указатели не должны выходить за пределы карты.

Каждый указатель содержит все буквы названия города и пробел. Для каждого названия города ширина и высота букв задана; пробел имеет такие же размеры, как и буква.

	L	a	n	g	a	q									
■							■								
								P	a	a	r	l	q		
q	C	e	r	e	s										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	

Символ q обозначает пробел
Символ $\blacksquare$ обозначает положение города

Рис. 9.25. Часть карты

Самый левый столбец карты имеет номер «0» по горизонтали, а самая нижняя строка карты имеет номер «0» по вертикали. На рис. 9.25 показана нижняя левая часть карты для городов *Langa* в позиции (0,3), *Ceres* в позиции (6,1) и *Paarl* в позиции (7,3). Все указатели размещены корректно, но это размещение не является единственным правильным.

Задание. Ваша программа должна для каждого города ввести его расположение на карте, затем размеры букв названия города и само название. После этого программа должна расположить на карте как можно больше указателей без нарушения вышеназванных ограничений и вывести расположение указателей, которые размещены на карте.

Входные данные. Входной файл начинается со строки, содержащей целое число n , задающее количество городов на карте. Затем для каждого города следует строка, содержащая:

- координату города по горизонтали (x),
- координату города по вертикали (y),
- ширину (w) каждой буквы в названии города в виде целого числа,
- высоту (h) каждой буквы в названии города в виде целого числа,
- само название города.

Название города состоит из одного слова. Количество городов не превышает 1000. Длина названия города не превышает 200 букв.

Пример входного файла:

MAPS.DAT	Объяснение
3	$n = 3$
0 3 1 1 Langa	$x = 0, y = 3, w = 1, h = 1$
6 1 1 1 Ceres	
7 3 1 2 Paarl	

Выходные данные. Ваша программа должна вывести n строк. Каждая строка должна содержать координату по горизонтали и следующую за ней координату по вертикали верхней левой ячейки указателя города. Если ваша программа не может разместить указатель на карте, то она должна вывести 1-1. Эти строки должны быть выведены в том же порядке, в котором заданы города во входном файле. Два числа в строке должны быть разделены одним пробелом.

Пример выходного файла:

MAPS.OUT	Объяснение
1 4	Указатель <i>Langa</i> — на (1, 4)
0 0	Указатель <i>Ceres</i> — на (0, 0)
8 2	Указатель <i>Paarl</i> — на (8, 2)

Подсчет баллов. Для набора данных каждого теста количество баллов будет представлять собой следующее:

- процентное отношение количества названий городов, которое располагает на карте ваша программа, к количеству названий городов в решении научного комитета;
- минимальное количество баллов равно 0%, максимальное — 100%;
- нуль баллов, если какой-нибудь указатель нарушает ограничения;
- нуль баллов, если указатели не соответствуют заданным городам.

Идея решения

Привести пример расположения городов на карте и размеров указателей, при которых их размещение невозможно, не составляет труда. Это говорит о том, что поиск точного алгоритма с учетом всех возможных пересечений положений указателей не предполагается. На это «наталкивает» и система оценки решения, поэтому следует искать некое приближенное размещение указателей на карте.

Первым шагом является упорядочивание указателей по их размеру в убывающем порядке и размещение их на карте с помощью «жадной» логики. При этом часть указателей окажется размещенной на карте, а часть — нет. Естественно, что при размещении учитывается и выход указателя за пределы карты, и пересечения с позициями городов, и пересечения с ранее размещенными указателями. После получения первого приближенного решения возможны различные последовательные методы его улучшения.

Введем понятие «итерации» (количество размещенных ранее, до итерации, указателей зафиксировано в переменной *count*). На итерации просматриваются все указатели в соответствии с определенным порядком (пока он определен в соответствии с убыванием размеров указателей). У нас для части указателей уже найдено место, для части — еще нет. Для первых из них делаем попытку размещения в другой возможной для указателя позиции, а при невозможности «передвижения» — оставляем его на

месте. Для второго типа указателей (не размещенных на карте) делаем попытку их размещения. С учетом изменения положений ранее размещенных указателей у нас появляется надежда на размещение части из них. Количество размещенных указателей на итерации — значение переменной *newcount*. Если *newcount* > *count*, то запоминаем размещение и изменяем значение *count*.

Итак, у нас есть приближенное решение, полученное с помощью «жадной» логики. Первый способ его улучшения. Введем счетчик числа итерации. Пока значение счетчика меньше четырех (четыре положения указателя), выполняем итерации. Если на итерации решение улучшено, то сбрасываем счетчик числа итерации в ноль. Другими словами, мы продолжаем обработку, пока в результате четырех последовательно выполненных итераций решение не улучшается или не истекло время решения задачи.

Предположим, что мы закончили обработку на первом способе улучшения (за четыре итерации решение не улучшено) и у нас еще осталось время, отведенное для решения задачи. Лучшее приближенное решение получено и зафиксировано в соответствующих структурах данных (количество размещенных указателей — *maxcount*). «Отправной» точкой его получения являлось решение, даваемое «жадной» логикой, и не очевидно, что это было лучшим начальным приближением.

Следующим (вторым) способом улучшения решения является задание случайного порядка на множестве указателей, т. е. не в соответствии с размерами указателей. Естественно, что после каждой такой операции все указатели считаются неразмещенными, а затем следует выполнение действий по первому способу улучшения решения с нулевым начальным приближением (нет размещенных указателей). Общая логика имеет вид:

```

While <не истекло время решения задачи> Do Begin
  <задать случайный порядок на множестве указателей>;
  t:=0;
  count:=0;
  newcount:=0;
  While t<4 Do Begin
    <выполнить итерацию>; {количество размещенных
указателей — значение переменной newcount}
    If newcount>count Then Begin
      count:= newcount;
      t:=0;
    End
  Else t:=t+1;

```


If maxcount < count Then <запомнить размещение и изменить значение maxcount>; {maxcount — количество размещенных указателей в лучшем решении}

End;

End;

Зачем требуется второй способ улучшения? Ответ достаточно очевиден, но рассмотрим это на примере конкретной карты (рис. 9.26).

Даны восемь городов. Города 1, 2, 3, 8 находятся в углах карты и их указатели могут быть размещены, соответственно, в позициях *c*, *g*, *h*, *a*. Города 5, 7 находятся на границе, для их указателей возможны места *d*, *f* и *b*, *e*. Размеры указателей совпадают. Исключение составляет город с номером 4. Его указатель можно разместить в позициях *a*, *b* и нельзя разместить в позициях *c*, *d*.

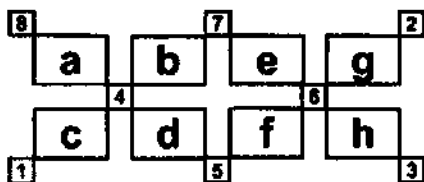


Рис. 9.26. Пример карты

После первой итерации получаем следующее размещение: 1 — *c*; 2 — *g*; 3 — *h*; 4 — *a*; 5 — *d*; 6 — *e*; 7 — *b*. Указатель города 8 оказывается не размещенным, и место *f* оказывается свободным. При первом способе улучшения вторая итерация даст решение: 1 — *c*; 2 — *g*; 3 — *h*; 4 — *a*; 5 — *f*; 6 — *e*; 7 — *b*, а третья вернет ситуацию к первому размещению, и процесс заикнется. Однако есть полное размещение, указателей: 1 — *c*; 2 — *g*; 3 — *h*; 4 — *b*; 5 — *d*; 6 — *f*; 7 — *e*; 8 — *a*, и оно достигается при определенном порядке просмотра городов, который может быть получен случайным образом.

Методические указания

Рассмотренный вариант решения задачи не позволит «упасть лицом в грязь» на олимпиаде, так как вероятно, что жюри и научный комитет ориентировались в своих рассуждениях на что-то аналогичное, ибо «не боги горшки обжигают», а такие же люди. Из-за отсутствия точного решения данной задаче присуща некая незавершенность, но в методическом плане она имеет определенную ценность как проблема, на которой отрабатываются навыки полного использования отведенного ресурса времени и совершенствуется техника программирования.

Самостоятельные исследования

Нами рассмотрена одна схема получения приближенного решения, и, естественно, она может быть улучшена. Например,

на стадии предварительной обработки находятся указатели, размещаемые только в одной из четырех позиций, или для некоторых из указателей есть позиции, в которых они «никому не мешают». Продолжите поиск эвристик.

«Складирование контейнеров»

Условие задачи

Компания транспортных перевозок «Нептун» имеет склад контейнеров. На склад принимаются контейнеры для хранения и последующей отправки.

Контейнеры поступают на склад для хранения через каждый час. Срок хранения на складе — целое количество часов. Когда прибывает контейнер, в его документах есть ожидаемое время его отправки. Первый контейнер прибывает во время, равное 1. В действительности может оказаться, что контейнер надо будет отправить раньше или позже ожидаемого времени в пределах ± 5 часов.

В этой задаче время выражается в часах целыми положительными числами, не превышающими 150.

Кран, работающий в помещении склада, перемещает контейнеры в склад, из склада и иногда переставляет их внутри склада. Кран может действовать в пределах склада.

Задача. Вам надо написать программу с хорошей стратегией приема, хранения и отправки контейнеров. Хорошая стратегия минимизирует общее количество перемещений крана.

Склад имеет форму прямоугольного параллелепипеда. Его длина (x), ширина (y) и высота (z) доступны вашей программе.

Каждый контейнер представляет собой куб размером $1 \times 1 \times 1$. Контейнеры складываются в стопки (стеки) друг на друга (рис. 9.27). Кран может брать только верхний контейнер из любой стопки.

Перемещение контейнера с одного места на другое — это одно перемещение крана. Все перемещения крана происходят между приемом и отправкой контейнеров. Перемещение краном контейнера с одного места на другое осуществляется мгновенно.

Если склад полон, ваша программа должна отказаться принимать новые контейнеры. Ваша программа может оказаться менее эффективной или неспособной продолжать работу, если склад почти полон. Ваша программа может отказаться принимать новые контейнеры в любой момент.

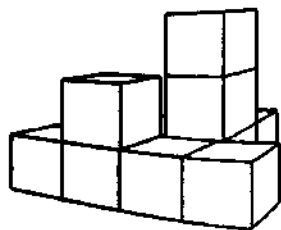


Рис. 9.27. Пример расположения контейнеров

Ввод. От вашей программы требуется взаимодействовать с моделирующей библиотекой, которая будет сообщать данные, и которой ваша программа будет передавать действия и сообщения. В начале работы вашей программы склад пуст.

При тестировании вашей программы библиотека будет выдавать осмысленные (разумные) величины для небольшого набора тестовых данных. Каждый контейнер обозначается уникальным положительным целым числом.

Ваша программа в любое время может вызывать следующие функции:

```
int GetX(); {Дать x}
function GetX: integer;
DECLARE FUNCTION GetX CDECL ()
```

Всегда принимает значение длины склада (integer).

```
int GetY(); {Дать y}
function GetY: integer;
DECLARE FUNCTION GetY CDECL ()
```

Всегда принимает значение ширины склада (integer).

```
int GetZ(); {Дать z}
function GetZ: integer;
DECLARE FUNCTION GetZ CDECL ()
```

Всегда принимает значение высоты склада (integer).

x, y, z не превосходят 32.

Далее описаны функции, которые дают информацию о последовательности действий (прибытие или отправка контейнера). Прием контейнера происходит точно в какой-либо час, а запросы на отправку приходят в промежутках между часами. Таким образом, для учета времени каждое прибытие контейнера означает, что прошел один час.

```
int GetNextContainer(); {Дать Следующий Контейнер}
function GetNextContainer: integer;
DECLARE FUNCTION GetNextContainer CDECL ()
```

Принимает положительное целое значение — номер следующего контейнера, который надо принять на хранение или отправить. Если контейнеров больше нет ни для приема, ни для отправки, функция принимает значение 0, что означает, что ваша программа должна закончить работу, даже если на складе еще есть контейнеры.

```
int GetNextAction();{Дать Следующее Действие}
function GetNextAction: integer;
DECLARE FUNCTION GetNextAction CDECL ()
```

Принимает положительное значение, представляющее действие, которое надо сделать: 1 — принять новый контейнер, 2 — отправить контейнер.

```
int GetNextStorageTime();{Дать Следующее Время Хранения}
function GetNextStorageTime: integer;
DECLARE FUNCTION GetNextStorageTime CDECL ()
```

Принимает значение ожидаемого времени в часах (абсолютное время), когда контейнер надо будет отправлять. Это нужно, чтобы ваша программа планировала свою работу; на самом деле запрос на отправку может прийти немного в другое время, которое может отличаться от планируемого не более чем на 5 часов. Эта функция принимает содержательное значение, только если GetNextAction равна 1.

Порядок вызова этих трех функций не имеет значения. Последовательные вызовы GetNextContainer, GetNextAction и GetNextStorageTime всегда будут возвращать информацию об одном и том же контейнере, пока не произойдет одно из следующих событий:

- очередной контейнер будет отклонен;
- принят на хранение;
- отправлен.

После этого данные функции будут сообщать информацию о следующем контейнере.

Вывод. Всякий раз, когда ваша программа получила нужную ей информацию о следующем контейнере, используйте такие функции для действий на складе:

```
int MoveContainer(int x1, int y1, int x2, int y2);
{Двигать Контейнер}
function MoveContainer(x1, y1, x2, y2: integer):
integer;
DECLARE FUNCTION MoveContainer CDECL (BYVAL x1 AS
INTEGER, BYVAL y1 AS INTEGER, BYVAL x2 AS INTEGER,
BYVAL y2 AS INTEGER)
```

Перемещает один контейнер с верха стопки с координатами $x1, y1$ на верх стопки с координатами $x2, y2$. Принимает значение 1, если действие правильно, и 0, если действие неправильно (то есть невозможно).

```
void RefuseContainer(); {Отказать Контейнеру}  
procedure RefuseContainer;  
DECLARE SUB RefuseContainer CDECL ()
```

Отказывает в приеме пришедшего контейнера.

```
void StoreArrivingContainer(int x, int y); {Принять  
Прибывший Контейнер}  
procedure StoreArrivingContainer(x, y: integer);  
DECLARE SUB StoreArrivingContainer CDECL (BYVAL x  
AS INTEGER, BYVAL y AS INTEGER)
```

Принимает прибывший контейнер на хранение на верху
стопки с координатами x, y .

```
void RemoveContainer(int x, int y); {Убрать Контейнер}  
procedure RemoveContainer(x, y: integer);  
DECLARE SUB RemoveContainer CDECL (BYVAL x AS  
INTEGER, BYVAL y AS INTEGER)
```

Отправляет со склада один контейнер с верха стопки с коор-
динатами x, y .

Если ваша программа не может ничего делать, она должна
закончить работу. Неверные перемещения игнорируются биб-
лиотекой и не влияют на состояние модели и на оценку реше-
ния. От вашей программы не требуется выводить что-либо в
файлы. Библиотека, с которой взаимодействует ваша програм-
ма, записывает события в файл. Этот файл используется при
проверке.

Порядок работы

Ваша программа должна получать информацию о контейне-
ре, который надо обработать следующим. Затем она должна пе-
ремещать контейнеры краном, если это нужно, а затем или
класть на хранение, или отправить, или отказаться принимать
контейнер.

Библиотека

Вам дается библиотека с именем StackLib, которую вы
должны присоединить к вашей программе. Стандартные биб-
лиотеки C и C++ содержат эту библиотеку и автоматически бу-
дут присоединяться к вашей программе, если вы подготовите
соответствующий файл заголовка (*header file*).

Если вы используете QuickBasic, вы должны подключить
библиотеку таким образом: QB /L STACKLIB. Примеры исход-
ных текстов представлены в каталоге задачи под именами
TESTSTK.BAS, TESTSTK.PAS, TESTSTK.CPP и TESTSTK.C.

Оценивание

Программа будет проверяться с различными наборами данных, для каждого набора данных ее производительность будет сравниваться с наиболее эффективным решением, известным научному комитету, при учете следующих факторов:

- общее количество перемещений крана вашей программой;
- штраф в 5 перемещений налагается за каждый контейнер, который программа отказалась принять на хранение;
- штраф в 5 перемещений налагается за каждый не сохраненный и неотправленный контейнер (в случае, если программа нормально завершается, а требуемые операции еще не закончены);
- общий счет вычисляется по отношению к лучшему известному научному комитету решению;
- если программа превышает необходимое количество операций более чем вдвое, она оценивается 0 баллами;
- минимальная оценка — 0%, максимальная — 100%.

Идея решения

Общая логика решения «читаема» из формулировки задачи и достаточно легко «набрасывается», ибо особых сложностей и с выбором структур данных нет:

Begin

 <узнать размеры склада xx , yy , zz >;

$t := \text{GetNextContainer}$;

 While $t <> 0$ Do Begin

$a := \text{GetNextAction}$;

 If <склад заполнен> And $(a = 1)$ Then RefuseContainer

 Else

 If $a = 1$ Then Begin

$b := \text{GetNextStorageTime}$;

$\text{time}[t] := b$; {Запоминаем время отправки

контейнера}

 <поиск места для хранения контейнера,

результат — значение переменных mx , my >;

 If <место не найдено> Then RefuseContainer

 Else <размещение контейнера в стопке с

координатами mx , my >;

 End

 Else Begin

 <поиск на складе контейнера с номером t ,

результат — значение переменных mx , my , mz >;

 If <контейнер не сверху> Then <переместить

контейнеры с верха стопки на наиболее подходящие

места в других стопках или временно в другие стопки с последующим возвратом в эту стопку>; (В первом случае логика поиска места совпадает с той, что использована при первоначальном размещении контейнеров)

RemoveContainer(Mx, My);

End;

t:=GetNextContainer;

End;

End.

Хорошая стратегия приема, хранения и отправки контейнеров, при которой общее количество перемещений крана имеет минимальное значение, определяется логикой фрагмента «поиск места для хранения контейнера». Фактически мы обязаны поддерживать работу с $xx*yy$ (площадь склада) стеками. В каждом стеке контейнеры должны быть упорядочены по времени их отправки. Место контейнера с минимальным временем отгрузки — на верху стека.

Предположим, что в задаче используется точное время отправки контейнера, т. е. условия «контейнер надо будет отправить раньше или позже ожидаемого времени в пределах ± 5 часов» нет. При наличии свободных мест (есть пустой стек) контейнер размещается на одном из них.

Пусть поступило $xx*yy$ контейнеров и ни один из них не отправлен, т. е. свободных мест на складе нет. Тогда очередной контейнер должен быть размещен в одном из стеков (стопок). Если время отправки поступившего контейнера меньше хотя бы одного времени отправки контейнера, находящегося вверху стопки, то решение очевидно — размещаем контейнер вверху соответствующей стопки. А если нет?

Вот в этот момент и начинается «стратегия». Предположим, что принято решение о выборе стопки с минимальным количеством элементов. В этом случае элементы с верха стопки (она должна быть «правильной» по времени отправки и после операции размещения) перемещаются в другие стопки. Пусть обратно в эту стопку они не перемещаются, тогда и те стопки объявлены быть «правильными». А это — рекурсивный процесс, так как для перемещаемых контейнеров, в свою очередь, может не выполняться условие «правильности» при добавлении новых контейнеров (мы пока считаем, что ограничение по высоте стопок выполняется всегда). Чтобы рекурсивного процесса не было, перемещаемые с верха стопки контейнеры после размещения поступившего на обработку следует возвращать обратно в свою же стопку. Итак, для исключения рекурсивности независимо от выбранного критерия (мы пока рассматриваем минимальность высоты стопки) контейнеры с верха выбранной стопки

следует возвращать обратно, а для временного их хранения может быть выбрана любая подходящая стопка.

Вернемся к критерию выбора стопки. Пусть по времени отправки очередному контейнеру предшествуют q контейнеров, и известны стопки, в которых они размещены. Пример приведен на рис. 9.28 (площадь склада 2×2 , возможны только 4 стопки) ... закрашенные кружки обозначают времена отправки ранее поступивших контейнеров, символ «*» — время отправки очередного контейнера, а цифры обозначают номера стопок. Подсчитаем, сколько из q контейнеров принадлежит каждой стопке (как минимум один есть, иначе контейнер можно было бы разместить вверху какой-либо стопки!). Выберем минимальное значение — оно определяет стопку, в которую следует поместить очередной контейнер. Так, для нашего примера, будет выбрана первая стопка, а не третья, как по предыдущему варианту критерия. Из первой стопки один контейнер временно перемещается в другую стопку. Поступивший на обработку контейнер идет в первую стопку, а затем в нее возвращается взятый контейнер.

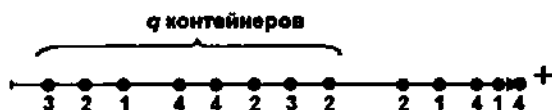


Рис. 9.28. Пример распределения контейнеров по стопкам

Условие в формулировке задачи — «может оказаться, что контейнер надо будет отправить раньше или позже ожидаемого времени в пределах 5 часов» — влияет только на выбор стопки, но не на общую стратегию. Выбор стопки необходимо делать так, чтобы время отгрузки соседних контейнеров было больше 10 часов. Однако время в задаче выражается в часах целыми положительными числами, не превышающими 150, а количество контейнеров может превышать значение 32000, поэтому делать это требование обязательным вряд ли разумно.

Методические указания

Только наличие моделирующей библиотеки делает работу с этой задачей содержательной. При ее отсутствии задачу следует переформулировать и тщательно разработать систему тестов.

Самостоятельные исследования

Пусть снимаемые с верха стопки контейнеры распределяются по другим стопкам, а не возвращаются обратно. Придумайте пример поступления контейнеров на склад, в котором при поступлении очередного контейнера осуществляются перемещения контейнеров во всех сформированных ранее стопках.

ЗАДАЧИ ПЕРВОГО ТУРА

«Контакт»

Условие задачи

Доктор Astro Insky работает в радиотелескопическом центре. Недавно она заметила очень странное пульсирующее микроволновое излучение, исходящее непосредственно из центра галактики. Посланы ли эти сигналы какой-нибудь внеземной формой разумной жизни? Или это простое сердцебиение звезд?

Задача. Вы должны помочь Доктору Insky отыскать истину, создав инструмент для анализа битовых последовательностей в файлах, которые она записала. Доктор Insky хочет найти битовые подпоследовательности длиной от A до B включительно, которые встречаются в записанной последовательности наиболее часто. Подпоследовательности могут перекрываться. Определяются n наибольших различных частот (т. е. количество вхождений). Учитываются подпоследовательности, которые встречаются хотя бы один раз.

Входные данные. Файл CONTACT.IN содержит данные в следующем формате:

- первая строка содержит целое A — минимальную длину подпоследовательности;
- вторая строка содержит целое B — максимальную длину подпоследовательности;
- третья строка содержит целое n — количество различных частот вхождения;
- четвертая строка содержит последовательность символов 0 и 1, ограниченную символом 2.

Пример входного файла:

```
2
4
10
01010010010001000111101100001
01001100111100001001001111001
00000002
```

В примере требуется отыскать десять (третья строка) наибольших частот вхождения подпоследовательностей длиной от двух (первая строка) до четырех (вторая строка) в последовательности (четвертая строка файла):

```
0101001001000100011110110000101
0011001111000010010011110010000
000
```

Обратите внимание на то, что четвертая строка входного файла разделена на несколько строк для того, чтобы поместиться на бумаге. В этом примере подпоследовательность 100 встречается 12 раз, а подпоследовательность 1000 — 5 раз. Наиболее часто встречается подпоследовательность 00.

Выходные данные. Файл CONTACT.OUT содержит не более n строк, каждая из которых содержит одну из n наибольших частот вхождения и соответствующие ей подпоследовательности. Строки должны располагаться в порядке убывания частот вхождения и иметь следующий вид: *частота подпосл. подпосл. ... подпосл.*

Частота — это количество вхождений указанных далее подпоследовательностей. Подпоследовательности в каждой строке должны располагаться в порядке убывания длины. Подпоследовательности одинаковой длины должны располагаться в порядке убывания их числового значения. Если в последовательности встречаются менее n различных частот, выходной файл будет содержать меньше n строк.

Пример выходного файла. Для указанного ранее примера выходной файл имеет следующий вид:

```
23 00
15 10 01
12 100
11 001 000 11
10 010
8 0100
7 1001 0010
6 0000 111
5 1000 110 011
4 1100 0011 0001
```

Ограничения. Размер входного файла не превышает 2 Мбайт. Параметры A , B и n имеют такие ограничения: $0 < n \leq 20$, $0 < A \leq B \leq 12$.

Обсуждение задачи

Количество различных подпоследовательностей не так велико, как кажется. Всего их 8191 ($2^1 + 2^2 + 2^3 + \dots + 2^{12} = 2^{13} - 1 = 8191$).

Введем массив d . Первоначально подсчитаем в нем частоты встречаемости каждой подпоследовательности; при этом значение подпоследовательности должно быть индексом для обращения к элементу массива d . Однако для реализации этой идеи следует решить одну проблему. Например, «00» и «000» дают одно и то же значение индекса. Для ее решения мы воспользуемся системой счисления с основанием «три».

Преобразуем при вводе исходную последовательность из 0 и 1 в последовательность из 1 и 2. Тогда исходная последовательность из формулировки задачи после преобразования будет иметь вид:

1212112111211122221221111212112211222211112112112222112111111.

При этом все подпоследовательности после такого преобразования становятся различимыми.

Функция преобразования строки в индекс имеет вид:

```
Function GetCode(s:String):Integer;
  Var t,code,i:Integer;
  Begin
    code:=0;
    t:=1;
    For i:=Length(s) DownTo 1 Do Begin
      If s[i]='1' Then code:=code + t
      Else code:=code + 2*t;
      t:=t*3;
    End;
    GetCode:=code;
  End;
```

Методическое отступление. Функцию обратного преобразования из кода в строку можно продемонстрировать или предложить в качестве самостоятельного упражнения по ходу занятия.:

```
Function GetString(code:Integer):String;
  Var s:String;
  Begin
    s:='';
    While code<>0 Do Begin
      If code Mod 3=1 Then s:='0'+s Else s:='1'+s;
      code:=code Div 3;
    End;
    GetString:=s;
  End;
```

Если бы задача сводилась к подсчету частоты встречаемости подпоследовательностей, то следующая простая логика (приведем фрагмент) достаточна для ее решения:

```

For i:=1 To m Do Begin {m — длина исходной
последовательности}
  s:='';
  For j:=0 To min(B,m-i) Do Begin {Функция min —
выбор минимального из двух чисел}
    s:=s+x[i+j]; {x — исходная последовательность
символов в троичной системе счисления}
    If j>=A Then d[GetCode(s)]:=d[GetCode(s)]+1;
{Два раза вызываем GetCode, но это — только для
наглядности}
  End
End;

```

Так, после обработки получим:

$$\begin{aligned}
 d[4] &= d[11_3] = d[00_2] = 23; \\
 d[8] &= d[22_3] = d[11_2] = 11; \\
 d[13] &= d[111_3] = d[000_2] = 11; \\
 d[14] &= d[112_3] = d[001_2] = 11.
 \end{aligned}$$

Итак, частоты подсчитаны. Осталось только организовать вывод результата в соответствии с требованиями формулировки задачи. Один из вариантов (при достаточной оперативной памяти) заключается в построении списков (цепочек) кодов, имеющих одинаковые частоты встречаемости. Для организации связей мы будем использовать все тот же массив d , а для хранения указателей на первые элементы цепочек — массив $head$. Так, в нашем примере по 11 раз встречаются три подпоследовательности. Для этого случая $head[11] = 8$, $d[8] = 13$, $d[13] = 14$ и $d[14] = -1$ (-1 — признак конца цепочки). Другими словами, индексом к элементам массива $head$ является частота встречаемости, а в массиве d мы связываем ссылками все элементы, имеющие эту частоту. Связи между элементами цепочки устанавливаются в соответствии с требованиями вывода (в порядке убывания длины, а при одинаковой длине — в порядке убывания их числовых значений).

Методическое отступление. Частоту «три» в рассматриваемом примере имеют четыре подпоследовательности:

$$\begin{aligned}
 d[23] &= d[212_3] = d[101_2] = 3; \\
 d[53] &= d[1222_3] = d[0111_2] = 3; \\
 d[79] &= d[2221_3] = d[1110_2] = 3; \\
 d[80] &= d[2222_3] = d[1111_2] = 3.
 \end{aligned}$$

Ответ на вопрос (и аналогичные вопросы) о значении *head[3]* и новых значений элементов массива *d* позволит закрепить понимание сути решения и перейти к реализации логики. Лучший программный код этого фрагмента может являться элементом небольшого соревнования.

Вывод результата после проделанной работы осуществляется по следующей схеме (приведем ее набросок):

```
...
For i:=<максимальная частота> DownTo 1 Do {Выводим
в порядке убывания частот}
  If head[i]<>0 Then Begin {Цепочка не пуста}
    Write(i, ' '); {Выводим частоту}
    n:=n-1;
    j:=head[i];
    While j<>-1 Do Begin {Идем по цепочке}
      Write(GetString(j)); {Выводим подпоследовательность}
      <вывод символа "пробел" или перевода на начало
следующей строки>;
      j:=d[j]; {Переходим к следующему элементу цепочки}
    End;
    If n=0 Then Break; {Количество требуемых частот
выбрано}
  End;
```

Методические указания

В этой задаче используется знание следующих тем: «Системы счисления», «Обработка строк», «Указатели», что делает ее достаточно ценной в методическом плане. Возможности по созданию различных вариантов решения зависят от искусства синтеза в единое целое структур данных и управления вычислительным процессом. Оценка лучшего программного кода, его показ и разбор для остальных школьников усиливает ее методическую значимость.

Самостоятельные исследования

Оцените усложнение задачи при расширении алфавита — когда исходные последовательности состоят не только из символов «0» и «1», а, например, из четырех символов.

«Лампы для праздника»

Условие задачи

Для освещения заключительного вечера IOI'98 имеется *n* цветных ламп, пронумерованных от 1 до *n*. Четыре кнопки позволяют управлять лампами следующим образом:

- кнопка 1 — когда эта кнопка нажата, все лампы изменяют свое состояние: те, что были включены, становятся выключенными, те, что были выключены — включаются;
- кнопка 2 — изменяет состояние всех ламп, имеющих нечетные номера;
- кнопка 3 — изменяет состояние всех ламп, имеющих четные номера;
- кнопка 4 — изменяет состояние всех ламп, имеющих номера, вычисляемые по формуле $3k + 1$ (где $k \geq 0$), то есть 1, 4, 7, ...

Имеется счетчик C , который учитывает (хранит) суммарное число нажатий всех кнопок.

В начале вечера все лампы были включены, а счетчик C равен нулю.

Задача. Задано значение счетчика C и информация о конечном состоянии некоторых ламп. Напишите программу для определения всех различных возможных конечных (окончательных) конфигураций n ламп, чтобы каждая конфигурация соответствовала заданной информации.

Входные данные. Файл, имеющий имя `PARTY.IN`, содержит четыре строки, задающие количество ламп n , конечное значение счетчика C нажатых клавиш и состояние некоторых ламп в окончательной конфигурации.

В первой строке содержится число n , во второй строке — конечное значение счетчика C . Третья строка содержит список номеров ламп, о которых известно, что в конечной конфигурации они включены. Номера ламп в строке отделены друг от друга одним пробелом, и список заканчивается числом -1 . Четвертая строка содержит список номеров ламп, о которых известно, что в окончательной конфигурации они выключены. Номера ламп в строке отделены друг от друга одним пробелом, и список заканчивается числом -1 .

Пример:

```
10
1
-1
7 -1
```

Здесь число ламп — 10, и только одна кнопка была нажата. Лампа 7 выключена в конечной конфигурации.

Выходные данные. Файл `PARTY.OUT` должен содержать все возможные различные окончательные конфигурации (без повторов) всех ламп. Каждая возможная конфигурация должна быть записана на отдельной строке. Конфигурации могут быть перечислены в произвольном порядке.

Каждая строка содержит n символов, где первый символ представляет состояние лампы номер 1, а последний символ представляет состояние лампы номер n . 0 (нуль) означает, что лампа выключена, а 1 (единица) означает, что лампа включена.

Пример:

```
0000000000
0110110110
0101010101
```

Видим, что имеется три возможные различные конечные конфигурации:

- все лампы выключены;
- лампы 1, 4, 7, 10 выключены, а лампы 2, 3, 5, 6, 8, 9 включены;
- лампы 1, 3, 5, 7, 9 выключены, а лампы 2, 4, 6, 8, 10 включены.

Ограничения:

- параметры n и C ограничены: $10 \leq n \leq 100$, $1 \leq C \leq 10000$;
- количество ламп, о которых известно, что в конечной конфигурации они включены, меньше или равно 2;
- количество ламп, о которых известно, что в конечной конфигурации они выключены, меньше или равно 2;
- гарантируется, что существует хотя бы одна конечная конфигурация для каждого входного файла.

Идея решения

Значение $C = 10000$ и количество различных вариантов 4^C явно подсказывают, что должна быть найдена некая идея решения, ибо простой перебор вариантов — занятие бессмысленное.

Первое, что следует из анализа типа преобразований, — то, что нажатие какой-либо кнопки два раза подряд не изменяет состояния лампочек (назовем это свойство «идентичностью»). Второе, главное свойство, — «коммутативность»: последовательность преобразований (a, b) эквивалентна последовательности (b, a) .

Пусть, например, у нас есть последовательность преобразований: $(1, 2, 3, 4, 1, 2, 3)$. Используя свойства идентичности и коммутативности, получим: $(1, 2, 3, 4, 1, 2, 3) \equiv (1, 2, 3, 1, 4, 2, 3) \equiv (1, 2, 1, 3, 4, 2, 3) \equiv (1, 1, 2, 3, 4, 2, 3) \equiv (2, 3, 4, 2, 3) \equiv \dots \equiv (4)$.

Методическое отступление. В данный момент занятия полезно произвести «ручной» просчет нескольких цепочек преобразований.

Вывод заключается в том, что любое значение C , большее 4, можно уменьшить на 2. Поэтому нечетные C , большие 4, заме-

няются на C , равное 3, а четные C , большие 5, на C , равное 4. Оператор `If (C>4) Then C:=4-C Mod 2`, примененный после ввода значения C , дает нам необходимое значение.

Остается перебрать различные варианты конечных конфигураций (сгенерировать все подмножества преобразований) при $C \leq 4$. Их — всего 15: (1), (2), (3), (4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4), (1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4), (1, 2, 3, 4), что, безусловно, гораздо меньше, чем значение 4^C .

Пусть в массиве `HEnd[1..n]` при вводе данных сформирована конечная конфигурация типа (1, 2, 0, 0, 2, ..., 1, 0, 2), где `HEnd[i] = 0` говорит о том, что лампа с номером i выключена, `HEnd[i] = 1` — что лампа с номером i находится во включенном состоянии и `HEnd[i] = 2` — что состояние лампы i не имеет значения в конечной конфигурации. Тогда, если генерируемая конфигурация ламп хранится в массиве `Now` (начальное состояние — все лампы включены), то проверка совпадения текущего состояния ламп с одним из конечных осуществляется с помощью следующей простой процедуры:

```

Procedure Check;
  Var i: Integer;
  Begin
    For i:=1 To n Do
      If Not(HEnd[i] In [Now[i],2]) Then Exit;
  {Нет совпадения}
  For i:=1 To n Do Write(Now[i]);
  WriteLn;
End;

```

А вся логика обработки сводится к рекурсивной процедуре (первый ее вызов производится командой `Solve(C,0)`):

```

Procedure Solve(k, l: Integer);
  Var i: Integer;
  Begin
    If k Mod 2=0 Then Check;
    If k>0 Then
      For i:=l+1 To 4 Do Begin
        MakeOp(i); {Процедура реализации нажатия кнопок.
Изменяется состояние массива Now, i — номер кнопки}
        Solve(k-1, i);
        MakeOp(i);
      End;
    End;
  End;

```


Методические указания

При изучении комбинаторных алгоритмов, а именно — при генерации всех подмножеств заданного множества эта задача может быть использована в качестве самостоятельного упражнения либо материала для дополнительного занятия.

Самостоятельные исследования

Изменение значения n , количества кнопок и содержания их действия (какие лампы при этом изменяют свое состояние) до определенных пределов не изменяют смысла задачи. Однако если дано два произвольных состояния ламп, одно из которых начальное, а другое конечное, и требуется за минимальное нажатие кнопок получить из начального конечное, то это — уже другая задача!

«Звездная ночь»

Домашние задачи

Посмотрите на ночное небо, где светящиеся звезды объединены в созвездия (кластеры) различных форм. *Созвездие* — это тесная группа рядом расположенных звезд, имеющих соседей звезды либо по горизонтали, либо по вертикали, либо по диагонали. Созвездие не может быть частью другого созвездия. Созвездия на небе могут быть одинаковыми. Два созвездия являются *одинаковыми*, если они имеют одну и ту же форму независимо от их ориентации. В общем случае число возможных ориентаций для созвездия равно восьми, как показано на рис. 10.1.

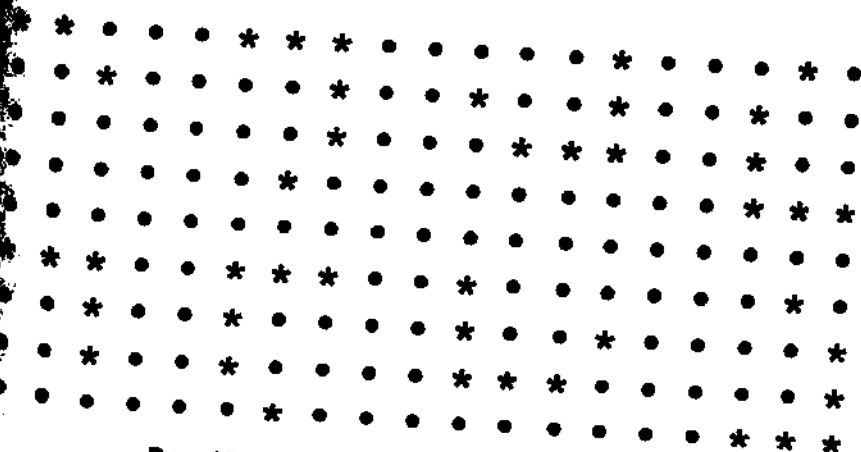


Рис. 10.1. Восемь одинаковых созвездий

Ночное небо представляется картой неба, которая, в свою очередь, является двумерной матрицей, состоящей из нулей и единиц. Элемент этой матрицы содержит цифру 1, если это звезда, и цифру 0 — в противном случае.

Задача. Для заданной карты неба отметить все созвездия, используя при этом маленькие буквы. Одинаковые созвездия должны быть отмечены одной и той же буквой; различные созвездия должны быть отмечены различными маленькими буквами. Для того, чтобы отметить созвездие, необходимо заменить каждую 1 (единицу) в созвездии соответствующей маленькой буквой.

Входные данные. Входной файл с именем STARRY.IN содержит в первых двух строках ширину w и высоту h матрицы, соответствующей карте неба. Далее в этом файле следуют h строк, в каждой из которых содержится w символов.

Пример:

```

23
15
10001000000000010000000
01111100011111000101101
01000000010001000111111
00000000010101000101111
00000111010001000000000
00001001011111000000000
10000001000000000000000
00101000000111110010000
00001000000100010011111
00000001110101010100010
00000100110100010000000
00010001110111110000000
00100001110000000100000
00001000100001000100101
00000001110001000111000

```

В этом случае матрица, соответствующая карте неба, имеет ширину 23 и высоту 15. Данному входному файлу соответствует изображение карты неба, приведенное на рис. 10.2.

Выходные данные. Файл STARRY.OUT содержит ту же самую матрицу карты неба, что и файл STARRY.IN, только созвездия в ней отмечены так, как описано в разделе «Задание».

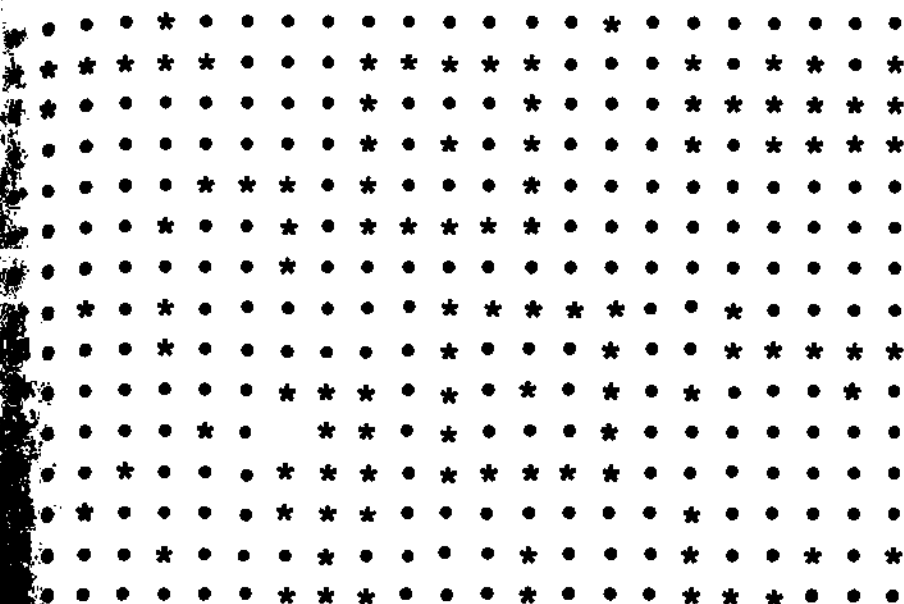


Рис. 10.2. Изображение карты неба

Пример:

```

00a0000000000b00000000
aaaa000cccc000d0dd0d
0000000c000c000ddddd
0000000c0b0c000d0ddd
000eee0c000c000000000
0e00e0cccc000000000
0000e00000000000000
0f000000cccc00a0000
0f000000c000c00aaaaa
0000ddd0c0b0c0a000a0
00b00dd0c000c0000000
0g000ddd0cccc0000000
0000ddd0000000e00000
00b000d0000f000e000b
0000ddd000f000eee000

```

Это один из возможных вариантов выходного файла для составленного выше входного файла. Этому выходному файлу соответствует изображение на рис. 10.3.

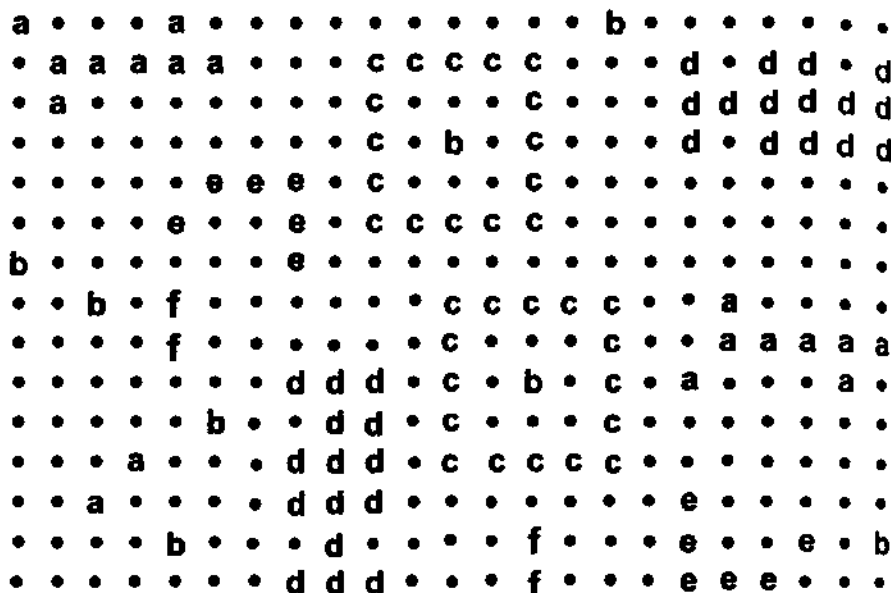


Рис. 10.3. Изображение карты неба с отмеченными созвездиями

Ограничения:

- $0 \leq w$ (ширина карты неба) ≤ 100 ;
- $0 \leq h$ (высота карты неба) ≤ 100 ;
- $0 \leq \text{число созвездий} \leq 500$;
- $0 \leq \text{число различных созвездий} \leq 26$ (a...z);
- $1 \leq \text{число звезд в каждом созвездии} \leq 160$.

Обсуждение задачи

Пусть карта описывается массивом s ($s: \text{TMap}$, где $\text{TMap} = \text{Array}[1..XMax, 1..YMax] \text{ Of Byte}$, и $XMax = 100$, максимальный размер карты). Вместо 1, описывающей звезду созвездия, элементу w (массив того же размера, но символьного типа) в процессе обработки мы будем присваивать символ, соответствующий данному созвездию.

Попробуем «набросать» общую логику, а затем начнем ее детализировать.

...

For i:=1 To h Do

For j:=1 To w Do

If s[i,j]=1 Then Begin

<Выделение созвездия, начиная с клетки (i,j)>;

<Получение всех восьми описаний созвездия>;

<Проверка ранее полученных созвездий с восемью описаниями выделенного созвездия. Если совпадения не было, то полученное созвездие запоминается и помечается очередной, еще свободной буквой. При совпадении — оно помечается найденным символом>;
End;

...

Выделение созвездия (при достаточной оперативной памяти) — это обычная логика поиска выхода из лабиринта:

```

Procedure GetStarry(i,j:Integer);
Var l:Integer;
Begin
  If (i<=h) And (i>0) And (j<=w) And (j>0) And
    (s[i,j]=1) Then Begin
    s[i,j]:=0;
    v[i,j]:=1; {v имеет тот же тип, что и s}
    vx:=min(vx,i); {Функция выбора минимального из
    двух чисел}
    vy:=min(vy,j);
    For l:=1 To 8 Do GetStarry(i+dx[l],j+dy[l]);
    {Массивы констант: dx: Array[1..8] Of Integer =
    (0, 0, 1, 1, 1, -1, 1, -1), dy: Array[1..8] Of
    Integer = (1, -1, 0, 1, -1, 0, 1, -1)}
  End;
End;

```

Созвездие запоминается в массиве *v*. Дополнением к логике выхода из лабиринта является запоминание координат левого верхнего угла прямоугольника, минимального по отношению к созвездию и содержащего его, в переменных *ux* и *uy*.

Воспользуемся множественным типом данных для описания созвездия в памяти компьютера (*TStarry* = Array[1..XMax] Of Set Of Byte). Введя массив *t*, элементами которого является *TStarry* (*t*:Array[1..8] Of *TStarry*), нормализуем описание конкретного созвездия (перевод из *v* в *t* с использованием *ux* и *uy*):

```

Procedure Normalize(v:TMap; Var q:TStarry); {Вызов
процедуры — Normalize(v,t[1])}
Var i,j:Integer;
Begin
  For i:=1 To XMax Do q[i]:=[];
  For i:=1 To h Do

```

```

For j:=1 To w Do
  If v[i,j]=1 Then q[i-vx+1]:=q[i-vx+1]+(j-vy+1);
End;

```

Так, первое созвездие на рис. 10.2 после нормализации имеет описание: $t[1] = ([1,5], [2..6], [2])$.

Методическое отступление. В этот момент обсуждения можно дать учащимся нормализованные описания всех созвездий на рис. 10.2.

После нормализации нашей задачей становится получение всех восьми (точнее семи — одно уже получено) созвездий, отличающихся только ориентацией. Из нормализованного созвездия в результате последовательных поворотов на 90° по часовой стрелке мы получаем еще три созвездия (рис. 10.4). Затем первое созвездие отражаем относительно вертикальной оси симметрии (рис. 10.5) и снова выполняем три таких же поворота. Приведем для наглядности вариант возможной реализации этих преобразований и вызывающую их логику:

```

Procedure Rotate(u:TStarry; Var q:TStarry);
Var i,j,vh:Integer;
Begin
  vh:=0;
  For i:=1 To XMax Do q[i]:=[];
  For i:=1 To XMax Do

```

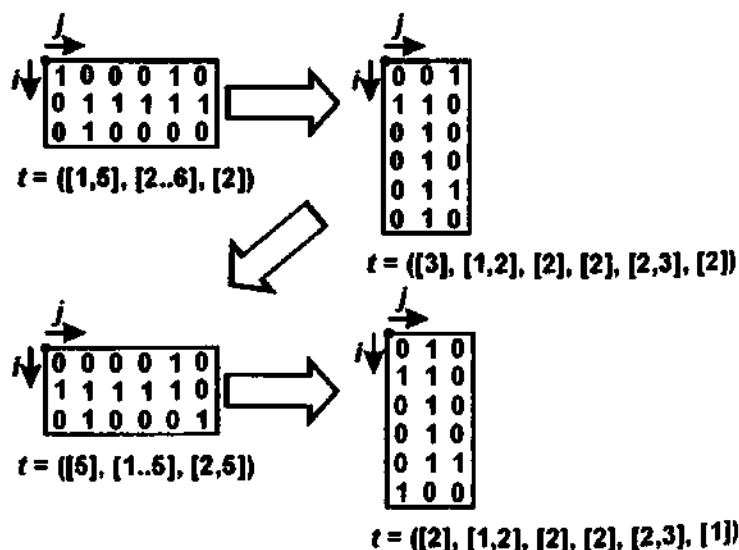


Рис. 10.4. Повороты созвездия

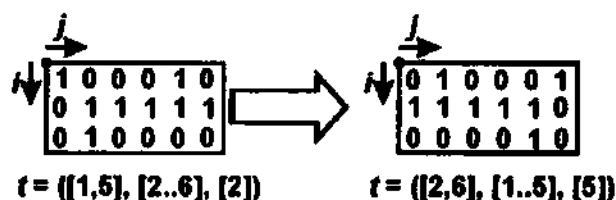


Рис. 10.5. Отражение созвездия

```

u[i] <> [] Then vh:=max(vh,i); {max - функция
    деления максимального из двух чисел}
For i:=1 To XMax Do
  For j:=1 To XMax Do
    If j In u[i] Then q[j]:=q[j]+[vh-i+1];
  End;
Procedure Reflect(u:TStarry; Var q:TStarry);
Var i,j,vw:Integer;
Begin
  vw:=0;
  For i:=1 To XMax Do q[i]:=[];
  For i:=1 To XMax Do
    For j:=1 To XMax Do
      If j In u[i] Then vw:=max(vw,j);
    For i:=1 To XMax Do
      For j:=1 To XMax Do
        If j In u[i] Then q[i]:=q[i]+[vw-j+1];
      End;
    End;
  End;
End;

```

вызывающая логика:

```

Realize(v,t[1]); {Из описания созвездия в массиве
получаем его нормализованный вид в t[1]}
For l:=2 To 4 Do Rotate(t[l-1],t[1]); {Получаем
второе, третье и четвертое описание}
Reflect(t[1],t[5]); {Путем операции отражения получаем
пятое описание}
For l:=6 To 8 Do Rotate(t[l-1],t[1]); {Путем поворотов
получаем шестое, седьмое и восьмое описание}

```

Методическое отступление. Получение всех описаний для любого из созвездий на рис. 10.2, а также произвольных созвездий — уместная и полезная для понимания работа.

стало бы детализировать последний фрагмент общей логики — сравнение ранее полученных созвездий с восемью полу-

ченными описаниями уже выделенного созвездия. Если такого совпадения не окажется, то созвездие требуется пометить первой свободной буквой и запомнить; в противном случае — только пометить в результате как повторное. Для хранения созвездий целесообразно ввести структуру данных типа `TNow=Array ['a'..'z'] Of TStarry` и запоминать количество найденных созвездий (*last*). В этом случае проблема решается организацией цикла по элементам массива *r* (типа *TNow*) и вложенным циклом по восьми полученным описаниям созвездия.

Методическое отступление. При обсуждении задачи реализация логики сравнения может быть вынесена в качестве самостоятельного упражнения. В этом случае иногда бывает целесообразно дать два необходимых для ее написания «кирпичика» — функцию сравнения двух описаний созвездий и процедуру пометки созвездия символом:

```
Function Eq(a,b:TStarry):Boolean;
  Var i:Integer;
      pp:Boolean;
  Begin
    pp:=True;
    For i:=1 To XMax Do pp:=pp And (a[i]=b[i]);
    Eq:=pp;
  End;
```

```
Procedure MetStarry(v:TMap; c:Char);
  Var i,j:Integer;
  Begin
    For i:=1 To XMax Do
      For j:=1 To XMax Do
        If v[i,j]=1 Then w[i,j]:=c; {v — первоначальное
(не нормализованное) описание выделенного созвездия,
w — результирующее описание карты неба}
      End;
```

Методические рекомендации

Методическая ценность данной задачи заключается в развитии умения конструировать сложные структуры данных. Так, в развернутом виде тип `TNow=Array ['a'..'z'] Of TStarry` имеет вид `Array ['a'..'z'] Of Array [1..XMax] Of Set Of Byte`. Это массив, элементами которого также являются массивы с элементами множественного типа. Здесь главным является не просто усложнение структур данных самих по себе, а сохранение при этом простого и ясного кода, создаваемого по принципу «как говорю, так и пишу».

Самостоятельные исследования

Обсуждение данной задачи было построено на предположении, что оперативной памяти достаточно для хранения всех структур данных. Например, для представления элемента карты используется целый байт, хотя достаточно одного бита. Разработайте вариант решения задачи с побитовым хранением описаний созвездий (карты).

ЗАДАЧИ ВТОРОГО ТУРА

«Картинки»

Условие задачи

На стене наклеено некоторое количество плакатов, фотографий и других картинок прямоугольной формы. Все их стороны либо вертикальны, либо горизонтальны. Каждый прямоугольник может быть частично или полностью перекрыт другими прямоугольниками. *Периметром* называется общая длина внутренних и внешних фигур, полученных путем наложения прямоугольников.

Задача. Написать программу, вычисляющую значение полученного периметра.

На рис. 10.6 показан набор из 7 прямоугольников

• Соответствующие границы полученной фигуры показаны на рис. 10.7.

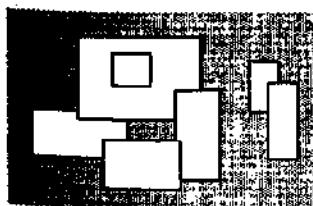


Рис. 10.6. Набор из 7
прямоугольников



Рис. 10.7. Границы фигур, полученных
наложением прямоугольников

Вершины всех прямоугольников имеют целочисленные координаты.

Входные данные. Первая строка входного файла PICTURE.IN содержит число прямоугольников, наклеенных на стене. В каждой последующей строке приводятся целочисленные координаты нижней левой вершины и верхней правой вершины каждого прямоугольника. Значения этих координат даны как упорядо-

ченные пары, состоящие из x -координаты и следующей за ней y -координаты.

Пример:

```

7
-15 0 5 10
-5 8 20 25
15 -4 24 14
0 -6 16 4
2 15 10 22
30 10 36 20
34 0 40 16

```

Этот файл соответствует примеру, приведенному на рис. 16.6.

Выходные данные. Выходной файл должен содержать только одну строку с неотрицательным целым числом, которое является периметром для заданных прямоугольников.

Пример: 228.

Это соответствует выходному файлу для примера, приведенного выше.

Ограничения:

- $0 \leq \text{число прямоугольников} < 5000$;
 - все координаты находятся в пределах $[-1000, 1000]$, и каждый заданный прямоугольник имеет положительную площадь.
- Для хранения результата достаточно 32 бита.

Обсуждение задачи

Упростим задачу. Пусть на координатной прямой задано своими координатами какое-то количество отрезков. Последние могут пересекаться, находиться один внутри другого и т. д. Требуется определить количество связных областей, образованных этими отрезками. На рис. 10.8 показано четыре отрезка с координатами $[2, 8)$, $[5, 10)$, $[12, 17)$, $[14, 16)$, которые образуют две связные области.

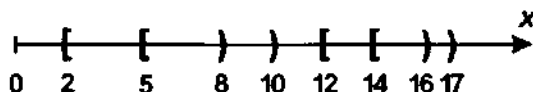


Рис. 10.8. Пример расположения четырех отрезков на прямой

Для нахождения количества связных областей требуется в отдельном массиве «пометить» концы отрезков признаками $+1$ (для левого конца) и -1 (для правого конца). Для нашего примера такой массив выглядит следующим образом: $p = (1, -1, 1, -1, 1, -1, 1, -1)$. При сортировке координат отрезков по увеличению одновременно переставляются и их признаки. В нашем

Примере $p = (1, 1, -1, -1, 1, 1, -1, -1)$. Подсчитывая сумму значений признаков и фиксируя равенство нулю ее текущего значения, мы определяем количество связанных областей.

Вернемся к исходной задаче. Очевидно, периметр можно подсчитывать отдельно, по частям — его составляющие по осям X и Y . На каждой оси (X или Y) мы получаем систему отрезков, разбивающую ось на определенные участки, «покрытые» одним или несколькими отрезками. Для каждого участка оси (X или Y) мы «выписываем» координаты отрезков по Y (X) с соответствующими признаками их начала и конца. Выполняем сортировку и подсчитываем количество связанных областей (t). Вклад в периметр по оси X (Y) определяется как $2 \times t \times l_{x(y)}$, где $l_{x(y)}$ — длина участка по оси X (Y).

Рассмотрим пример (рис. 10.9). Подсчитаем составляющую периметра по оси Y :

Участок $[0, 1)$ — одна связанная область; вклад в периметр — $2 \times 1 \times 1 = 2$;

Участок $[1, 2)$ — массив признаков p после сортировки равен $(1, 1, -1, 1, -1, -1)$; одна связанная область; вклад в периметр — $2 \times 1 \times 1 = 2$;

Участок $[2, 3)$, $p = (1, -1, 1, -1)$ — две связанные области; вклад в периметр — $2 \times 2 \times 1 = 4$;

Участок $[3, 4)$ — одна связанная область, вклад в периметр — $2 \times 1 \times 1 = 2$;

Участок $[4, 6)$; вклад в периметр — $2 \times 1 \times 2 = 4$.

Составляющая периметра по оси Y равна 14.

Аналогично подсчет выполняется и по оси X . Он дает значение 18. Ответ для приведенного примера — 32.

Методические указания

Методическая ценность задачи состоит не столько в ее содержательном наполнении (оно достаточно известно), сколько в создании простого и изящного программного кода. Рекомендуется применять технологию «сверху вниз», метод последовательного уточнения. Следует добиться от школьника (особенно на начальной стадии обучения) того, чтобы код «читался» и в нем не было повторов, — а это не так просто! Всегда в ход идет аргумент: «ведь программа работает!».

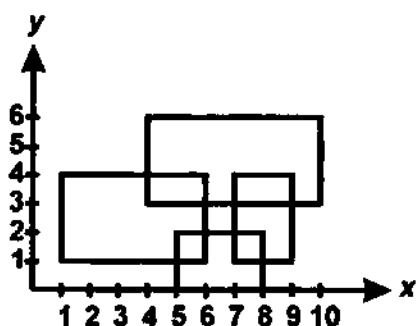


Рис. 10.9. Пример из 4 прямоугольников

Наиболее целесообразно использовать задачу при изучении алгоритмов сортировки.

Самостоятельные исследования

1. Найдите площадь пересечения n прямоугольников со сторонами, параллельными осям координат.

Входные данные. Число n (количество прямоугольников $2 \leq n \leq 1000$) и n четверок вещественных чисел $x_{i1}, y_{i1}, x_{i2}, y_{i2}$, задающих координаты левого нижнего и правого верхнего углов прямоугольника.

Выходные данные. Площадь пересечения (общей части) всех прямоугольников, либо 0 — признак того, что они не пересекаются.

2. На плоскости задано n прямоугольников со сторонами, параллельными осям координат. Координаты вершин — вещественные числа с точностью до двух знаков после запятой. Найдите площадь фигуры, получающейся в результате объединения прямоугольников. Формат данных и ограничения — см. в предыдущем задании.

«Камелот»

Условие задачи

Давным давно король Артур и рыцари Круглого Стола обычно собирались на Новый год, чтобы отпраздновать свою дружбу. В память об этих событиях придумали настольную игру «Камелот» для одного игрока, в которой фигура Короля и несколько фигур Рыцарей произвольно располагаются в различных клетках доски.

Доска имеет размер 8×8 клеток (рис. 10.10).

Король может перемещаться в любую смежную клетку доски из $\bullet$ в $\circ$ так, как показано на рис. 10.11, при этом не выходя за пределы доски.

Рыцарь может перемещаться из клетки $\bullet$ в одну из клеток $\circ$, как показано на рис. 10.12, при этом не выходя за пределы доски.

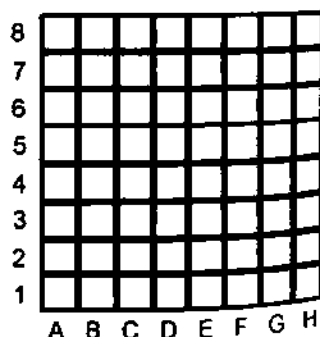


Рис. 10.10. Доска

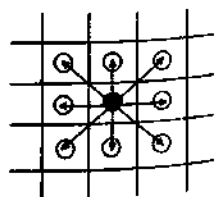


Рис. 10.11. Все возможные перемещения Короля

Во время игры игрок может поместить более одной фигуры в одну клетку. Клетки считаются достаточно большими, и не возникает препятствий для свободного перемещения фигур.

Игроку необходимо так перемещать фигуры Короля и Рыцарей, чтобы собрать их всех в какой-то одной клетке за наименьшее число ходов. Ходы фигурами необходимо делать по правилам, описанным выше. Дополнительно к этому, если Король и один или более Рыцарей находятся в одной клетке, игрок может Короля и одного из Рыцарей переместить вместе по правилам перемещения Рыцаря и считать это одним ходом.

Задача. Напишите программу для вычисления минимального количества ходов, необходимых для перемещения всех фигур в одну клетку доски.

Входные данные. Файл CAMELOT.IN содержит одну строку символов, описывающую начальное расположение фигур на доске. Строка содержит последовательность клеток доски, первая из которых — клетка Короля, остальные — клетки Рыцарей. Каждая клетка описывается парой «буква-цифра». Буква означает горизонтальную, а цифра — вертикальную координату клетки доски. Все фигуры в начале игры расположены в указанных клетках.

Пример. Король расположен в клетке D4. Четыре Рыцаря — в клетках A3, A8, H1 и H8:

A3A8H1H8

Выходные данные. Файл CAMELOT.OUT должен содержать единственную строку с положительным целым числом, обозначающим минимальное число ходов игрока, необходимых для перемещения всех фигур в одну клетку доски.

Пример: 10

Ограничения. $0 \leq$ количество Рыцарей ≤ 63 .

Идея решения

Методом «волны» (обходом в ширину) подсчитаем минимальное количество ходов, за которое Рыцарь может попасть из клетки с координатами (x, y) в клетку (i, j) . Сделаем это для каждой пар клеток и запомним результаты в четырехмерной мат-

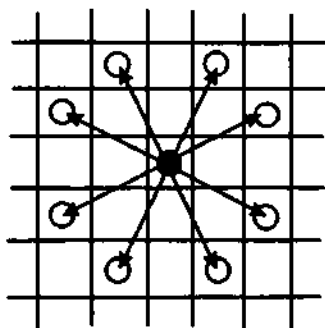


Рис. 10.12. Все возможные перемещения Рыцаря

рице $p[1..8, 1..8, 1..8, 1..8]$. Количество ходов для Короля из клетки (x, y) в (i, j) равно $\text{Max}(\text{Abs}(i-x), \text{Abs}(j-y))$. Если запретить совместное перемещение Короля и Рыцаря, то задача сводится к поиску клетки (i, j) , сумма расстояний до которой от клеток, занимаемых Рыцарями и Королем, имеет минимальное значение (рис. 10.13 а). Предположим, что в клетке (x_r, y_r) Король встречается с Рыцарем, имеющим номер k , и дальнейшее перемещение до клетки (i, j) они выполняют совместно (рис. 10.13 б). В этом случае требуется подсчитать количество ходов Короля от клетки (x_k, y_k) до (x_r, y_r) и прибавить к нему величины $p[x[k], y[k], x_r, y_r]$ и $p[x_r, y_r, i, j]$. Прделав эту работу для всех пар (i, j) и (x_r, y_r) , не забывая при этом фиксировать минимальную оценку, мы получим искомый результат.

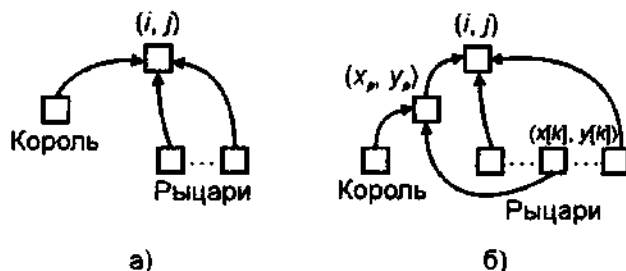


Рис. 10.13. а) — Король и Рыцари перемещаются в клетку (i, j) самостоятельно. б) — Король до клетки (x_r, y_r) перемещается самостоятельно, а из клетки (x_r, y_r) в клетку (i, j) — с помощью Рыцаря с номером k

Методическое отступление. Задание на разработку программного кода может сопровождаться демонстрацией наброска основной логики. Ориентируясь на этот набросок, школьник должен дописать оставшуюся часть программы.

Procedure Solve;

Var $i, j, k, x_r, y_r, tr, rr$:Integer;

Begin

$res := \text{maxint}$; {maxint — максимальное целое число}

For $i := 1$ To 8 **Do** {Циклы по координатам конечной клетки}

For $j := 1$ To 8 **Do Begin**

$rr := \text{Max}(\text{Abs}(i-x_k), \text{Abs}(j-y_k));$

{Max — максимальное из двух чисел}

For $k := 1$ To n **Do** $rr := rr + p[x[k], y[k], i, j];$

{Подсчитываем "стоимость" самостоятельного перемещения в клетку (i, j) ; n — количество Рыцарей}

$tr := rr;$

```

For xр:=1 To 8 Do {Циклы по координатам
клетки встречи Короля с Рыцарями}
  For ур:=1 To 8 Do
    For k:=1 To n Do
      tr:=Min(tr,rr-Max(Abs(i-xk), Abs(j-yk)) -
p[x[k],y[k],i,j]+p[x[k],y[k],xр,ур] + Max(Abs(xр-xk),
Abs(ур-yk))+p[xр,ур,i,j]);
      res:=Min(res,tr); {Min - минимальное из
всех чисел}
    End;
  End;
End;

```

Методические указания

Эта алгоритмическая классика — задачи о расстановке ферзей, обхода конем шахматной доски, поиска выхода из лабиринта и т. п. — решение данной задачи найти не сложно. Ее можно рассматривать как обобщение на идейном уровне и использовать для самостоятельной работы при изучении темы «Перебор и методы его сокращения». Интерпретация же доски в виде графа позволяет установить связь с такими известными алгоритмами поиска кратчайших путей, как алгоритмы Хампстры и Флойда.

Самостоятельные исследования

1. Разработайте программный код, независимый от размеров шахматной доски. Исследование его временных характеристик в очередной раз позволяет убедиться в ограниченных возможностях компьютера.
2. Ограничим себя только двумерными структурами, например матрица p имеет вид: $p[1..64, 1..64]$. Изменение программного кода в этом случае можно рассматривать как упражнение.

«Полигон»

Условие задачи

Существует игра для одного игрока, которая начинается с задания Полигона из n вершинами. Пример графического представления Полигона при $n = 4$ показан на рис. 10.14. Для каждой вершины Полигона задается значение — целое число, а для каждого ребра — метка операции «+» (сложение), либо «*» (умножение).

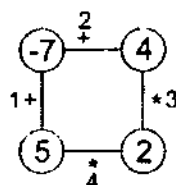


Рис. 10.14.
Графическое изображение Полигона

Ребра Полигона пронумерованы от 1 до n .

Первым ходом в игре удаляется одно из ребер.

Каждый последующий ход состоит из следующих шагов:

- выбирается ребро E и две вершины V_1 и V_2 , которые соединены ребром E ;
- ребро E и вершины V_1 и V_2 заменяются новой вершиной со значением, равным результату выполнения операции, определенной меткой ребра E , над значениями вершин V_1 и V_2 .

Игра заканчивается, когда больше нет ни одного ребра. Результат игры — это число, равное значению оставшейся вершины.

Пример игры. Рассмотрим полигон, приведенный в условии задачи.

Игрок начал игру с удаления ребра 3 (рис. 10.15). После этого игрок выбирает ребро 1 (рис. 10.16). Затем — ребро 4 (рис. 10.17). И, наконец, ребро 2 (рис. 10.18). Результатом игры будет число 0.

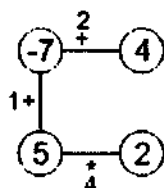


Рис. 10.15. Удаление ребра 3

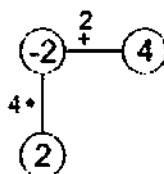


Рис. 10.16. Результат выбора ребра 1

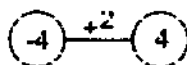


Рис. 10.17. Результат выбора ребра 4



Рис. 10.18. Результат выбора ребра 2

Требуется написать программу, которая по заданному Полигону вычисляет максимальное значение оставшейся вершины и выводит список всех ребер, удаление которых на первом ходе игры позволяет получить это значение.

Входные данные. Файл POLYGON.IN описывает Полигон с n вершинами. Файл содержит две строки. В первой строке записано число n . Вторая строка содержит метки ребер с номерами 1, ..., n , между которыми записаны через один пробел значения вершин (первое из них соответствует вершине, смежной ребрам 1 и 2, следующее — смежной ребрам 2 и 3, и так далее, последнее — смежной ребрам n и 1). Метка ребра — это буква $+$, соответствующая операции $+$, или буква x , соответствующая операции $*$.

Выходные данные. В первой строке выходного файла POLYGON.OUT программа должна записать максимальное значение оставшейся вершины, которое может быть достигнуто для заданного Полигона. Во второй строке должен быть записан список всех ребер, удаление которых на первом ходе позволяет получить это значение. Номера ребер должны записываться в возрастающем порядке и отделяться друг от друга пробелом.

Ограничения: $3 \leq n \leq 50$; для любой последовательности ходов значения вершин находятся в пределах $[-32768, 32767]$.

Пример входного файла:

```

04
14
-7 1 4 x 2 x 5

```

Пример выходного файла:

```

33
1 2

```

Усложнение задачи

Рассмотрим вначале более простой вариант задачи. Пусть значения в вершинах неотрицательны, а на ребрах мы сами выбираем операцию $+$ или $*$. Из значений в вершинах (операндов) и операций составляется арифметическое выражение. Задача сводится к такой расстановке скобок в арифметическом выражении, чтобы результат вычисления этого выражения был минимальным.

Пусть у нас есть строка s , описывающая конкретное выражение. Подзадача — это решение задачи для подстроки с i -го по j -й операнд. Получив один раз результат для подзадачи, мы можем в неизменном виде (без пересчета) использовать его для решения подзадач большей размерности (выражения большей длины). В массиве W мы при этом храним результаты решения подзадач; $W[i, j]$ равно максимальному значению выражения, полученного при решении задачи для подстроки $s[i..j]$. Значение $W[i, i]$ равно i -му операнду. Очевидно, что $W[i, j] = 0$ при $i > j$. Особенность заполнения массива W заключается в том, что он заполняется по диагоналям, параллельным главным: $j-i=0, j-i=1, j-i=2, \dots, j-i=n-1$. Принцип такого заполнения описывается следующими рекуррентными соотношениями:

$$W[i, j] = \begin{cases} s[i] & \text{при } i = j, 1 \leq i, j \leq n; \\ \max(W[i, k] * W[k+1, j], W[i, k] + W[k+1, j]) & \text{при } i < j. \end{cases}$$

ЗАДАЧИ ПЕРВОГО ТУРА

«Цветочный магазин»

Условие задачи

Вы хотите оформить витрину вашего цветочного магазина наилучшим образом. У вас есть f букетов (каждый букет состоит из цветов одного вида, все цветы одного вида собраны в одном букете) и не меньшее количество ваз, расположенных в ряд. Вазы приклеены к полке и пронумерованы слева направо последовательно от 1 до v , где v — количество ваз. Ваза 1 является крайней левой в ряду, а ваза v — крайней правой. Букеты можно помещать в разные вазы, и каждый букет имеет уникальный номер в пределах от 1 до f . Порядок размещения букетов в вазах следующий: букет i всегда содержится в вазе, которая стоит левее вазы с букетом j , если $i < j$. Пусть, например, у вас есть букет азалий с номером 1, букет бегоний с номером 2 и букет гвоздик с номером 3. Все букеты должны быть помещены в вазы. Букет азалий должен быть помещен в вазу, расположенную левее вазы с бегониями, а букет бегоний — левее вазы с гвоздиками. Если ваз будет больше, чем букетов, то некоторые вазы останутся пустыми. В вазе может быть не более одного букета.

Каждая ваза имеет собственные характеристики, так же как и цветы. Поэтому помещенный в вазу букет имеет некоторую эстетическую характеристику, выраженную целым числом. Значения эстетических характеристик приведены в табл. 11.1. Пустая ваза имеет значение эстетической характеристики, равное нулю.

Таблица 11.1

Букеты	Вазы				
	1	2	3	4	5
1 (азалии)	7	23	-5	-24	16
2 (бегонии)	5	21	-4	10	23
3 (гвоздики)	-21	5	-4	-20	20

Согласно этой таблице, азалии прекрасно выглядят во второй вазе и ужасно — в четвертой.

Для достижения наилучшего эстетического оформления витрины вы должны максимизировать сумму значений эстетических характеристик размещений цветов в вазах, сохраняя требуемое упорядочение букетов. Если существует более одного размещения, имеющих максимальную сумму, то вы должны вывести только одно из них.

Ограничения:

$1 \leq f \leq 100$, где f — количество букетов; букеты пронумерованы от 1 до f ;

$1 \leq v \leq 100$, где v — количество ваз;

$-50 \leq A_{ij} \leq 50$, где A_{ij} — значение эстетической характеристики, получаемое при помещении букета i в вазу j .

Входные данные. Входной текстовый файл называется `flower.inp`:

Первая строка содержит два числа: f , v ;

Следующие f строк: каждая из этих строк содержит V целых чисел, A_{ij} является j -м числом в $(i+1)$ -й строке входного файла.

Выходные данные. Выходной файл должен быть текстовым файлом с именем `flower.out` и состоять из двух строк:

Первая строка содержит сумму значений эстетических характеристик вашего размещения цветов в вазах;

Вторая строка содержит последовательность из f чисел, где k -е число определяет номер вазы, в которую помещен букет k .

Пример.

`flower.inp`:

```
5
23 -5 -24 16
21 -4 10 23
-21 5 -4 -20 20
```

`flower.out`:

```
53
2 4 5
```

Проверка. ваша программа должна выполняться не более 2 секунд. Частичные решения в каждом тесте оцениваться не будут.

Ключевые слова:

Ключевая фраза в формулировке задачи — «порядок размещения букетов в вазах следующий: букет i всегда содержится в вазе, которая стоит левее вазы с букетом j , если $i < j$ ». Она гово-

рит об аддитивности задачи и возможности использования динамической схемы для ее решения.

Пусть эстетические характеристики хранятся в матрице b , а максимальные суммы размещения i ($1 \leq i \leq f$) букетов в k ($1 \leq k \leq v$) вазах — в матрице r . Действительно, если требуется найти оптимальное размещение i букетов по k вазам $r[i, k]$ и известны оптимальные решения для задач меньшей размерности — размещения t ($1 \leq t \leq i$) букетов по q ($1 \leq q \leq k-1$) вазам, то $r[i, k] = \text{Max}(r[i, k-1], b[i, k] + r[i-1, k-1])$. В этом рекуррентном соотношении осуществляется выбор максимального значения из стоимости размещения i букетов в $k-1$ вазе и размещения букета i в вазе k , плюс наилучшего размещения $i-1$ букета в $k-1$ вазе. Естественно, что до «запуска в работу» этого рекуррентного соотношения следует сформировать граничные решения: размещения одного букета по всем вазам и размещения i букетов по i вазам.

Ответом на первую часть задания — максимальное значение суммы эстетических характеристик размещений цветов в вазах — является значение $r[f, v]$.

Методическое отступление. При необходимости возможно детальное рассмотрение решения на конкретном примере. Пусть

$$b = \begin{pmatrix} 7 & 23 & -5 & -24 & 16 \\ 23 & 21 & -4 & 10 & 23 \\ -21 & 5 & -4 & -20 & 20 \end{pmatrix} \text{ (из формулировки задачи). Размес-}$$

$$\text{тим один букет азалии: } r = \begin{pmatrix} 7 & 23 & 23 & 23 & 23 \\ * & * & * & * & * \\ * & * & * & * & * \end{pmatrix}. \text{ Очевидно, что}$$

букет следует поставить во вторую вазу, и это размещение — наилучшее. (Символом «*» здесь отмечены несформированные элементы r). При совпадении количества букетов (i) с количеством ваз (k) это размещение — единственное, и эстетические

$$\text{характеристики просто складываются: } r = \begin{pmatrix} 7 & 23 & 23 & 23 & 23 \\ * & 28 & * & * & * \\ * & * & 24 & * & * \end{pmatrix}.$$

Если значение i больше k , то задача не имеет решения — соответствующие элементы r не заполняются.

Далее идет расчет пяти значений:

- $r[2, 3] = \text{Max}(r[2, 2], b[2, 3] + r[1, 2]) = 28;$
- $r[2, 4] = \text{Max}(r[2, 3], b[2, 4] + r[1, 3]) = 33;$
- $r[2, 5] = \text{Max}(r[2, 4], b[2, 5] + r[1, 4]) = 46;$

- $r[3, 4] = \text{Max}(r[3, 3], b[3, 4] + r[2, 3]) = 24$;
- $r[3, 5] = \text{Max}(r[3, 4], b[3, 5] + r[2, 4]) = 53$.

Каждое вычисление сопровождается рассуждениями типа (для первого): если второй букет не ставится во вторую вазу, то наилучшее размещение остается прежним — $r[2, 2]$. При постановке второго букета в третью вазу следует взять значение $r[1, 2]$ — стоимость наилучшего размещения одного букета в двух вазах плюс эстетическую характеристику размещения второго букета в третьей вазе — $b[2, 3]$.

В итоге матрица r имеет вид: $r = \begin{pmatrix} 7 & 23 & 23 & 23 & 23 \\ * & 28 & 28 & 33 & 46 \\ * & * & 24 & 24 & 53 \end{pmatrix}$.

Значение $r[3, 5] = 53$ является ответом задачи.

Для вывода размещения букетов по вазам достаточно информации, содержащейся в r , но можно (для упрощения) использовать и дополнительные структуры данных.

Начинаем с последней строки (последнего букета). Если максимальный элемент единственный, то номер столбца соответствует номеру вазы. Так, в примере третий букет размещается в третьей вазе. Переходим к предыдущей строке и находим максимальный элемент, но не во всех столбцах, а только в не задействованных ранее. Если максимальных элементов несколько, то выбирается «самый левый», и номер соответствующего столбца является номером вазы. Так, в первой строке примера максимальный элемент находится во втором и третьем столбцах (четвертый и пятый не рассматриваются) — тогда мы выбираем второй столбец, ибо он первый дал это значение максимума.

Методические рекомендации

Эта задача в «идейном» плане является традиционной по теме «Динамическое программирование» с явной схемой аддитивности. Ее можно использовать как при объяснении темы (сократив формулировку), так и в качестве задания для самостоятельной работы.

Другой вариант разбора этой задачи можно найти в книге М. С. Долинского*.

Самостоятельные исследования

Если снять ограничение в формулировке задачи — «порядок размещения букетов в вазах следующий: букет i всегда содер-

* Долинский М. С. Алгоритмизация и программирование в TURBO PASCAL. От простых до олимпиадных задач. СПб.: Питер, 2005. С. 132–137.

жится в вазе, которая стоит левее вазы с букетом j , если $i < j$, то она превращается в задачу на перебор вариантов. Сравните временные характеристики решений и в очередной раз убедитесь в ограниченности возможностей вашего компьютера.

«Скрытые коды»

Условие задачи

Дано множество кодовых слов и текст. Предложенный текст содержит сообщение, созданное из кодовых слов, включенных в текст некоторым, возможно неоднозначным, способом.

Кодовые слова и текст содержат только большие и малые (прописные и строчные) буквы английского алфавита. Большая и малая буквы считаются различными. Длина кодового слова определяется обычным способом: например, длина кодового слова ALL равна 3.

Буквы кодового слова встречаются в данном тексте не обязательно последовательно. Например, кодовое слово ALL будет всегда присутствовать в тексте, содержащем подпоследовательность вида $AuLvL$, где u и v являются последовательностями букв (возможно, пустыми). Будем говорить, что $AuLvL$ является *покрывающей последовательностью* для ALL. В общем случае покрывающая последовательность для кодового слова определяется как подпоследовательность текста, в которой первая и последняя буквы совпадают, соответственно, с первой и последней буквами кодового слова и существует способ восстановить кодовое слово удалением, если необходимо, некоторых букв подпоследовательности. Заметим, что кодовое слово может содержаться в одной или нескольких покрывающих последовательностях или может не встречаться в тексте вообще. Кроме того, покрывающая последовательность может являться покрывающей последовательностью для более чем одного кодового слова.

Покрывающая последовательность определяется своей начальной и конечной позициями в тексте (позициями первой и последней букв, соответственно). Первая буква текста находится в позиции 1. Будем говорить, что покрывающие последовательности s_1 и s_2 не перекрываются, если начальная позиция s_1 больше ($>$), чем конечная позиция s_2 , или наоборот. В противном случае назовем эти две последовательности *перекрывающимися*.

Решение задачи заключается в извлечении скрытого сообщения из текста. Решение есть множество элементов, каждый из которых состоит из кодового слова и покрывающей последовательности для этого кодового слова. При этом должны соблюдаться следующие условия:

- покрывающие последовательности не перекрывают друг друга;
- длина покрывающей последовательности не превышает 1000;
- сумма длин кодовых слов максимальна (каждый элемент добавляет длину кодового слова, которое он содержит, к сумме длин).

При наличии более одного решения вам необходимо указать только одно из них.

Ограничения:

- $1 \leq n \leq 100$, где n — количество кодовых слов;
- максимальная длина кодового слова — 100 букв;
- $1 \leq$ длина заданного текста $\leq 1\,000\,000$ букв.

Будем говорить, что покрывающая последовательность s для кодового слова w является *минимальной справа*, если никакой собственный префикс s (собственным префиксом s является любая начальная подпоследовательность s , такая, что она не равна s) не является покрывающей последовательностью для w . Например, для кодового слова ALL, AAALAL является *минимальной справа* покрывающей последовательностью, в то время как AAALALAL также является покрывающей последовательностью, но не является минимальной справа.

Гарантируется, что в данном тексте:

- для любой позиции в тексте количество минимальных справа покрывающих последовательностей, содержащих эту позицию, не превышает 2500;
- количество минимальных справа покрывающих последовательностей не превышает 10 000.

Входные данные. Имеется два входных текстовых файла: words.inp и text.inp. Файл words.inp содержит список кодовых слов, а файл text.inp содержит текст.

- Первая строка файла words.inp содержит значение n . Каждая из последующих n строк содержит кодовое слово, которое является последовательностью букв без пробелов между ними. Каждое кодовое слово однозначно определено своим порядковым номером в файле words.inp: целые числа от 1 до n являются номерами кодовых слов.
- Файл text.inp состоит из последовательности букв, заканчивающейся одним символом конца строки (*end-of-line character*), за которым следует символ конца файла (*the end-of-file symbol*). Этот файл не содержит пробелов.

Рекомендации для программирующих на языке Pascal. Для обеспечения эффективности рекомендуется объявлять входные файлы типа text.

Выходные данные. Вывод должен производиться в текстовый файл `codes.out`:

- первая строка должна содержать сумму, которую вычислила ваша программа;
- каждая следующая строка должна содержать один элемент вашего решения. Строка состоит из трех целых чисел i, s, e , где i — номер кодового слова, которое содержится в покрывающей последовательности, заданной начальной позицией s и конечной позицией e . Порядок вывода всех строк, начиная со второй, не имеет значения.

Пример

`words.inp:`

4

RuN

RaBbit

HoBbit

StoP

`text.inp:`

StXRuYNvRuHoABbvizXztNwRRuNNP

`codes.out:`

12

2 9 21

1 4 7

1 24 28

Примечание. Скрытое сообщение, которое соответствует приведенному примеру, выглядит так: «RuN RaBbit RuN». (Другим возможным сообщением является «RuN HoBbit RuN»). Обратите внимание, что сообщение не должно появляться в выходном файле.

Проверка. Ваша программа должна выполняться не более 10 секунд. Частичные решения в каждом тесте оцениваться не будут.

Обсуждение задачи

Для наглядности обсуждения «привяжем» его к примеру из формулировки задачи.

Задача логически разбивается на две части:

- выделение из текста минимальных справа покрывающих последовательностей;
- нахождение неперекрывающейся цепочки последовательностей с максимальным суммарным весом, под которым понимается сумма длин кодовых слов, содержащихся в последовательностях.

Рассмотрим первую подзадачу. Пронумеруем символы исходного текста (только для обсуждения) так, как показано в табл. 11.2.

Таблица 11.2

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	t	x	R	u	Y	N	v	R	u	H	o	a	B	b
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
y	i	z	X	z	t	N	w	R	R	u	u	N	N	P

При вводе текста следует создать 52 (количество букв английского алфавита) динамических массива. Их реализация, в зависимости от используемой системы программирования, может быть различна (например — как 52 линейных списка). Структура массивов для нашего примера приведена в табл. 11.3. Первый элемент массива определяет количество вхождений символа в текст, а затем идут позиции, на которых он встречается в тексте.

Таблица 11.3

Символ	Количество	Позиции в тексте			
S	1	1			
t	22	2	21		
X	2	3	19		
R	4	4	9	24	25
u	4	5	10	26	27
Y	1	6			
N	4	7	22	28	29
v	2	8	16		
H	1	11			
o	1	12			
a	1	13			
B	1	14			
b	1	15			
i	1	17			
z	2	18	20		
w	1	23			
P	1	30			

Примечание. Очередность символов здесь дана, для удобства, по исходному тексту. Фактический их порядок следования определяется их кодировкой по стандарту.

Такое представление текста превращает выделение минимальных справа покрывающих последовательностей для каждого кодового слова в чисто техническую задачу. Для каждого кодового слова осуществляется просмотр соответствующих массивов (списков), чтобы номер позиции символа в тексте был больше номера предыдущего символа слова. Описание этих последовательностей следует сохранить — массив $f[1..10000, 1..3]$, структура которого представлена в табл. 11.4. «Вес» последовательности мы определим как длину входящего в нее кодового слова.

Таблица 11.4

Номер покрывающей последовательности	Номер кодового слова	Начальная позиция в тексте	Конечная позиция в тексте	Вес покрывающей последовательности
1	1	4	7	3
2	1	9	22	3
3	1	24	28	3
4	1	25	28	3
5	2	4	21	6
6	2	9	21	6
7	3	11	21	6
8	4	1	30	3

Примечание. Обратите внимание на то, что данные в табл. 11.4 (столбцы 2, 3, 4) представлены в соответствии с требованиями к выводу результата в условии задачи.

Методическое отступление. Соревнование на лучший программный код процедуры выделения покрывающих последовательностей украсит занятие.

Перейдем теперь ко второй подзадаче. Исходные данные для ее решения зафиксированы в массиве f (табл. 11.4). Выполним сортировку (по неубыванию) элементов массива по значению конечной позиции последовательности в тексте. Сам массив мы при этом «трогать» не будем, а зафиксируем номера отсортированных элементов в массиве v . Для нашего примера он имеет вид: $v = (1, 5, 6, 7, 2, 3, 4, 8)$.

Что нам требуется сделать? Иллюстрация дана на рис. 11.1. Итак, мы берем очередную последовательность с номером i (из

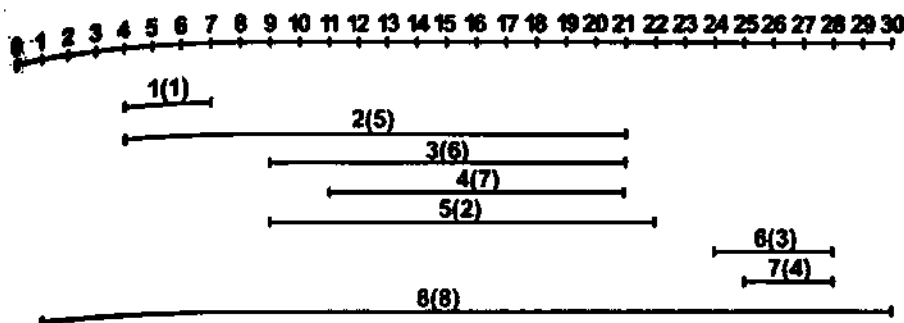


Рис. 11.1. Покрывающие последовательности отсортированы по значению правой границы. Число в круглых скобках указывает на номер последовательности в табл. 11.4

массива v), с учетом условия непересечения последовательностей просматриваем все ранее построенные цепочки, расположенные левее текущей, и выбираем из них ту, которая имеет максимальный вес. Это ни что иное, как вариант «жадной» логики. При этом для хранения суммарных длин последовательностей (суммы весов) необходим массив (d), а для хранения номеров последовательностей, входящих в цепочку, — массив (p). Изменение значений элементов массивов в процессе обработки последовательностей приведено в табл. 11.5.

Таблица 11.5

Номер элемента массива v (значение i), номер покрывающей последовательности (значение $v[i]$)	Массив d	Массив p
1, 1	(3, 0, 0, 0, 0, 0, 0, 0)	(0, 0, 0, 0, 0, 0, 0, 0)
2, 5	(3, 0, 0, 0, 6, 0, 0, 0)	(0, 0, 0, 0, 0, 0, 0, 0)
3, 6	(3, 0, 0, 0, 6, 9, 0, 0)	(0, 0, 0, 0, 0, 1, 0, 0)
4, 7	(3, 0, 0, 0, 6, 9, 9, 0)	(0, 0, 0, 0, 0, 1, 1, 0)
5, 2	(3, 6, 0, 0, 6, 9, 9, 0)	(0, 1, 0, 0, 0, 1, 1, 0)
6, 3	(3, 6, 12, 0, 6, 9, 9, 0)	(0, 1, 6, 0, 0, 1, 1, 0)
7, 4	(3, 6, 12, 12, 6, 9, 9, 0)	(0, 1, 6, 6, 0, 1, 1, 0)
8, 8	(3, 6, 12, 12, 6, 9, 9, 4)	(0, 1, 6, 6, 0, 1, 1, 0)

Для вывода результата требуется найти один максимальный элемент в массиве d , а затем, используя данные из массива p , вывести характеристики покрывающих последовательностей.

Методическое отступление. Другой вариант объяснения подзадачи возможен с использованием «наброска» основной процедуры решения:

Procedure Solve;

Var i, j, Best, q: Integer;

Begin

 <Элементам массива *p* присвоить значение "нуль">;
 {Массив *p* используется для организации цепочек
 непересекающихся последовательностей}

For *i*:=1 **To** *m* **Do Begin** {*m* — количество покрывающих
 последовательностей, ненулевых элементов массива *f*}

 {Поиск наилучшей цепочки покрывающих
 последовательностей, расположенных "слева"
 от текущей последовательности с номером *i*}

Best:=0; *j*:=1;

While (*j*<*i*) **And** (*f*[*v*[*j*],3]<*f*[*v*[*i*],2]) **Do Begin**
 {Конечная позиция цепочки меньше начальной позиции
 текущей цепочки с номером *i*}

If (*Best*=0) **Or** (*d*[*v*[*j*]]>*d*[*v*[*Best*]]) **Then** *Best*:=*j*;
 {Если длина цепочки, заканчивающейся последовательностью
 с номером *j*, больше ранее найденной, то считаем
 ее лучшей}

j:=*j*+1;

End;

d[*v*[*i*]]:=**Length**(*w*[*f*[*v*[*i*],1]]); {В массиве *w*
 храним кодовые слова}

If *Best*<>0 **Then Begin**

d[*v*[*i*]]:=*d*[*v*[*i*]]+*d*[*v*[*Best*]]; {Увеличиваем
 вес цепочки, заканчивающейся последовательностью
 с номером *i*}

p[*v*[*i*]]:=*v*[*Best*]; {Запоминаем, к какой
 последовательности мы должны идти от текущей}

End;

End;

End;

Методические указания

Для полного решения задачи требуется значительный объем оперативной памяти. Возможности среды *Turbo-Pascal* в этом плане ограничены. Необходимо использовать или динамическую память («кучу») и работать со списками, или другую, более современную, систему программирования.

После того, как исходный (входной) текст структурирован (табл. 11.3), задача сводится к модификации известных мето-

дов поиска подстроки в строке и к «классике» информатики под названием «жадные алгоритмы».

Самостоятельные исследования

Разработка системы тестов для предельных случаев из формулировки задачи и оценка временных характеристик работы программы даст возможность построить зависимость времени выполнения полиномиальных алгоритмов.

«Подземный город»

Условие задачи

Вы оказались в одном из подземных городов Каппадокии. Будучи в темноте, вы случайно обнаруживаете карту города. К сожалению, на карте не отмечено то место, где вы находитесь. Ваша задача — исследуя город, определить это место.

Карта города представляет собой прямоугольную сетку с единичными квадратами. Каждый квадрат может быть либо пустым, отмеченным буквой «O», либо частью стены, отмеченной буквой «W». На карте также указано направление на север. К счастью, у вас есть в руке компас, так что вы можете правильно ориентироваться по карте. Первоначально вы находитесь на пустом квадрате.

Исследование города начинается с вызова процедуры (или функции) `start`, не имеющей аргументов. При исследовании города вы можете использовать процедуры (или функции) `look` и `move`.

Вы можете задавать вопросы, вызывая функцию с именем `look(dir)`, где `dir` обозначает направление исследования. Оно может иметь значения «N», «S», «E» или «W», обозначающие, соответственно, север, юг, восток и запад. Пусть аргумент `dir` имеет значение «N». Тогда ответом будет буква «O», если квадрат, который вы анализируете в северном направлении, является пустым. Если там стена, то ответом является буква «W». Аналогично можно анализировать и получить информацию о других соседних квадратах.

Вы можете перемещаться в один из четырех соседних квадратов, вызывая процедуру `move(dir)`, где `dir` обозначает направление вашего движения. Вы можете перемещаться только в пустые квадраты. Перемещение на клетку, занятую стеной, является ошибкой. Из любого пустого квадрата можно попасть в любой другой пустой квадрат.

Вам необходимо найти координаты пустого квадрата, в котором вы обнаружили карту, вызывая процедуру `look(dir)` минимальное число раз. Когда вы, наконец, определите эти координаты,

наты, вы должны вызвать процедуру (функцию) $\text{finish}(x, y)$, где x — горизонтальная (восток-запад), а y — вертикальная (юг-север) координаты этого пустого квадрата.

Ограничения:

- $3 \leq U \leq 100$, где U есть ширина карты, то есть количество квадратов на карте в горизонтальном (запад-восток) направлении;
- $3 \leq V \leq 100$, где V есть высота карты, то есть количество квадратов карты в вертикальном (юг-север) направлении;
- город окружен стенами, которые обозначены на карте;
- юго-западный угол города имеет координаты $(1,1)$, а северо-восточный угол имеет координаты (U, V) .

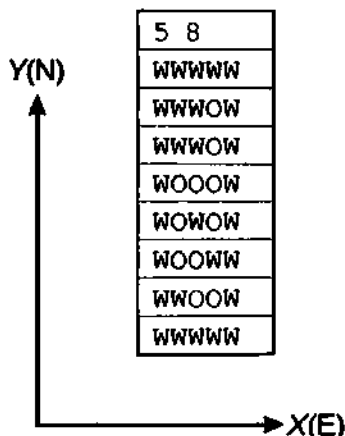
Входные данные. Входным файлом является текстовый файл с именем `under.inp`:

- первая строка содержит два числа: U, V ;
- каждая из следующих V строк содержит информацию о строках карты. Строка состоит из U значений. При этом x -й символ строки $(V-y+2)$ входного файла задает координаты позиции (x, y) на карте: это либо буква «W», обозначающая стену, либо буква «O», обозначающая пустой квадрат. Строки не могут содержать пробелы.

Выходные данные. Выходной файл не должен создаваться. Результат решения задачи сообщается вызовом процедуры (функции) $\text{finish}(x, y)$.

Пример:

`under.inp`:



Возможная последовательность вызовов процедур, заканчивающаяся корректным вызовом процедуры finish:

Interaction:	
Start()	
Look('N')	'W'
Look('E')	'O'
Move('E')	
Look('E')	'W'
Finish(3,5)	

Инструкция для программирующих на языке Pascal. В исходный файл вашей программы включите следующую строку: `uses undercpu;` Этот `cpu` будет содержать следующее:

```

procedure start; {must be called first}
function look (dir:char):char;
procedure move (dir:char);
procedure finish (x,y:integer); {must be called last}
end

```

Инструкция для программирующих на языке C/C++. В исходном файле вашей программы нужна следующая строка: `#include "under.h"`. Там будут содержаться следующие объявления:

```

void start (void); /* must be called first */
char look (char);
void move (char);
void finish (int,int); /* must be called last */

```

Также создайте проект с именем `under`, который должен включать вашу программу и библиотеку `underobj.obj`. Для того, чтобы это сделать, вам нужно использовать `project` меню IDE и выбрать `open`, чтобы создать проект, а затем использовать `add item`, чтобы включить ваш исходный файл (`under.c` или `under.cpp`) и файл `underobj.obj`. Используйте `LARGE memory model compiler option`. (Будьте внимательны: этот режим перекрывает режим, упомянутый в правилах соревнования.)

Проверка. Ваша программа должна выполняться не более 5 секунд. Чтобы получить полный балл А за тест, нужно, чтобы количество вызовов функции `look`, было меньше или равно числу M , установленному тестирующей программой. Заме-

тим, что M выбирается таким, чтобы быть больше ($>$), чем минимум. В частности, M не зависит от того, выбрано ли направление осмотра по или против часовой стрелки. Вы можете получить часть баллов, если число обращений к функции `look` больше ($>$) M , но меньше ($<$) удвоенного M . Баллы, которые вы получите, вычисляются путем округления первого десятичного знака того значения, которое определяется по следующей формуле:

$$\begin{cases} A \text{ if } x \leq M; \\ A(2M - x) / M \text{ if } M < x < 2M; \\ 0 \text{ if } x \geq 2M. \end{cases}$$

Если программа работает с нарушением правил, то она оценивается в 0 баллов. К нарушениям правил относятся следующие:

- обращение к библиотечной процедуре (функции) с недопустимым аргументом, например с символом, который не обозначает направление;
- попытка двигаться на стену;
- несоответствие условиям задачи.

Как проверить работу вашей программы? Создайте текстовый файл с именем `place.txt`, включающий координаты квадрата, в котором находится карта. Запустите вашу программу. Просмотрите результат в файле `result.txt`.

Файл `place.txt` должен содержать одну строку, включающую горизонтальную и вертикальную координаты клетки, в которой находится карта. Вам надо создать свой собственный входной файл `under.inp`. Файл `result.txt` будет содержать две строки. Первая строка будет содержать параметры x и y вызова процедуры `finish(x, y)`. Вторая строка будет содержать фразу «*You used look nnn times*». Следует заметить, что эта возможность предназначена для того, чтобы проверить совместимость вашей программы с библиотекой. Это никак не свидетельствует о правильности вашего решения.

Идея решения*

Задача является очередной вариацией на тему «лабиринта».

Классические идеи, основанные на обходе в глубину:

- из каждой клетки выбирается еще не исследованный путь;
- если из текущей клетки просмотрены все пути или она тупиковая, то возвращаемся на шаг назад,

или на обходе в ширину:

* Авторы благодарят И. В. Матанцева за помощь в разборе задачи.

- берем клетку (помеченную, но еще не обработанную) из очереди;
- просматриваем все соседние с ней свободные клетки, заносим их в очередь и присваиваем значение метки на единицу большее, чем значение метки текущей клетки,

Оба эти варианта имеют временную характеристику $O(n^2)$, где n — количество клеток лабиринта (в нашем случае максимальное значение n равно 40000).

Используя классические методы, в принципе, можно решить эту задачу: из множества «кандидатов» (а оно первоначально равно множеству всех свободных клеток) путем последовательного, согласно обозначенным идеям, опроса соседних клеток исключать клетки до тех пор, пока не останется одна-единственная клетка. Однако, анализируя правила проверки задачи, можно с полной уверенностью предположить, что реализация этих идей в лучшем случае даст минимальное количество баллов. Поэтому, оставаясь в рамках общей схемы (последовательное уменьшение множества «кандидатов»), следует искать другие принципы ее реализации или, что то же самое, следует определить эвристики (эвристику), минимизирующие количество использований процедуры `look(dir)`.

Обратимся к конкретному примеру. Пусть дан лабиринт, показанный на рис. 11.2, с начальным положением (4,6). Оно нам не известно, и его требуется определить.

Схема анализа множества «кандидатов» от начального расположения наблюдателя представлена в табл. 11.6. Здесь мы, по сути, анализируем множество «кандидатов» (соседних клеток) не для отдельной клетки, а для некоей построенной на предыдущих шагах области. Для каждой клетки области

(количество таких областей в поле лабиринта мы можем определить) мы вычисляем количество соседствующих с ней стен (это можно сделать без использования процедуры `look(dir)`, но поле лабиринта у нас есть, следует только просмотреть его) и суммируем по всем возможным областям. Затем, на основании некой эвристики, мы определяем клетку области, из которой выполняем процедуру `look(dir)`. Образно говоря, «волна» здесь идет не из одной текущей клетки, как в классических алгоритмах, а от построенной области, и отыскиваются в ней не просто соседние и свободные с областью клетки, а наоборот, задатые! Мы как бы стремимся в тупик лабиринта, чтобы максимально быстро сократить множество «кандидатов» на начальное положение наблюдателя.

	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								
7								
8								

Рис. 11.2. Пример лабиринта

Впрочем, обратимся к табл. 11.6.

Таблица 11.6

Номер шага	Множество S	Состояние обхода	Клетка, из которой задается вопрос, и ответ на него	Измененное множество S																																				
1	(2,2)(2,4)(2,5)(2,7) (3,2)(3,4)(3,5)(3,7) (4,2)(4,3)(4,5)(4,6) (4,7)(5,2)(5,5)(5,6) (6,2)(6,3)(6,5)(7,2) (7,3)(7,4)(7,5)(7,6) (7,7) Мощность множества — 25	<table><tr><td></td><td>-2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr><tr><td>-2</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-1</td><td></td><td></td><td>9</td><td></td><td></td></tr><tr><td>0</td><td></td><td>13</td><td></td><td>13</td><td></td></tr><tr><td>1</td><td></td><td></td><td>9</td><td></td><td></td></tr><tr><td>2</td><td></td><td></td><td></td><td></td><td></td></tr></table>		-2	-1	0	1	2	-2						-1			9			0		13		13		1			9			2						(0,0) слева — свободно	(2,5)(3,5)(4,3) (4,6)(4,7)(5,6) (6,3)(7,3)(7,4) (7,5)(7,6)(7,7)
	-2	-1	0	1	2																																			
-2																																								
-1			9																																					
0		13		13																																				
1			9																																					
2																																								
2	(2,5)(3,5)(4,3) (4,6)(4,7)(5,6) (6,3)(7,3)(7,4) (7,5)(7,6)(7,7) Мощность множества — 12	<table><tr><td></td><td>-2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr><tr><td>-2</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-1</td><td></td><td>3</td><td>7</td><td></td><td></td></tr><tr><td>0</td><td>7</td><td></td><td></td><td>6</td><td></td></tr><tr><td>1</td><td></td><td>6</td><td>7</td><td></td><td></td></tr><tr><td>2</td><td></td><td></td><td></td><td></td><td></td></tr></table>		-2	-1	0	1	2	-2						-1		3	7			0	7			6		1		6	7			2						(0,-1) снизу — свободно	(2,5)(4,3)(4,6) (4,7)(5,6)(6,3)
	-2	-1	0	1	2																																			
-2																																								
-1		3	7																																					
0	7			6																																				
1		6	7																																					
2																																								
3	(2,5)(4,3)(4,6) (4,7)(5,6)(6,3) Мощность множества — 6	<table><tr><td></td><td>-2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr><tr><td>-2</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-1</td><td></td><td>2</td><td>4</td><td></td><td></td></tr><tr><td>0</td><td>5</td><td></td><td></td><td>5</td><td></td></tr><tr><td>1</td><td>5</td><td></td><td>3</td><td></td><td></td></tr><tr><td>2</td><td></td><td>3</td><td></td><td></td><td></td></tr></table>		-2	-1	0	1	2	-2						-1		2	4			0	5			5		1	5		3			2		3				(1,-1) справа — свободно	(2,5)(4,6)(6,3)
	-2	-1	0	1	2																																			
-2																																								
-1		2	4																																					
0	5			5																																				
1	5		3																																					
2		3																																						
4	(2,5)(4,6)(6,3) Мощность множества — 3	<table><tr><td></td><td>-2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr><tr><td>-2</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-1</td><td></td><td>1</td><td>3</td><td></td><td></td></tr><tr><td>0</td><td>3</td><td></td><td></td><td>1</td><td></td></tr><tr><td>1</td><td>3</td><td></td><td></td><td>2</td><td></td></tr><tr><td>2</td><td></td><td>2</td><td>2</td><td></td><td></td></tr></table>		-2	-1	0	1	2	-2						-1		1	3			0	3			1		1	3			2		2		2	2			(1,0) справа — занято	(2,5)(4,6)
	-2	-1	0	1	2																																			
-2																																								
-1		1	3																																					
0	3			1																																				
1	3			2																																				
2		2	2																																					
5	(2,5)(4,6) Мощность множества — 2	<table><tr><td></td><td>-2</td><td>-1</td><td>0</td><td>1</td><td>2</td></tr><tr><td>-2</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>-1</td><td></td><td>1</td><td></td><td></td><td></td></tr><tr><td>0</td><td></td><td></td><td></td><td>1</td><td></td></tr><tr><td>1</td><td></td><td></td><td></td><td></td><td></td></tr><tr><td>2</td><td></td><td>1</td><td>1</td><td></td><td></td></tr></table>		-2	-1	0	1	2	-2						-1		1				0				1		1						2		1	1			(0,-1) сверху — свободно	(4,6)
	-2	-1	0	1	2																																			
-2																																								
-1		1																																						
0				1																																				
1																																								
2		1	1																																					

На первом шаге область состоит только из одной клетки; ее относительные координаты $(0,0)$. Таких областей в поле лабиринта 25. Определяем для каждой области наличие стен слева, справа, снизу и сверху и суммируем по всем областям. В нашем примере сверху и снизу по 9 стен, а слева и справа по 13. Выберем соседнюю для области клетку. Критерием выбора (эвристикой) является минимум разности между $|S| \text{ Div } 2$ и количеством стен в этой клетке. Эта клетка имеет относительные координаты $(0, -1)$. Она находится слева от нашего начального состояния (естественно, что если эта ветвь перебора не приведет к результату, то следует взять следующую клетку). Выполняем процедуру $\text{look}(\text{dir})$. На основании ответа (слева — свободно) исключаем из S все клетки, для которых слева находится стена. Область после таких действий состоит из двух клеток (горизонтальной полосы).

На втором шаге мы вновь как бы «накладываем» построенную область на каждую клетку из S , определяем количество стен в соседних клетках при каждом наложении и эти значения суммируем по всем наложениям. А затем, на основании критерия выбора, выбираем клетку и направление для выполнения процедуры $\text{look}(\text{dir})$. Если эта клетка не совпадает с той клеткой, в которой находится предполагаемый наблюдатель, то осуществляется требуемое перемещение. Для нашего примера эта клетка имеет координаты $(0, -1)$, а направление — вниз. Выполняем процедуру $\text{look}(\text{dir})$, получаем ответ — «свободно». Убираем из множества все клетки, для которых не выполняется условие: клетка снизу от клетки с координатами $(x+0, y-1)$ свободна, где (x, y) — координаты точки из S . Добавляем клетку с координатами $(x+1, y-1)$ в область.

На третьем шаге повторяем те же действия, что и в предыдущих шагах. После подсчета стен и применения критерия выбора процедура $\text{look}(\text{dir})$ выполняется для клетки с координатами $(x+1, y-1)$ и направлением «вправо». Ответом является значение «не занято». Удаляем из S все клетки, для которых не выполняется условие: клетка правее $(x+1, y-1)$ свободна, а область увеличиваем на одну клетку — она становится квадратом.

На четвертом шаге повторяем те же действия. После подсчета стен и применения критерия выбора процедура $\text{look}(\text{dir})$ выполняется для клетки $(x+1, y+0)$ и направления вправо. Ответ — «занято». Область не изменяется, а из S мы исключаем всех «кандидатов», для которых не выполняется найденное условие — справа от клетки $(x+1, y+0)$ находится занятая клетка.

На пятом шаге мы повторяем эти действия, и в итоге в S остается только одна клетка. Ее координаты и являются ответом задачи.

Обратим внимание на клетки, выделенные темным цветом в табл. 11.6 на пятом шаге. Количество стен совпадает с мощностью множества S . Это говорит о том, что для всех возможных конфигураций, находящихся в лабиринте и определяемых построенной областью, в темных клетках находится стена. Их следует «подключить» к строящейся области и пометить как занятые. Кроме того, если окажется, что в какой-то из соседних с областью клеток количество стен равно нулю, то эта клетка свободна, ее следует присоединить к области и пересчитать стены уже у новой области.

Примечание. На каждом шаге мощность множества S уменьшается примерно в два раза, следовательно, за грубую оценку временной сложности метода допустимо принять зависимость $O(n \cdot \log_2 n)$.

Методические указания

Если собрать в единое целое все задачи по теме «Лабиринт», то одно только их решение и исследование позволит познать многие фундаментальные положения информатики. Работа с данной задачей требует значительного «обрамления»: библиотеки, тестов, оценивающей программы, — но их разработка окупится возможностью создать на занятии творческую среду.

Самостоятельные исследования

В обсуждении задачи была рассмотрена только одна из эвристик. Их дальнейший поиск, экспериментальная проверка и сравнение — тема для творческой работы школьника.

ЗАДАЧИ ВТОРОГО ТУРА

«Светофоры»

Условие задачи

Дорожное движение в городе Дингилвилле устроено необычным образом. В городе есть перекрестки и дороги, которыми перекрестки связаны между собой. Любые два перекрестка могут быть связаны не более чем одной дорогой. Не существует дорог, соединяющих один и тот же перекресток с самим собой. Время проезда по дороге в обоих направлениях одинаково. На каждом перекрестке находится один светофор, который в каждый момент времени может быть либо голубым, либо красным.

Цвет каждого светофора изменяется периодически: в течение некоторого интервала времени он голубой, а затем, в течение некоторого другого интервала — красный. Движение по дороге между любыми двумя перекрестками разрешено тогда и только тогда, когда светофоры на обоих перекрестках этой дороги имеют один и тот же цвет в момент въезда на эту дорогу. Если транспортное средство прибывает на перекресток в момент переключения светофора, то его поведение будет определяться новым светом светофора. Транспортные средства могут находиться в состоянии ожидания на перекрестках. У вас есть карта города, которая показывает:

- время прохождения каждой дороги (целые числа);
- длительность горения каждого цвета для каждого светофора (целые числа);
- начальный цвет и оставшееся время горения этого цвета (целые числа) для каждого светофора.

Вы должны определить путь между двумя заданными перекрестками, позволяющий транспортному средству проехать от начального перекрестка к конечному за минимальное время с момента старта. Если существует несколько таких путей, то вы должны вывести только один из них.

Ограничения:

- $2 \leq n \leq 300$, где n — количество перекрестков; перекрестки нумеруются целыми числами от 1 до n ;
- $1 \leq m \leq 14000$, где m — количество дорог;
- $1 \leq l_{ij} \leq 100$, где l_{ij} — время, необходимое для перемещения от перекрестка i до перекрестка j , используя дорогу, которая соединяет перекрестки i и j ;
- $1 \leq t_{ic} \leq 100$, где t_{ic} — длительность горения цвета c для светофора на перекрестке i . Индекс c принимает значение «В» для голубого цвета и «Р» — для красного («В» и «Р» — большие латинские буквы);
- $1 \leq r_{ic} \leq t_{ic}$ где r_{ic} — оставшееся время свечения начального цвета c на перекрестке i .

Входные данные. Входной файл является текстовым файлом с именем `lights.inp`:

- первая строка содержит два числа: номер начального перекрестка и номер конечного перекрестка;
- вторая строка содержит два числа: n , m ;
- следующие n строк содержат информацию об n перекрестках. $(i+2)$ -я строка входного файла содержит информацию о перекрестке i : C_i , r_{ic} , t_{iB} , t_{iP} , где C_i («В» или «Р») — начальный цвет светофора на перекрестке i ;

- следующие m строк описывают m дорог. Каждая строка имеет вид: i, j, l_{ij} , где i и j — номера перекрестков, которые связывает эта дорога.

Выходные данные. Выходной текстовый файл должен иметь имя `lights.out`.

Если путь существует:

- первая строка должна содержать минимальное время, необходимое для достижения конечного перекрестка из начального;
- вторая строка должна содержать список перекрестков, который соответствует найденному минимальному пути.

Вы должны записать перекрестки в порядке их посещения. Первое целое число должно быть номером начального перекрестка, последнее — конечного.

Если путь не существует, то единственная строка должна содержать только целое число 0.

Пример.

`lights.inp:`

```
1 4
4 5
В 2 16 99
Р 6 32 13
Р 2 87 4
Р 38 96 49
1 2 4
1 3 40
2 3 75
2 4 76
3 4 77
```

`lights.out:`

```
127
1 2 4
```

Проверка. Ваша программа должна выполняться не более 2 секунд. Частичные решения в каждом тесте оцениваться не будут.

Обсуждение задачи

Если из формулировки задачи убрать условие: «движение по дороге между любыми двумя перекрестками разрешено тогда и только тогда, когда светофоры на обоих перекрестках этой дороги имеют один и тот же цвет в момент въезда на эту дорогу», то задача сводится к реализации известного алгоритма Дейкстры нахождения кратчайшего пути между вершинами графа.

Методическое отступление. Разбор задачи можно начать с повторения алгоритма Е. Дейкстры. С этой целью приводится один из возможных способов реализации и ставится вопрос об оптимизации кода (например, в приведенном тексте процедуры Solve требуется убрать лишние переменные). Разумеется, такая работа в некоторых случаях требует трассировки алгоритма для простых примеров входных данных.

Procedure Solve(v:Integer); {D — массив кратчайших расстояний (минимальных времен). Path — массив для хранения номера вершины, из которой мы попадаем в текущую. Используется для вывода пути с помощью простой процедуры:

Procedure Print(v:Integer);

Begin

If Path[v]<>0 **Then** Print(Path[v]);

Write(v, ' ');

End; }

{A — матрица смежности графа. Переменные D, Path, A являются глобальными}

Const max=32765;

Var Nnew:Array[1..NMax] **Of** Boolean;

R: Array[1..NMax] **Of** Integer;

i,j,s,p,min:Integer;

Begin

For i:=1 **To** n **Do** **Begin** Nnew[i]:=True; D[i]:=max; **End;**

{Начальные присвоения}

Nnew[v]:=False; {Начинаем с вершины v}

D[v] := 0;

s := v;

For i:= 2 **To** n **Do** **Begin** {Требуется n-1 итерация}

min:=max;

For j := 1 **To** n **Do**

If Nnew[j] **Then** **Begin** {Оценка для вершины не получена}

If A[s,j]<>0 **Then** R[j] := D[s]+A[s,j]

Else R[j]:=max;

If min>R[j] **Then** **Begin** min := R[j]; p:= j; **End;**

{Одновременно с вычислением оценки находим вершину, для которой она минимальна}

End;

For j := 1 **To** n **Do**

If D[j]>R[j] **Then** **Begin** {Корректируем расстояния до вершин и запоминаем, из какой вершины мы попадаем в вершину j}

Методическое отступление. Пусть $\text{GetColor}(i, t: \text{Integer})$ — функция определения цвета светофора i в момент времени t , а $\text{DetTimes}(i, c, t: \text{integer}; \text{Var } mt: \text{SmlM})$ — процедура вычисления трех моментов времени переключения цветов светофора i , имеющего цвет c в момент времени t . Соревнование на написание лучшего программного кода этих модулей — интересная задача.

Методические указания

Данная задача является хорошим творческим заданием при изучении темы «Алгоритмы на графах», а именно, при разборе алгоритмов поиска кратчайших путей в графе.

Самостоятельные исследования

Обоснуйте достаточность трех моментов времени переключения цветов каждого из двух светофоров для нахождения времени совпадения их цветов свечения.

«Выравнивание»

Условие задачи

Для игры в пасьянс задаются последовательно расположенные n стопок, в каждой из которых находится некоторое количество (возможно нулевое) карточек (рис. 11.3а). Стопки перенумерованы целыми числами от 1 до n . Для того, чтобы сделать один ход, вы должны указать номер стопки, например p , и назвать некоторое число, например m . Это соответствует тому, что m карточек перемещаются из стопки с номером p на каждую из соседних с ней стопок (рис. 11.3б). Для стопки p соседними являются стопки с номерами $p-1$ и $p+1$ при $1 < p < n$. Для $p=1$ соседней является только стопка с номером 2, а для $p=n$ — стопка $n-1$. Для выполнения одного хода в стопке p должно находиться по меньшей мере $2m$ карточек, если она имеет двух соседей, и по меньшей мере m карточек, если имеется только один сосед.

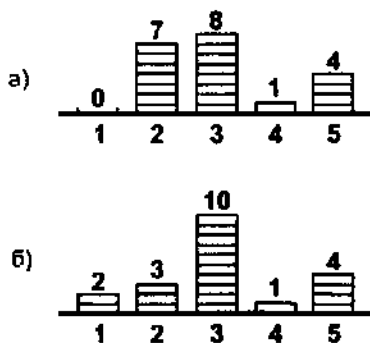


Рис. 11.3. Пять стопок с количеством 0, 7, 8, 1 и 4 (а); те же стопки после хода для $p=2$, $m=2$ (б)

Целью игры является «выравнивание» всех стопок, то есть требуется сделать так, чтобы в стопках находилось одинаковое количество карточек, при этом затратив как можно меньшее количество шагов. Если существует более одного способа, то вы должны вывести только один из них.

Ограничения:

- гарантируется, что можно выровнять заданные стопки карточек не более чем за 10000 шагов;
- $2 \leq n \leq 200$;
- $0 \leq C_i \leq 2000$, где C_i — количество карточек в стопке i в момент начала игры ($1 \leq i \leq n$).

Входные данные. Ввод осуществляется из текстового файла с именем `flat.inp`, содержащего две строки:

- первая строка — n ;
- вторая строка содержит n целых чисел C_i .

Выходные данные. Вывод производится в выходной текстовый файл с именем `flat.out`:

- первая строка: количество шагов q ;
- каждая из последующих q строк содержит два целых числа, описывающих соответствующий ход: p, m .

Порядок ходов в выходном файле должен быть таким же, как и в игре. Так, ваш первый ход будет записан во второй строке выходного файла.

Пример.

```
flat.inp:
5
0 7 8 1 4
```

```
flat.out
5
5 2
3 4
2 4
3 1
4 2
```

Проверка. Ваша программа должна выполняться не более 3 секунд. Чтобы получить полный балл A за тест, нужно, чтобы количество шагов x было меньше или равно числу B , установленному тестирующей программой. Обратите внимание, что B не обязательно является минимумом. Фактически B зависит от

количества ходов, сделанных в соответствии с некоторой простой стратегией без избыточных ходов, и от среднего количества карточек в стопках. Вы можете получить часть баллов, назначенных за тест. Баллы, которые вы получите, вычисляются путем округления вычисленного по следующей формуле числа до ближайшего целого:

$$\begin{cases} A \text{ if } x \leq B; \\ 2A\left(\frac{3}{2}B - x\right) / B \text{ if } B < x < \frac{3}{2}B; \\ 0 \text{ if } x \geq \frac{3}{2}B. \end{cases}$$

Обсуждение задачи

Красота этой задачи заключается в том, что традиционные методы конструирования пространства состояний задачи (проблемного пространства) в данном случае не работают! Если бы нам удалось его построить, то потребовалось бы найти кратчайший путь из начального состояния в конечное. О переборе всех вариантов и говорить не приходится. Требуется идея — конструктивная идея, причем система проверки подсказывает эвристический характер решения.

Рассмотрим простой пример. Пусть даны три стопки и три карточки (обозначим их парой чисел $(3, 1)$ — количество стопок и количество карточек в стопке в конечном состоянии). Тогда все возможные переходы из состояния в состояние приведены на рис. 11.4.

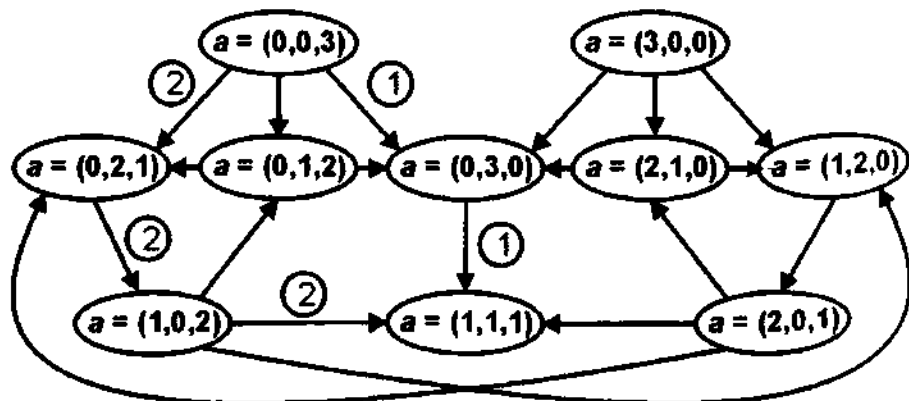


Рис. 11.4. Простой пример переходов из состояния в состояние при трех стопках и трех карточках

Начальное состояние — $(0, 0, 3)$. В конечное состояние — $(1, 1, 1)$ — мы можем попасть и за два хода, и за три хода. На рис. 11.4 эти пути отмечены цифрами ① и ②. Существуют и еще более длинные цепочки ходов: $(0, 0, 3) \Rightarrow (0, 2, 1) \Rightarrow (1, 0, 2) \Rightarrow (1, 2, 0) \Rightarrow (2, 0, 1) \Rightarrow (1, 1, 1)$, а также циклы: $(0, 2, 1) \Rightarrow (1, 0, 2) \Rightarrow (1, 2, 0) \Rightarrow (2, 0, 1) \Rightarrow (0, 2, 1)$.

Зададим вопрос — есть ли что-то общее у цепочек ходов, отмеченных цифрами ① и ② на рис. 11.4? Да! Из первой стопки мы не перекладывали карточки, из второй стопки — переложили две карточки, из третьей стопки — три карточки. Зафиксируем это в некотором массиве: $v = (0, 2, 3)$.

Еще одно наблюдение: если не допускать из стопки i перекладываний, приводящих к суммарному значению карточек (переложенных из стопки), большему, чем значение $v[i]$, то более длинные цепочки из рассмотрения исключаются. А если не допускать перекладываний из стопки, отмеченной в v нулевыми значениями, то исключаются и циклические цепочки.

Итак, суть идеи мы сформулируем в виде вопроса — можно ли по начальному состоянию карточек сформировать в некой структуре данных (массиве v) описание этого состояния, показывающее, сколько карточек из каждой стопки следует переместить для достижения конечного состояния?

Оказывается, это возможно! На рис. 11.5 приведено изменение элементов массива v при переходах из одного состояния в другое, указанных на рис. 11.4. Карточки перемещаются из стопок, для которых соответствующий элемент массива v не равен нулю.

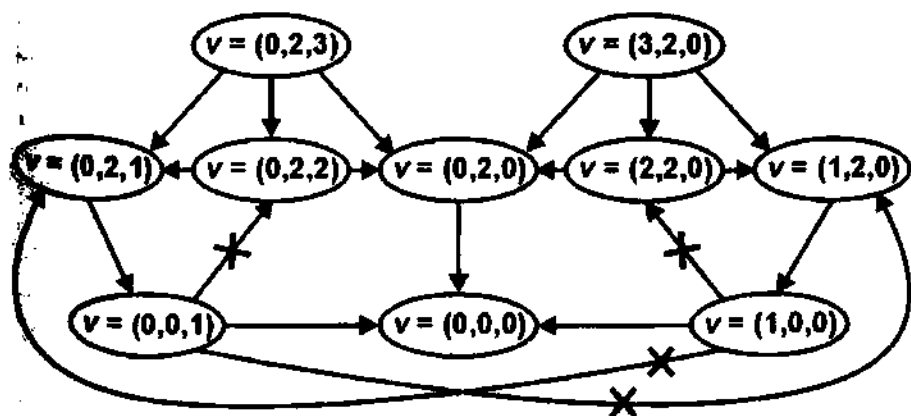


Рис. 11.5. Изменение элементов массива v при переходах из состояния в состояние на рис. 11.4. Символом «X» отмечены запрещенные переходы

Итак, задача логически разбивается на две части:

- построение по начальному состоянию карточек, заданных в массиве a , массива v , в котором для каждой стопки указано количество перемещаемых из нее карточек при достижении конечного состояния;
- конструирование стратегии поиска наименьшего количества ходов, приводящих к результату, и наилучшего хода в каждой текущей позиции.

Методическое отступление. Пауза для обсуждения предложений школьников по логике формирования массива v будет, вероятно, вполне уместна.

Пусть в переменной n хранится значение количества карточек, которое должно быть в стопках после окончания игры (оно легко определяется при вводе данных). Величину $n - a[i]$ мы назовем *дисбалансом стопки* с номером i . Если дисбаланс первой стопки больше нуля, то из второй стопки в первую требуется переместить количество карточек, равное его значению ($v[2] := n - a[1]$, $v[1] := 0$). Для случая отрицательного дисбаланса — нужны обратные действия ($v[1] := a[1] - n$, $v[2] := 0$).

Сформулируем вопрос: сколько требуется взять (переместить) карточек из стопки с номером $i+1$, чтобы достигнуть баланса в стопках с номерами от 1 до i ?

Методическое отступление. Используя примеры, необходимо создать ситуацию, в которой школьник для себя откроет зависимость: $n = a[i] - 2*v[i] + v[i-1] + v[i+1]$.

Рассматриваем не крайнюю стопку с номером i . В начале игры в ней $a[i]$ карточек. Отдаем из нее $2*v[i]$ карточек. Получаем: $(v[i-1] + v[i+1])$. В конце игры в ней — n карточек. Из этой формулы получаем: $v[i+1] = n - a[i] - v[i-1] + 2*v[i]$. Таким образом, для достижения баланса в стопках с номерами от 1 до i из стопки $i+1$ требуется взять $v[i+1]$ карточек. Естественно, что затем, после просчета для всех средних стопок, это значение требуется удвоить.

Пример 1. Пусть $a = (0, 0, 0, 4)$ — пара $(4, 1)$, $n = 1$. Из второй стопки в первую требуется переместить одну карточку. Имеем: $v = (0, 1, 0, 0)$. Другими словами, для достижения баланса в первой стопке из второй требуется взять одну карточку. Далее, $v[3] = n - a[2] - v[1] + 2*v[2] = 3$. Получаем: $v = (0, 1, 3, 0)$. При подсчете учитывается: дисбаланс стопки — $(n - a[2])$; количество карточек, которые следует взять для достижения баланса в предыдущих стопках, — $v[1]$; количество карточек для

выполнения их перемещений из текущей стопки — $2*v[2]$. Итак, для достижения баланса в двух стопках из третьей необходимо взять не две карточки, а три. Продолжим: $v[4] = n - a[3] - v[2] + 2*v[3] = 6$ и имеем: $v = (0, 1, 3, 6)$. Для стопок, находящихся в середине, значение элементов массива v следует удвоить, — так учитывается специфика хода не из крайней стопки. Окончательный вид массива: $v = (0, 2, 6, 6)$.

Пример 2. Пусть $a = (0, 0, 7, 1)$ — пара (4, 2). После первого шага имеем: $v = (0, 2, 0, 0)$. Затем $v[3] = 2 - 0 - 0 + 2*2 = 6$, $v = (0, 2, 6, 0)$. Следующий шаг: $v[4] = 2 - 7 - 2 + 2*6 = 5$, $v = (0, 2, 6, 5)$. Окончательный вид: $v = (0, 4, 12, 5)$.

Пример 3. Пусть $a = (3, 1, 0, 0)$ — пара (4, 1). После первого шага имеем: $v = (2, 0, 0, 0)$. Затем $v[3] = 1 - 1 - 2 + 2*0 = -2$, $v = (2, 0, -2, 0)$. Следующий шаг: $v[4] = 1 - 0 - 0 + 2*(-2) = -3$, $v = (2, 0, -2, -3)$. Появление отрицательных чисел отражает факт перемещения карточек слева направо, а не справа налево, как в предыдущем примере. Переводим в положительную область: $v = (5, 3, 1, 0)$. Выполняем удвоение. Окончательный вид: $v = (5, 6, 2, 0)$.

Пример 4. Пусть $a = (1, 2, 3, 4, 10)$ — пара (5, 4). После первого шага имеем: $v = (0, 3, 0, 0, 0)$. Для достижения баланса в первой стопке из второй стопки следует взять три карточки. Затем $v[3] = 4 - 2 - 0 + 2*3 = 8$, $v = (0, 3, 8, 0, 0)$. Следующий шаг: $v[4] = 4 - 3 - 3 + 2*8 = 14$, $v = (0, 3, 8, 14, 0)$. Последний шаг: $v[5] = 4 - 4 - 8 + 2*14 = 20$, $v = (0, 3, 8, 14, 20)$. Окончательный вид: $v = (0, 6, 16, 28, 20)$.

Методическое отступление. Для усвоения логики формирования значений элементов массива v следует выполнить несколько примеров при произвольном размещении карточек в стопках. С этой целью школьникам можно предоставить процедуру Give_Info. Выполняя ручную трассировку (о программном коде пока нет и речи) — этот известный методический прием достижения понимания*, они приближаются к состоянию «Эврика», причем к самостоятельному его достижению (а только так оно и возможно). Главное после этого — «вычленить» основное: мы перевели исходное описание задачи в виде размещения карточек по стопкам в другую «плоскость», сохранив идентичность представления в смысле достижения целевых установок, и решать ее мы будем в этой новой плоскости описания. Если хотите, то здесь применен принцип «астрала»

(из фантастики); мы как бы взглянули на задачу из другого измерения, перевели ее в другое измерение!

```

Procedure Give_Info;
  Var min,i:Integer;
  Begin
    min:=0; {Для учета перемещений слева направо}
    If (n-a[1]>0) Then Begin v[2]:=n-a[1];v[1]:=0; End
    Else Begin v[1]:=a[1]-n; v[2]:=0; End;
  {Формируем первые два значения}
  For i:=2 To m-1 Do Begin {Подсчет}
    v[i+1]:=n-a[i]-v[i-1]+(2*v[i]);
    If (v[i+1]<min) Then min:=v[i+1];
  End;
  If min<0 Then {Переводим в положительную область}
    For i:=1 To m Do v[i]:=v[i]-min;
  For i:=2 To m-1 Do v[i] := 2*v[i]; {Формируем
  окончательные значения}
  cnt:=0; {Подсчитываем количество ненулевых
  элементов массива v. Оно определяет число итераций
  по достижении конца игры. Почему? В чем заключается
  разница между ходом в игре и итерацией?}
  For i:=1 To m Do
    If v[i]<>0 Then cnt:=cnt+1;
  End;

```

После проделанной работы можно перейти ко второй части обсуждения. Количество ненулевых элементов в массиве v (напомним, что он формируется один раз по исходным данным, а затем происходит только изменение его элементов в сторону уменьшения) мы подсчитали — это значение cnt . Наша задача заключается в том, чтобы сделать все элементы массива v равными нулю за наименьшее количество ходов. Очевидно, что брать карточки из стопки, для которой соответствующий элемент v равен нулю, бессмысленно.

Итерацией мы назовем обнуление одного элемента массива v , количество итераций — cnt . Одна итерация может состоять из нескольких шагов. Процесс игры конечен — на каждом шаге мы делаем ход, уменьшающий значение одного из элементов массива v (увеличений не бывает).

Методическое отступление. На данном этапе обсуждения главным является уяснение основной стратегии игры. Это можно сделать с помощью примеров. Так, на рис. 11.6 представлены два примера игры для пары (4, 2). Их разбор и прорисовка

дают осознание идеи и позволяют «набросать» (именно «набросать» — сделать «костяк»!) основную логику, представленную в процедуре Solve. Далее следует искать способы улучшения стратегии, реализация которых естественным образом «укладывается» в структуру Solve. Так как задача имеет эвристический характер решения, то многое зависит от эксперимента, а для этого требуется как генератор тестов, так и работающий вариант основной логики с реализацией технических деталей в виде запоминания ходов в некоторых структурах данных и вывода результата.

Procedure Solve;

Var i, val, t:Integer;
pp:Boolean;

Begin

While cnt>0 **Do Begin** {cnt — глобальная переменная, количество ненулевых элементов v}

val:=0;

pp:=False;

For i:=1 **To** m **Do Begin**

If {a[i]*v[i]}<>0 **Then Begin**

If a[i] >= v[i] **Then Begin**

<Запомнить i и v[i]>;

cnt:=cnt-1;

pp:=True;

Break; {Обязательный ход, приводящий к уменьшению значения cnt}

End;

{В стопке с номером i количество карточек меньше значения v[i]}

<Подсчитать количество перемещаемых карточек из стопки с номером i — значение переменной t>

If t>val **Then Begin**

val:=t;

<Запомнить i и t>;

End;

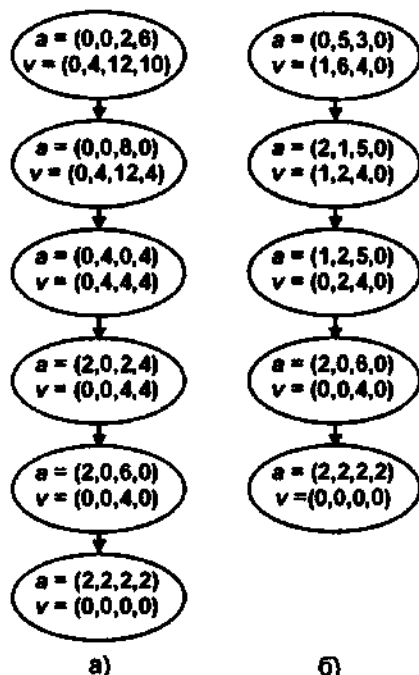


Рис. 11.6. Два примера игры для пары (4, 2)

If $t=val$ **Then** <Ответить на вопрос: какой ход лучше – ранее запомненный или из стопки i ?
Количество перемещаемых карточек одинаково>

End; {Конец **If** $(a[i]*v[i])<0$ }

End; {Конец цикла по i }

If Not(pp) **Then** <Нет стопок с $a[i] \geq v[i]$.

Можно уточнить логику выбора хода, ускоряющую процесс сходимости к результату. Например, чтобы следующий ход был обязательным – приводил к уменьшению значения cnt >

<Сделать ход>;

End; {Конец цикла по cnt }

End;

Итак, наша стратегия состоит из двух частей:

- есть однозначный ход, т. е. за один ход мы обнуляем один элемент массива v , реализуя тем самым одну итерацию;
- нет однозначного хода.

Первая ветвь стратегии очевидна.

Рассмотрим второй случай. «Напрашивается» решение, заключающееся в поиске стопки, из которой можно взять максимальное количество карточек; при этом граничные стопки (первая и m -я) могут идти при оценке с некоторым весовым коэффициентом в результат. А если максимум «дают» несколько стопок? В этом случае следует выбирать стопку, приводящую к наибольшему значению количества карточек в соседних стопках. Таким способом мы увеличиваем вероятность того, что следующий ход завершит итерацию. А если есть ход, приводящий к тому, что при следующем ходе будет завершена итерация, то он и выбирается!

Последний поиск можно осуществить с помощью простых функций.

Function $Get(t:Integer):Integer$;

Begin

If $(t=1)$ **Or** $(t=m)$ **Then** $Get := a[t]$

Else If $a[t] \bmod 2 \neq 0$ **Then** $Get := a[t]-1$

Else $Get := a[t]$;

End;

Function $Find:Integer$; {Находим стопку, взятие карточек из которой приведет на следующем ходе к завершению итерации}

Var $i, max, ent:Integer$;


```

Begin
  max: = 0;
  ent: = 0;
  If (a[1]*v[1]<>0) And (((a[1]+a[2])>v[2]) And
(v[2]<>0)) Then Begin
    ent: = 1;
    max: = a[1];
  End; {Оценка первой стопки}
  For i:=2 To m-1 Do {Оценка средних стопок}
    If ((a[i]*v[i])<>0) Then {Из стопки можно
взять карточки}
      If (((Get(i) Div 2)+a[i-1])>v[i-1]) And
(v[i-1]<>0)) Or
        (((Get(i) Div 2)+a[i+1])>v[i+1]) And
(v[i+1]<>0)) Then
        If (Get(i)>max) Then Begin
          ent := i;
          max := Get(i);
        End;
      If ((a[m]*v[m])<>0) And (((a[m]+a[m-1])>v[m-1])
And (v[m-1]<>0)) Then
        If a[m]>max Then ent:=m; {Оценка последней
стопки}
  Find:=ent;
End;

```

Методические указания

Данная задача интересна своей идеей перевода ее проблемно-го пространства в другое «измерение». Она не требует знания каких-то специальных алгоритмов, а ее эвристичность позволяет (при наличии тестов и проверяющей программы) организовать соревнование по поиску наилучшего результата.

Самостоятельные исследования

1. Разработка программного кода и поиск экспериментальным путем наилучших эвристик является мини-исследованием.
2. Приведет ли к упрощению задачи «снятие» особенности перемещения карточек из крайних стопок и разрешение их перемещения по кругу по общему правилу?

«Полоска земли»

Условие задачи

Жители Дингилвилля хотят найти место для строительства аэропорта. У них есть карта местности. Карта представляет со-

бой прямоугольную сетку с единичными квадратами, каждый из которых определен своими координатами (x, y) , где x — горизонтальная (запад–восток), а y — вертикальная (юг–север) координата. Для каждого квадрата на карте указана высота местности.

Требуется найти прямоугольную область наибольшей площади, состоящую из квадратов карты и удовлетворяющую следующим условиям:

- разница между высотой самого высокого и самого низкого квадрата этой области должна быть меньше или равной заданному числу q ;
- ширина этой области, то есть количество квадратов вдоль направления запад — восток, должна быть не больше 100.

Если существует более одной искомой области, вы должны указать только одну из них.

Ограничения:

- $1 \leq u \leq 700$, $1 \leq v \leq 700$, где u и v определяют размеры карты. В частности, u определяет количество квадратов в направлении запад–восток, а v — в направлении юг–север;
- $0 \leq q \leq 10$;
- $-30000 \leq h_{xy} \leq 30000$, где целое число h_{xy} — высота местности квадрата с координатами (x, y) , $1 \leq x \leq u$, $1 \leq y \leq v$;
- квадрат в юго-западном углу карты имеет координаты $(1, 1)$, а в северо-восточном — координаты (u, v) .

Входные данные. Входные данные вводятся из текстового файла с именем `land.inp`. Первая строка содержит три целых числа: u , v и q . Каждая из последующих v строк содержит целые числа h_{xy} для $x = 1, \dots, u$. В частности, h_{xy} является x -м числом в $(v-y+2)$ -й строке входного файла.

Выходные данные. Вывод должен осуществляться в текстовый файл с именем `land.out`, состоящий из одной строки. В этой строке должны быть четыре числа: x_{min} , y_{min} , x_{max} , y_{max} , где (x_{min}, y_{min}) — координаты юго-западного квадрата, а (x_{max}, y_{max}) — координаты северо-восточного квадрата этой области.

Обсуждение задачи

Фактически здесь в двумерном массиве нам требуется найти прямоугольную область наибольшей площади, в которой максимальный перепад высот не превышает заданного значения.

Введем понятие «скользящего окна» (рис. 11.7), являющееся эквивалентным понятию «прямоугольная область», но в большей степени отражающее суть задачи.



Рис. 11.7. Иллюстрация понятия «скользящее окно»

Принцип работы со «скользящим окном». Для пары значений (i, j) , где $j \geq i$, задающих размеры окна по вертикали, пара (q, k) , где $k \geq q$, задающая размеры окна по горизонтали, изменяется так: k увеличивается до тех пор, пока перепад высот находится в пределах нормы, q — увеличивается, если в окне перепад высот превышает допустимое значение. Таким образом, просматривая весь двумерный массив (всю местность) и фиксируя площади и координаты всех допустимых окон, мы и найдем наибольшую площадь, пригодную для строительства аэропорта.

Сложность этой задачи — не в «идейном» плане, а в способе его реализации, ибо алгоритмическое решение с временной характеристикой $O(n^4)$ — ее дают четыре цикла по границам окна (без учета времени пересчета границ окна), — является недопустимым. Однако приведем общую логику: с одной стороны, для ясности того, что мы делаем, а с другой — для формулировки требований к неизвестным пока составляющим решения.

Procedure Solve;

Var i, j, k, q, TSq : Integer;

Begin

$x1:=1; y1:=1; x2:=1; y2:=1; TSq:=1;$ {Начальные значения результирующей области}

For $i:=1$ **To** n **Do** **Begin** {Цикл по верхней вертикальной границе окна}

<A>;

For $j:=i$ **To** n **Do** **Begin** {Цикл по нижней вертикальной границе окна}

;

<C>;

$q:=1;$ {Значение левой горизонтальной границы окна}

```

For k:=1 To m Do Begin {Цикл по правой
горизонтальной границе окна}
  <D>;
  If <окно недопустимо> Then Begin
    <E>;
    q:=q+1; {Увеличиваем значение левой
горизонтальной границы}
  End;
  If (j-i+1)*(k-q+1)>TSq Then Begin
{Изменяем результат}
    x1:=q; x2:=k;
    y1:=n-j+1; y2:=n-i+1;
    TSq:=(j-i+1)*(k-q+1);
  End;
End;
End;
End;
End;

```

Допустимые временные характеристики «черных ящиков» A, B и $C — O(n)$, а D и $E — O(\log_2 n)$. В этом случае мы остаемся в разумных пределах времени решения задачи.

Первым шагом в разрешении оставшихся противоречий является хранение для вертикальных границ окна значений максимума и минимума в каждом столбце. Введем два массива max и min . Логика A в этом случае сводится к присвоению начальных значений:

```

For t:=1 To m Do Begin
  max[t]:=p[i,t]; {p — исходный массив, описывающий
высоту квадратов местности}
  min[t]:=p[i,t];
End;

```

А логика B — сводится к корректировке значений элементов max и min :

```

For t:=1 To m Do Begin
  If max[t]<p[j,t] Then max[t]:=p[j,t];
  If min[t]>p[j,t] Then min[t]:=p[j,t];
End;

```

Пусть $i = 1, j = 3$. Для примера из формулировки задачи массив $max = (44, 41, 43, 40, 39, 40, 41, 42, 42, 42)$, а массив $min = (39, 40, 39, 38, 36, 37, 35, 38, 35, 37)$. Если бы нас инте-

ресовали только перепады высот по вертикали (по одному измерению, а не по двум), то задача была бы решена. Детализация «черных ящиков» D и E в этом случае не вызывает сложности.

Методическое отступление. Просчет для различных значений i и j не вызывает сложности, но лучше его сделать, уточнить логику Solve и найти полосу максимальной площади, в которой перепады высот по вертикали не превышают значение c .

Если в зависимости от горизонтальных границ окна нам придется искать минимальный и максимальный элементы в соответствующих пределах массивов max и min , то мы получим линейное время, а нам требуется решать проблему за логарифмическую зависимость. Единственный путь — это организация данных в некую древовидную структуру. Пусть это будет «пирамида». Напомним ее понятие. Элементы массива b образуют пирамиду, если для любого допустимого значения i выполняется: $b[i] \leq b[2*i]$ и $b[i] \leq b[2*i+1]$. Особенностью работы с пирамидой в данном случае является то, что она не создается по некоему уже заданному массиву, а формируется и изменяется в процессе работы с массивом. Поэтому уместно назвать ее *динамической пирамидой*.

В данной задаче требуется работа с двумя динамическими пирамидами: данные массивов max и min , — точнее *индексы элементов* массивов, — организуются в соответствующем виде в массивах $HMax$ и $HMin$.

Обратимся к примеру. Случай $i = 1, j = 1, q = 1$ и $k = 10$ удовлетворяет условию задачи. Это полоска с перепадом высот (разностью), не превышающим значение c . Значение элементов массивов max и min совпадает: (41, 40, 41, 38, 39, 39, 40, 42, 40, 40), а значения $HMax = (10, 8, 1, 3, 2, 10, 6, 7, 4, 9, 5)$ и $HMin = (10, 4, 5, 6, 9, 2, 3, 7, 8, 1, 10)$. В $HMax[0]$ и $HMin[0]$ фиксируется количество элементов в пирамидах. Графическое изображение пирамид приведено на рис. 11.8.

Но есть еще одна проблема. Если включение элемента в пирамиду при изменении правой границы окна по горизонтали решается за логарифмическое время, то при изменении левой границы окна требуется удалять элемент из пирамиды. Для оперативного выполнения удале-

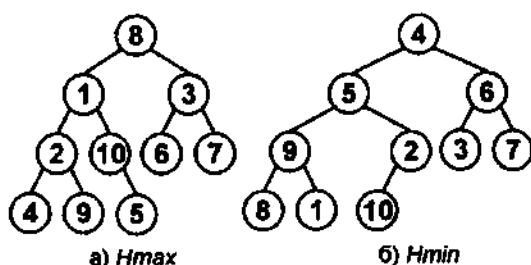


Рис. 11.8. Пирамиды для случая $i = 1, j = 1, q = 1$ и $k = 10$

ния введем для каждой пирамиды массив $NMax$ ($NMin$), элемент i которого будет указывать на место расположения элемента $max[i]$ ($min[i]$) в пирамиде. Для рассматриваемого случая они равны $NMax = (2, 4, 3, 8, 10, 6, 7, 1, 9, 5)$ и $NMin = (9, 5, 6, 1, 2, 3, 7, 8, 4, 10)$. В этом случае «черный ящик» S сводится к операторам $HMax[0] := 0$ и $HMin[0] := 0$, «черный ящик» D — к вставке индекса k (правой горизонтальной границы окна) в пирамиды $HMax$ и $HMin$ (вызов процедур $InsMax(k)$ и $InsMin(k)$), а «черный ящик» E — к удалению из пирамид элементов, находящихся в них на местах $NMax[q]$ и $NMin[q]$. В качестве образца логики приведем процедуру вставки значения в пирамиду индексов минимальных элементов окна с одновременным изменением массива $NMin$:

```

Procedure InsMin(a:Integer);
  Var i:Integer;
  Begin
    HMin[0] := HMin[0] + 1;
    i := HMin[0];
    While (i > 1) And (min[HMin[i Div 2]] > min[a]) Do
  Begin {Пока вставляемый элемент меньше очередного
    элемента, идем вверх по пирамиде, освобождая место
    и уменьшая значение i в два раза}
    NMin[HMin[i Div 2]] := i;
    HMin[i] := HMin[i Div 2];
    i := i Div 2;
  End;
  HMin[i] := a; NMin[a] := i;
End;

```

Методическое отступление. Добавив процедуру $InsMax$, следует попробовать построить пирамиды и сформировать массивы $NMin$ и $NMax$ для произвольной прямоугольной области массива p . Естественно, что перед этой работой следует выписать массивы max и min .

Ситуацию с удалением элементов из пирамид проиллюстрируем примерами. Пусть $i = 1$, $j = 2$, $q = 1$ и $k = 4$. Массив $max = (41, 40, 43, 40, \dots)$, а $min = (39, 40, 41, 38, \dots)$. Пирамиды следует изменять (рис. 11.9), так как перепад высот равен 5. Для случая $i = 1$, $j = 2$, $q = 4$ и $k = 7$ соответствующие участки массивов имеют вид: $max = (\dots, 40, 39, 39, 40, \dots)$, а $min = (\dots, 38, 36, 37, 35, \dots)$. Справа получен перепад высот 5. Сдвигаем левую горизонтальную границу (рис. 11.10) окна.

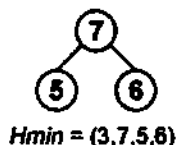
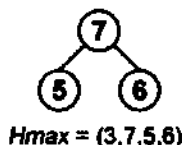
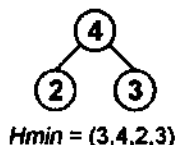
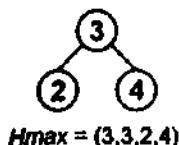
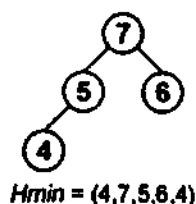
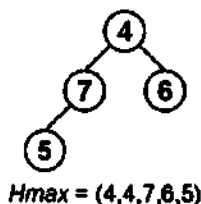
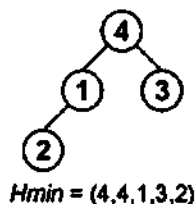
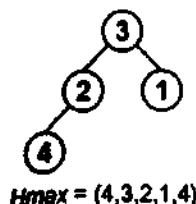


Рис. 11.9. Изменение пирамид для случая $i = 1, j = 2, q = 1, k = 4$. Левая горизонтальная граница окна смещается на одну позицию вправо

Рис. 11.10. Изменение пирамид для случая $i = 1, j = 2, q = 4, k = 7$. Левая горизонтальная граница окна смещается на одну позицию вправо

Методическое отступление. Для значений $i = 5, j = 11, q = 4, k = 9$ массив $\max = (42, 41, 44, 42, 41, 40, 41, 41, 43, 42)$, $\min = (31, 37, 36, 38, 38, 38, 39, 39, 37, 39)$, $H_{\max} = (6, 9, 5, 4, 7, 8, 6, \dots)$, $N_{\max} = (*, *, *, 3, 2, 6, 4, 5, 1, \dots)$, $H_{\min} = (6, 9, 5, 4, 7, 8, 6, \dots)$ и $N_{\min} = (*, *, *, 3, 2, 6, 4, 5, 1, \dots)$. Просьба к учащимся о прорисовке пирамид до и после изменения левой горизонтальной границы окна должна предшествовать написанию (разработке) программного кода.

Методические указания

Решение задачи при «слабых» временных ограничениях и небольших размерностях исходных данных не требует умения работать с динамическими структурами данных (двоичными деревьями и т. д.). В этом случае ее можно использовать как обычную задачу повышенной сложности при изучении двумерных массивов. Но наиболее лаконично она «вписывается» в тему «Алгоритмы быстрой сортировки» — когда изучена пирамидальная сортировка как на обычном массиве, так и с использованием двоичных деревьев.

Самостоятельные исследования

Разработка как минимум двух способов программной реализации задачи и сравнение их временных характеристик на объектных тестах является предметом самостоятельного исследования для школьника, желающего надеяться на значительный прирост профессиональных навыков и умений.

ЗАДАЧИ ПЕРВОГО ТУРА

«Парковка»

Условие задачи

Парковка около Великой Стены состоит из длинного ряда парковочных мест, полностью заполненного машинами. Один из краев считается левым, а другой — правым. Каждая из машин имеет свой тип. Типы перенумерованы натуральными числами, и несколько машин могут иметь один и тот же тип. Рабочие решили упорядочить машины в парковочном ряду так, чтобы их типы возрастали слева направо. Упорядочение состоит из нескольких этапов. В начале каждого этапа рабочие одновременно выводят машины с их мест и затем ставят машины на свободные места, образовавшиеся в результате выезда машин на данном этапе, то есть меняют их местами. Каждый рабочий перемещает только одну машину за этап, и некоторые рабочие могут не участвовать в перемещении машин на этапе. Эффективность процесса упорядочивания машин определяется количеством этапов.

Предположим, что на парковке находится n машин и w рабочих. Требуется написать программу, которая находит такое упорядочивание, при котором количество этапов не превосходит значения $n/(w-1)$, округленного в большую сторону, то есть $\lceil n/(w-1) \rceil$. Известно, что минимальное число этапов не превосходит $\lceil n/(w-1) \rceil$.

Рассмотрим следующий пример. На парковке находятся 10 машин типов 1, 2, 3, 4 и четверо рабочих. Начальное положение машин слева направо, заданное их типами, — 2 3 3 4 4 2 1 1 3 1.

Минимальное количество этапов — 3, и они могут быть выполнены так, что расположение машин по типам после каждого из этапов следующее:

2 1 1 4 4 2 3 3 3 1 — после первого этапа,

2 1 1 2 4 3 3 3 4 1 — после второго этапа,

1 1 1 2 2 3 3 3 4 4 — после третьего этапа.

Входные данные. Входной файл имеет имя CAR.IN. Первая строка входного файла состоит из трех чисел. Первое — количество машин n , $2 \leq n \leq 20000$. Второе — количество типов ма-

ция m , $2 \leq m \leq 50$, причем тип машины является натуральным числом от 1 до m . На парковке есть хотя бы одна машина каждого типа. Третье — количество рабочих w , $2 \leq w \leq m$. Вторая строка содержит n целых чисел, где i -е число является типом i -й машины в парковочном ряду, считая слева направо.

Выходные данные. Выходной файл имеет имя CAR.OUT. Первая строка выходного файла содержит число r — количество этапов. Далее следует r строк, где $i+1$ строка выходного файла описывает i -й этап. Формат строки, описывающей этап, следующий. Первое число — количество машин s , перемещаемых на данном этапе. Далее следует s пар чисел, определяющих перемещения машин на данном этапе. Первое число пары — позиция, с которой машина перемещается, второе число пары — позиция, на которую машина перемещается. Позиция машины является целым числом от 1 до n и отсчитывается с левого конца ряда.

В случае, если существует несколько оптимальных решений, выдайте только одно из них.

Пример.

```
CAR.IN
10 4 4
2 3 3 4 4 2 1 1 3 1
```

```
CAR.OUT
3
4 2 7 3 8 7 2 8 3
3 4 9 9 6 6 4
3 1 5 5 10 10 1
```

Оценка решения. Предположим, что количество этапов, выданных вашей программой, — r , а $\lceil n/(w-1) \rceil = q$. Если какой-то из r этапов выдан некорректно или после выполнения всех этапов типы машин не выстраиваются в возрастающем порядке, вы набираете 0 баллов за тест. В противном случае ваши баллы будут начислены в соответствии со следующими правилами:

- $r \leq q$ — 100% баллов;
- $r = q+1$ — 50% баллов;
- $r = q+2$ — 20% баллов;
- $r \geq q+3$ — 0% баллов.

Обсуждение задачи

Рассмотрим сначала более простую задачу. Дан массив $A[1..n]$, элементы которого равны 0 или 1 (записаны в случай-

ном порядке). Требуется переставить элементы так, чтобы в начале массива были записаны нули, а затем единицы.

Решение такой задачи очевидно:

```
i:=0; j:=n;
While i<j Do
  If A[i+1]=0 Then i:=i+1
  Else If A[j]=1 Then j:=j-1
      Else Begin A[i+1]↔A[j];
                i:=i+1; j:=j-1;
      End;
End;
```

Такая идея — просмотр элементов массива с двух сторон — является основной в двоичном поиске.

Усложним задачу: пусть элементы массива равны 0, 1 или 2*. Она решается по той же схеме:

```
i:=0; m:=0; j:=n;
While m<>j Do Begin
  If A[m+1]=1 Then Begin
    m:=m+1;
  End
  Else
    If A[m+1]>1 Then Begin
      A[m+1]↔A[j];
      j:=j-1;
    End
    Else Begin
      A[m+1]↔A[i+1];
      i:=i+1; m:=m+1;
    End;
End;
```

Можно ли обобщить эту задачу для случая 0, 1, 2 или 3, и есть ли в этом смысл? Мы теряем изящество, простоту решения и приходим к стандартным методам сортировки.

Другое направление обобщения — считать одной операцией не перестановку двух элементов массива, а одновременную перестановку w элементов. Эта сторона обобщения и представлена в данной задаче.

Методическое отступление. Решение сформулированных задач при $w = 3$ или 4 окажется хорошей пропедевтикой рассматриваемой проблемы.

* Шень А. Программирование: теоремы и задачи. М.:МЦНМО, 1995. С. 37.

Рассмотрим пример, приведенный в формулировке задачи. На рис. 12.1 дана схема его решения. В чем ее особенность?

Берем первую машину, стоящую не на своем месте. Значит, на том месте, где расположены машины с этим номером, есть машина, стоящая также не на своем месте. Находим первое такое место и с новой машиной выполняем эти же действия. Процесс продолжается до тех пор, пока есть рабочие (рис. 12.1,

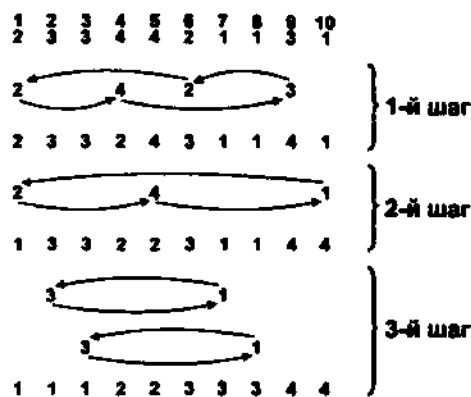


Рис. 12.1. Пример решения задачи о парковке

1-й шаг) или не будет получен цикл (рис. 12.1, 2-й шаг). В первом случае последняя машина может оказаться не на своем месте, но в этом нет ничего страшного, так как $w-1$ машин мы ставим на нужные места. Во втором же случае, если осталось более одного рабочего, процесс возобновляется, но с другой начальной машиной (рис. 12.1, 3-й шаг).

Что гарантирует нам этот простой эвристический алгоритм? На каждом шаге не менее $w-1$ машин оказываются на своем месте, а всего шагов — $\lceil n/(w-1) \rceil$, т. е. мы получаем 100% баллов.

Методическое отступление. Желательно в этот момент обсуждения обратиться к примерам и в «ручном» режиме просчитать их.

Обратимся к особенностям реализации решения. Какие дополнительные данные облегчат нам задачу? При вводе исходного размещения машин (пусть это будет массив A) следует сформировать, на наш взгляд, два дополнительных массива: t и $need$. В первом из них мы определяем конечную позицию для каждого типа машин в результирующем размещении, а во втором — само размещение. Для примера из формулировки задачи они имеют вид: $t = (3, 5, 8, 10)$ и $need = (1, 1, 1, 2, 2, 3, 3, 3, 4, 4)$. Небольшая «добавка» при вводе данных решает эту задачу. Фрагмент ее реализации имеет вид:

```
...
Read (n, m, w);
For i:=1 To n Do Begin {Вводим текущее положение
машин и подсчитываем количество машин каждого типа}
```

```

    Read(A[i]);
    t[A[i]]:=t[A[i]]+1;
End;
t[0] := 0;
For i:=1 To m Do t[i]:=t[i]+t[i-1];
r := 1; {Формируем требуемую расстановку машин на
стоянке}
For i:=1 To n Do Begin
    need[i] := r;
    If t[r]=i Then r:=r+1;
End;
...

```

В этом случае поиск машины, стоящей не на своем месте, реализуется следующим образом:

```

...
q:=tt; {tt - номер текущей машины}
j:=t[q-1]+1;
While (j<=t[q]) And (A[j]=need[j]) Do j:=j+1;
If j<=t[q] Then Begin
    <фиксируем для вывода результата найденную
перестановку двух машин>;
    q:=A[j]; A[j]:=tt; {Ставим машину с номером tt
на свое место}
    <выполняем рекурсивный вызов логики для машины
с номером q, находящейся на месте j>;
End
...

```

По аналогичной схеме реализуется и поиск первой машины, находящейся не на своем месте:

```

j := 1;
While (j<=n) And (need[j]=A[j]) Do j:=j+1;
If j<=n Then <рекурсивный вызов логики для машины
с номером A[j], находящейся на месте j>;

```

Если машину с номером q требуется поставить на место j , то проверка того, что получен цикл, сводится к анализу равенства $need[j]=q$. Дальнейшее уточнение особенностей реализации оставим для практической работы школьника.

Методические указания

Эта задача по своему содержанию относится к теме «Сортировка и поиск данных». Она требует достаточно совершенного

владения техникой написания рекурсивных программ, и ее использование в качестве творческого задания разумно и целесообразно.

Самостоятельные исследования

1. Поиск результата за минимальное количество шагов остался за рамками обсуждения.
2. Исследуйте зависимость минимального количества шагов от значения величины w .

«Медианная энергия»

Условие задачи

В новом космическом эксперименте участвуют n объектов, пронумерованных от 1 до n . Известно, что n — нечетно. Каждый объект имеет уникальную, но неизвестную энергию y , выражающуюся натуральным числом, $1 \leq y \leq n$. Назовем объектом с медианной энергией такой объект x , для которого количество объектов с энергией, меньшей, чем у x , равно количеству объектов с энергией, большей, чем у x . Требуется написать программу, которая определяет объект с медианной энергией. К сожалению, сравнивать энергии можно только с помощью устройства, способного для трех различных заданных объектов определить, какой из них имеет медианную энергию.

Библиотека. Вам предоставляется библиотека `device` с тремя операциями:

- `GetN`, вызывается только один раз в начале программы; не имеет аргументов; возвращает значение n ;
- `Med3`, вызывается с тремя различными номерами объектов в качестве аргументов; возвращает номер объекта, имеющего медианную (среднюю) энергию;
- `Answer`, вызывается только один раз в конце программы; единственный аргумент — номер объекта x , имеющего медианную энергию. Эта функция корректно завершает выполнение вашей программы.

Библиотека `device` генерирует два текстовых файла: `MEDIAN.OUT` и `MEDIAN.LOG`. Первая строка файла `MEDIAN.OUT` содержит одно целое число: номер объекта, переданного библиотеке при вызове `Answer`. Вторая строка содержит одно целое число: количество вызовов `Med3`, которое было сделано вашей программой. Диалог между вашей программой и библиотекой записывается в файл `MEDIAN.LOG`.

Инструкция для программирующих на языке Pascal: включите строку `uses device;` в текст вашей программы.

Инструкция для программирующих на C/C++: включите строку `#include "device.h"` в текст вашей программы, создайте проект MEDIAN.PRJ и добавьте файлы MEDIAN.C (MEDIAN.CPP) и DEVICE.OBJ в этот проект.

Экспериментирование. Вы можете поэкспериментировать с библиотекой, создав текстовый файл DEVICE.IN. Файл должен содержать две строки. В первой строке записано одно целое число: количество объектов n . Во второй строке — целые числа от 1 до n в некотором порядке: i -е число является энергией объекта с номером i .

Пример.

```
DEVICE.IN:
5
2 5 4 3 1
```

Файл DEVICE.IN, указанный выше, описывает 5 объектов с их энергиями:

Номер объекта	1	2	3	4	5
Энергия	2	5	4	3	1

Ниже приведена корректная последовательность с пятью обращениями к библиотеке:

- 1) `GetN` (in Pascal) or `GetN()` (in C/C++) возвращает 5;
- 2) `Med3(1, 2, 3)` возвращает 3;
- 3) `Med3(3, 4, 1)` возвращает 4;
- 4) `Med3(4, 2, 5)` возвращает 4;
- 5) `Answer(4)`.

Ограничения:

- количество объектов n нечетно и $5 \leq n \leq 1499$;
- для объекта с номером i выполняется $1 \leq i \leq n$;
- для энергии объекта y выполняется $1 \leq y \leq n$, и все энергии различны;
- библиотека для Pascal содержится в файле: `device.tpu`;
- объявление функций и процедуры на Pascal:

```
function GetN: integer;
function Med3(x,y,z:integer):integer;
procedure Answer(m:integer);
```

- библиотека для C/C++: `device.h`, `device.obj` (используйте модель памяти `large`);

• заголовки функций для C/C++ :

```
int GetN(void);
int Med3(int x, int y, int z);
void Answer(int m);
```

- в ходе исполнения программы разрешается использовать не более 7777 вызовов функции Med3;
- ваша программа не должна читать или писать какие-либо файлы.

Обсуждение задачи

Для наглядности обсуждения «привяжем» его к конкретному примеру. Пусть дан массив с энергиями объектов, представленный в табл. 12.1. Нам требуется найти номер элемента массива, имеющего 7-е значение энергии.

Таблица 12.1

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	10	2	11	12	9	5
Признак	M	L	L	M	N	L	R	R	L	R	R	N	L

Все, что мы можем сделать с помощью операции Med3, — это разбить массив на три части, выбрав случайным образом два несовпадающих номера (a и b) элементов. Это не деление пополам, как в бинарном поиске, а деление на три части — «*тринарный*» поиск. Пусть $a = 5$ и $b = 12$. Просматриваем все номера объектов (i), помеченные признаком True. Если операция Med3(a, b, i) дает в качестве результата:

- значение a , то помечаем объект с номером i как элемент, принадлежащий левой части (признак Left);
- значение i , то помечаем объект с номером i как элемент, принадлежащий средней части (признак Middle);
- значение b , то помечаем объект с номером i как элемент, принадлежащий правой части (признак Right).

В табл. 12.1 буквами «L», «M» и «R» (в третьей строке) обозначены признаки Left, Middle и Right. Буквой «N» обозначен признак None, который указывает, что данный элемент не рассматривается на данном шаге анализа.

Разбили, — а что дальше? Направивается единственно разумный ход — выбрать одну из трех частей и продолжить поиск (другого просто не дано).

Вся сложность задачи состоит в выборе параметров поиска и в аккуратном переходе от одного этапа поиска к другому.

Какую часть выбрать для следующего этапа поиска? Мы имеем в левой части результата поиска 5 элементов, в правой — 4 и в средней (без учета a и b) — 2. После этого этапа необходимо искать элемент с первым значением энергии среди элементов, помеченных признаком Middle.

Изменим массив энергий — табл. 12.2: мы поменяли значения у пятого и двенадцатого элементов массива.

Таблица 12.2

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	10	2	11	12	6	5
Признак	M	R	R	M	N	R	L	L	R	L	L	N	R

Признаком «L» помечено 4 элемента, а признаком «R» — 5 элементов. Согласно предыдущей логике, далее требуется искать элемент со вторым значением энергии среди элементов, помеченных признаком Middle.

Еще один пример, точнее модификация рассматриваемого нами. Пусть $a = 8$ и $b = 11$. Данные первого просмотра приведены в табл. 12.3.

Таблица 12.3

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	10	2	11	12	6	5
Признак	L	L	L	L	L	L	R	N	L	M	N	L	L

Очевидно, что на следующем шаге элемент со средней энергией следует искать в части массива с признаком «L». А если поменять значения восьмого и одиннадцатого элементов (табл. 12.4), то результат — где искать? — становится уже не столь очевидным.

Таблица 12.4

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	12	2	11	10	6	5
Признак	R	R	R	R	R	R	L	N	R	M	N	R	R

После разрешения этой неясности (образно выражаясь, «идти туда, не зная куда», а мы бы еще добавили — «не зная как») все остальное — чисто техническая работа.

Методическое отступление. Рекомендуется в этот момент обсуждения сделать несколько «ручных» просчетов и попытаться сгенерировать идею разрешения неясности.

Для конкретизации дальнейшего обсуждения сделаем несколько технических замечаний. Опишем задачу следующей совокупностью величин:

```
Const MaxN=1500;  
Type TDir = (None, Middle, Left, Right);  
Var n:Integer;  
M:Array [1..MaxN] Of TDir
```

Введение типа данных TDir повышает наглядность кода. Тогда выделение элементов массива, среди которых ищется элемент, осуществляется с помощью простой процедуры MakeMiddle:

```
Procedure MakeMiddle(Dir:TDir); {Помечаем элементы,  
среди которых осуществляется поиск}  
Var i:Word;  
Begin  
  For i := 1 To n Do  
    If M[i]=Dir Then M[i]:=Middle Else M[i]:=None;  
End;
```

2.)

В начальный момент все элементы массива *M* имеют значение Middle.

Формирование значений переменных *a* и *b* имеет свои, чисто технические, особенности. Дело в том, что номера оставшихся элементов (не на первом шаге) для поиска идут не подряд, а значения *a* и *b* должны указывать на элементы, подлежащие поиску:

```
Procedure GetNext(Var a,b:Word; len:Word);  
{Генерируем номера a и b элементов массива энергий  
или массива M. Значение len — длина интервала  
поиска, количество чисел, помеченных признаком Middle}  
Var i,t,ca,cb:Word;  
Begin  
  t:=0;  
  ca:=Random(len)+1; {Функция Random дает  
псевдослучайным образом целое число в интервале  
[0, len-1]}  
  cb:=Random(len)+1;
```

```

While ca=cb Do cb:=Random(len)+1;
For i:=1 To n Do
  If M[i]=Middle Then Begin {Поиск осуществляется
    только среди элементов, помеченных признаком Middle;}
    t:=t+1;
    If ca=t Then a:=i;
    If cb=t Then b:=i;
  End;
End;

```

Вернемся к обсуждению задачи и зададим себе вопрос: в том ли направлении мы идем? Другими словами, так ли принципиальны значения Left и Right? Да, они задают, определяют область дальнейшего поиска, — а дальше? Вернемся к табл. 12.3. Перед вторым шагом поиска она имеет вид, приведенный в табл. 12.5. Мы ищем седьмой элемент, но уже среди девяти элементов.

Таблица 12.5

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	10	2	11	12	6	5
Признак	M	M	M	M	M	M	N	N	M	N	N	M	M

Пусть $a = 1$ и $b = 6$. После поиска по нашей логике массив признаков имеет вид, представленный в табл. 12.6.

Таблица 12.6

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	10	2	11	12	6	5
Признак	N	R	R	M	L	N	N	N	R	N	N	M	M

Признаком «L» помечен один элемент, знаком «M» — три. С учетом двух элементов на границах следует искать первый элемент среди тех, что помечены знаком «R». Ответ неверен! А если бы $a = 6$, $b = 1$, то данные поиска имели бы значения, показанные в табл. 12.7.

Таблица 12.7

Номер	1	2	3	4	5	6	7	8	9	10	11	12	13
Энергия	8	1	3	7	6	4	13	10	2	11	12	6	5
Признак	N	L	L	M	R	N	N	N	L	N	N	M	M

Признаком «L» здесь помечено три элемента, один элемент на a , три элемента помечены знаком «M». Требуется искать в дальнейшем третий элемент среди трех с знаком «M». Ответ верен.

Итак, мы приблизились к главному — следует сохранять структуру значений a и b при переходе от одного шага поиска к другому. Раскроем это утверждение:

- если мы переходим при поиске на участок массива, элементы которого помечены знаком «L», то новые значения a_{new} , b_{new} должны быть связаны с предыдущим значением a_{old} так, чтобы $a_{new} < b_{new} < a_{old}$. Выполнить это требование с помощью операции Med3 можно следующим образом:

```
If Med3(anew, bnew, aold) <> bnew Then anew ↔ bnew
```

- если мы переходим при поиске на участок массива, элементы которого помечены знаком «M», то новые значения a_{new} , b_{new} должны быть связаны с предыдущим значением b_{old} так, чтобы $a_{new} < b_{new} < b_{old}$. Выполнить это требование с помощью операции Med3 можно следующим образом:

```
If Med3(anew, bnew, bold) <> bnew Then anew ↔ bnew
```

- если мы переходим при поиске на участок массива, элементы которого помечены знаком «R», то новые значения a_{new} , b_{new} должны быть связаны с предыдущим значением b_{old} так, чтобы $b_{old} < a_{new} < b_{new}$. Выполнить это требование с помощью операции Med3 можно следующим образом:

```
If Med3(bold, anew, bnew) <> anew Then anew ↔ bnew
```

Методическое отступление. Следует привести примеры для всех трех вариантов перехода. В каждом варианте необходимо проанализировать случаи правильного и неправильного выбора новых значений a и b . Для облегчения работы можно предоставить школьникам формализованную запись логики решения задачи:

```
Procedure Rec(Dir:TDir; res, len, lb: Word);
  Var a, b, i, t,lt,lr:Word;
  Begin
    {Выделяем элементы, среди которых осуществляется
    поиск}
    MakeMiddle(Dir);
    If len=1 Then Begin i:=1;
```

```

        While M[i]<>Middle Do i:=i+1;
        Answer(i);
    End
Else Begin
    GetNext(a,b,len);
    {Корректируем значения a и b с учетом предыдущего
    поиска}
    If (Dir=Right) Then Begin
        If Med3(lb,a,b)<>a Then a $\leftrightarrow$ b
        End
    Else If Med3(a,b,lb)<>b Then a $\leftrightarrow$ b;
    If len=2 Then Begin
        If res=1 Then Answer(a)
        Else Answer(b)
        End
    Else Begin {Поиск или разбивка элементов
    массива M, помеченных признаком Middle на три части}
        lt:=0;
        rt:=0;
        M[a]:=None; M[b]:=None;
        For i:=1 To n Do
            If M[i]=Middle Then Begin
                t:=Med3(a,b,i);
                If t=b Then Begin rt:=rt+1; M[i]:=Right End
                Else If t=a Then Begin
                    lt:=lt+1;
                    M[i]:=Left;
                    End;
            End;
        End;
    {Искомый элемент попал на границу интервала, поиск
    завершен}
    If res=lt+1 Then Answer(a)
    Else
        If res=len-rt Then Answer(b)
        Else
            {Переход на поиск с новыми параметрами}
            If res<lt+1 Then Rec(Left,res,lt,a)
            Else
                If res<len-rt Then
                    Rec(Middle,res-lt-1,len-lt-rt-2,b)
                Else Rec(Right,res-len+rt,rt,b);
            End;
        End;
    End;
End;
End;

```

Методические указания

Целесообразно обсудить задачу после детального изучения двоичного поиска*. В данной книге методика обсуждения изменена в сторону улучшения (на наш взгляд) доступности. Для практической работы следует подготовить модуль device с учетом специфики используемой системы программирования и достаточно полный набор тестов. Соревнование на количество используемых вызовов Med3 создаст на занятии творческую атмосферу.

Самостоятельные исследования

Пусть с массивом энергий допустима не только операция $\text{Med3}(a, b, c)$, но и операция $\text{Swap}(a, b)$ — когда элементы с индексами a и b меняются местами. Разработайте алгоритм сортировки по возрастанию элементов массива энергий, использующий минимальное количество операций Swap .

«Палиндром»

Условие задачи

Палиндром — это симметричная строка, то есть она одинаково читается как слева направо, так и справа налево. Вы должны написать программу, которая по заданной строке определяет минимальное количество символов, которые необходимо вставить в строку для образования палиндрома.

Например, вставкой двух символов строка «Ab3bd» может быть преобразована в палиндром («dAb3bAd» или «Adb3bdA»), вставкой менее двух символов палиндром в этом примере получить нельзя.

Входные данные. Входной файл имеет имя PALIN.IN и состоит из двух строк. Первая строка содержит одно целое число — длину n входной строки, $3 \leq n \leq 5000$. Вторая — строку длины n , которая состоит из прописных (заглавных) букв от «A» до «Z», строчных букв от «a» до «z» и цифр от «0» до «9». Прописные и строчные буквы считаются различными.

Выходные данные. Выходной файл имеет имя PALIN.OUT и состоит из одной строки. Эта строка содержит одно целое число, которое является искомым минимальным числом символов.

Пример ввода и вывода.

PALIN.IN:

5

Ab3bd

PALIN.OUT:

2

```

        While M[i]<>Middle Do i:=i+1;
        Answer(i);
    End
Else Begin
    GetNext(a,b,len);
    {Корректируем значения a и b с учетом предыдущего
    поиска}
    If (Dir=Right) Then Begin
        If Med3(lb,a,b)<>a Then a<=>b
        End
    Else If Med3(a,b,lb)<>b Then a<=>b;
    If len=2 Then Begin
        If res=1 Then Answer(a)
        Else Answer(b)
        End
    Else Begin {Поиск или разбивка элементов
    массива M, помеченных признаком Middle на три части}
        lt:=0;
        rt:=0;
        M[a]:=None; M[b]:=None;
        For i:=1 To n Do
            If M[i]=Middle Then Begin
                t:=Med3(a,b,i);
                If t=b Then Begin rt:=rt+1; M[i]:=Right End
                Else If t=a Then Begin
                    lt:=lt+1;
                    M[i]:=Left;
                End;
            End;
        End;
        {Искомый элемент попал на границу интервала, поиск
        завершен}
        If res=lt+1 Then Answer(a)
        Else
            If res=len-rt Then Answer(b)
            Else
                {Переход на поиск с новыми параметрами}
                If res<lt+1 Then Rec(Left,res,lt,a)
                Else
                    If res<len-rt Then
                        Rec(Middle,res-lt-1,len-lt-rt-2,b)
                    Else Rec(Right,res-len+rt,rt,b);
                End;
            End;
        End;
    End;
End;
End;

```

Методические указания

Целесообразно обсудить задачу после детального изучения двойного поиска*. В данной книге методика обсуждения изменена в сторону улучшения (на наш взгляд) доступности. Для практической работы следует подготовить модуль `device` с учетом специфики используемой системы программирования и достаточно полный набор тестов. Соревнование на количество используемых вызовов `Med3` создаст на занятии творческую атмосферу.

Самостоятельные исследования

Пусть с массивом энергий допустима не только операция `Med3(a, b, c)`, но и операция `Swap(a, b)` — когда элементы с индексами a и b меняются местами. Разработайте алгоритм сортировки по возрастанию элементов массива энергий, использующий минимальное количество операций `Swap`.

«Палиндром»

Условие задачи

Палиндром — это симметричная строка, то есть она одинаково читается как слева направо, так и справа налево. Вы должны написать программу, которая по заданной строке определяет минимальное количество символов, которые необходимо вставить в строку для образования палиндрома.

Например, вставкой двух символов строка `«Ab3bd»` может быть преобразована в палиндром (`«dAb3bAd»` или `«Adb3bdA»`), а вставкой менее двух символов палиндром в этом примере получить нельзя.

Входные данные. Входной файл имеет имя `PALIN.IN` и состоит из двух строк. Первая строка содержит одно целое число — длину n входной строки, $3 \leq n \leq 5000$. Вторая — строку длины n , которая состоит из прописных (заглавных) букв от `«A»` до `«Z»`, строчных букв от `«a»` до `«z»` и цифр от `«0»` до `«9»`. Прописные и строчные буквы считаются различными.

Выходные данные. Выходной файл имеет имя `PALIN.OUT` и состоит из одной строки. Эта строка содержит одно целое число, которое является искомым минимальным числом символов.

Пример ввода и вывода.

`PALIN.IN:`

`5`

`Ab3bd`

`PALIN.OUT:`

`2`

* Окулов С. М. Основы программирования. 2-е изд., испр. М.: БИНОМ. Лаборатория знаний, 2005. С. 271–295.

Идея решения

Формируем матрицу $R[i, j]$ ($i, j = 1..n$), где $R[i, j]$ — минимальное количество букв, которые необходимо вставить в подстроку $S(i, j)$ (с i -го символа по j -й символ строки S), чтобы получить из нее палиндром. Искомый результат — $R[1, n]$.

Как формируется матрица R ?

$$R[i, j] = \begin{cases} 0, & \text{при } i \geq j; \\ R[i + 1, j - 1], & \text{при } S[i] = S[j] \text{ и } i < j; \\ \min(R[i + 1, j], R[i, j - 1]) + 1, & \text{при } S[i] \neq S[j] \text{ и } i < j. \end{cases}$$

Если символы $S[i]$ и $S[j]$ совпадают, то для преобразования строки в палиндром требуется столько же символов, сколько для преобразования подстроки $S(i+1, j-1)$. При несовпадении же символов требуется добавить один символ или к подстроке $S(i+1, j)$, или к подстроке $S(i, j-1)$, — к той из них, для которой преобразование в палиндром осуществляется за минимальное количество символов.

Методическое отступление. Следует устроить небольшой тренинг по заполнению матрицы R для конкретных примеров.

Пример 1. Пусть $S = \text{«Ab3bd»}$. Матрица R имеет вид, приведенный в табл. 12.8.

Таблица 12.8

0	1	2	1	2
—	0	1	0	1
—	—	0	1	2
—	—	—	0	1
—	—	—	—	0

Пример 2. Пусть $S = \text{«abcdba3»}$. Матрица R имеет вид, приведенный в табл. 12.9.

Таблица 12.9

0	1	2	3	2	1	2
—	0	1	2	1	2	3
—	—	0	1	2	3	4
—	—	—	0	1	2	3
—	—	—	—	0	1	2
—	—	—	—	—	0	1
—	—	—	—	—	—	0

Особенности программной реализации. Разумеется, хранить в памяти компьютера всю матрицу не следует, — достаточно одномерных массивов. Такой массив формируется за n итераций (i), и на каждой итерации вычисляются элементы от i до n . Другими словами, работает конструкция типа:

```
For i:=n DownTo 1 Do Begin
```

```
...
```

```
For j:=i To n Do <Корректировка элемента с номером j>
```

```
End;
```

Примечание. При реализации задачи в среде программирования Турбо-Паскаль возникает проблема нехватки оперативной памяти. Требуется использовать «кучу» или решать задачу в другой среде программирования.

Методические рекомендации

Это простая задача по теме «Динамическое программирование», она вполне доступна при первичном изучении данной темы. Увеличение значения n приводит к чисто техническим трудностям при реализации, но не более, если у нас есть возможность остаться в рамках временной сложности алгоритма (n^2).

Независимые исследования

1. Дана последовательность символов $\{s_1, s_2, \dots, s_n\}$, $1 \leq n \leq 200$; либо является пробелом, либо принадлежит множеству $[A, B, \dots, z]$. Группы символов, разделенные пробелами (одним или несколькими) и не содержащие пробелов внутри себя, мы будем называть словами. Палиндромом мы назовем такое слово $s_1, \dots, s_k$ ($k \leq 30$), что $s_1 = s_k, s_2 = s_{k-1}, s_3 = s_{k-2}, \dots$. При сравнении учитывается регистр символов, т. е. «А» $\neq$ «а». Требуется найти длину самого длинного палиндрома*.

Входные данные. В первой строке — значение n . Во второй строке — последовательность символов $s_1, s_2, \dots, s_n$.

Выходные данные. Выходной файл должен содержать длину самого длинного палиндрома.

2. Вводится строка S . Необходимо удалить из нее минимальное количество символов так, чтобы получился палиндром.

Строка S не пустая, имеет длину (n) не более 100 символов и состоит только из прописных латинских букв*.

Оцените, для каких значений n алгоритм решения задачи с временной сложностью $O(n^2)$ будет работать 1 секунду.

3. Заданы две символьные строки A и B , не содержащие пробелов. Требуется вычислить, сколькими способами можно получить строку B из строки A , вычеркивая некоторые символы**.

Пример. Если искомые строки A и B имеют, соответственно, вид «СамаринаИрина» и «Сара», то искомое число равно 7; для строк «aaabbbbccss» и «abc» это число равно 36.

(Эта задача предлагалась на областной олимпиаде школьников Кировской области в 1997 г.)

ЗАДАЧИ ВТОРОГО ТУРА

«Почтовые отделения»

Условие задачи

Вдоль прямой дороги расположены деревни. Дорога представляется целочисленной осью, а расположение каждой деревни задается одним целым числом — координатой на этой оси. Никакие две деревни не имеют одинаковых координат. Расстояние между двумя деревнями вычисляется как модуль разности их координат.

В некоторых, не обязательно во всех, деревнях будут построены почтовые отделения. Деревня и расположенное в ней почтовое отделение имеют одинаковые координаты. Почтовые отделения необходимо расположить в деревнях таким образом, чтобы общая сумма расстояний от каждой деревни до ближайшего к ней почтового отделения была минимальной.

Напишите программу, которая по заданным координатам деревень и количеству почтовых отделений находит такое расположение почтовых отделений по деревням, при котором общая сумма расстояний от каждой деревни до ее ближайшего почтового отделения будет минимальной.

Входные данные. Входной файл имеет имя POST.IN. В первой строке содержатся два целых числа: первое число — количество деревень v , $1 \leq v \leq 300$, второе число — количество почтовых отделений p , $1 \leq p \leq 30$, $p \leq v$. Вторая строка содержит v

* Долинский М. С. Алгоритмизация и программирование на Turbo Pascal: от простых до олимпиадных задач. СПб.: Питер, 2005. С. 137–139.

** Окулов С. М., Пестов А. А., Пестов О. А. Информатика в задачах. Киров: Изд-во ВГПУ, 1998. С. 173.

целых чисел в возрастающем порядке, являющихся координатами деревень. Для каждой координаты x выполняется $1 \leq x \leq 10000$.

Выходные данные. Выходной файл имеет имя POST.OUT. Первая строка содержит одно целое число s — общую сумму расстояний от каждой деревни до ее ближайшего почтового отделения для расположения почтовых отделений, описанного во второй строке. Вторая строка содержит p целых чисел в возрастающем порядке. Эти числа являются искомыми координатами почтовых отделений. Если для заданного расположения деревень есть несколько решений, то программа должна найти одно из них.

Пример ввода и вывода.

POST.IN:

10 5

1 2 3 6 7 9 11 22 44 50

POST.OUT:

9

2 7 22 44 50

Оценка решения. Если выходной файл не соответствует описанным требованиям — оценка 0. В противном случае оценка будет вычислена по табл. 12.10 следующим образом: если найденная сумма — s , а минимальная сумма — S_{min} , то необходимо вычислить $q = s/S_{min}$ и найти соответствующую оценку c в нижней строке.

Таблица 12.10

$q = s/S_{min}$	$q = 1.0$	$1.0 < q \leq 1.1$	$1.1 < q \leq 1.15$	$1.15 < q \leq 1.2$
c	10	5	4	3
$q = s/S_{min}$	$1.2 < q \leq 1.25$	$1.25 < q \leq 1.3$	$1.3 < q$	
c	2	1	0	

Обсуждение задачи

При размещении одного почтового отделения эта задача известна как *задача Лео Мозера* (1952 г.). Ее ответ: $v \text{ Div } 2+1$ (v — нечетное число) или любая из двух средних деревень (v — четное). Решение задачи с одним почтовым отделением на взве-

шенном графе (сеть дорог не на одной прямой) основано на использовании *алгоритма Р. Флойда** и поиска вершины с минимальной суммой расстояний до других вершин.

Однако при увеличении количества почтовых отделений эти идеи уже не работают. Переборные схемы применимы при ограничениях на входные данные в несколько десятков единиц, а мы в задаче имеем сотни единиц. Следовательно, остается поиск динамической схемы.

Рассмотрим пример, приведенный в формулировке задачи. Размещаем одно почтовое отделение в деревне с номером i . Оценим расстояние до деревень, находящихся левее ее. Они представлены в табл. 12.11. Так, если почтовое отделение размещено в четвертой деревне, то сумма расстояний до деревень 1, 2 и 3 равна 12.

Таблица 12.11

Номер деревни	1	2	3	4	5	6	7	8	9	10
Сумма расстояний	0	1	3	12	16	26	38	115	291	345

Что это дает? Пока ничего. Попробуем решить задачу для двух почтовых отделений. Рассмотрим деревни с номерами от 1 до i . Одно почтовое отделение размещается в деревне с номером i . Найдем место второго почтового отделения, дающего минимальную сумму расстояний. Пример (i равно 5) приведен на рис. 12.2. Первое почтовое отделение находится в деревне с номером 5. Вверху над точками проставлены расстояния, а «крестиками» обозначены деревни с почтовыми отделениями. Справа на рис. 12.2 приведены суммы расстояний до ближайших

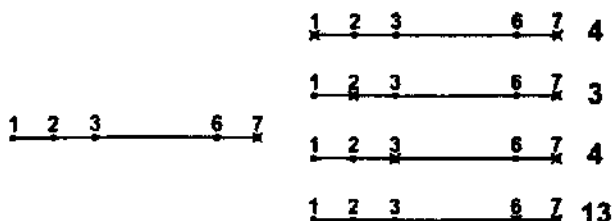


Рис. 12.2. Решение задачи для пяти деревень и двух почтовых отделений

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 166.

почтовых отделений. Из четырех этих вариантов мы выбираем второй, дающий минимальную оценку — 3.

Просчитаем то же самое для всех значений i . Массив оценок имеет вид, приведенный в табл. 12.12. Минимальная сумма расстояний до двух почтовых отделений (при условии, что одно почтовое отделение находится в деревне с номером i), приведено во второй строке табл. 12.12. Символом « ∞ » обозначен бессмысленный случай — два почтовых отделения в одной деревне.

Таблица 12.12

Номер деревни	1	2	3	4	5	6	7	8	9	10
Сумма расстояний	∞	0	1	2	3	7	12	21	37	43

Продолжим рассмотрение — пусть у нас теперь три почтовых отделения. На рис. 12.3 приведен случай при $i = 7$ (в 7 деревнях размещаются 3 почтовых отделения, где третье находится в деревне с номером 7). Справа первый отрезок характеризует вариант, при котором два почтовых отделения расположены в деревнях с 1-й по 6-ю. Он подсчитан (табл. 12.12) и равен 7. Третье отделение в седьмой деревне не изменяет итоговой оценки, — она по-прежнему равна 7. Мы почти «нащупали» динамическую схему! Второй отрезок — два почтовых отделения в деревнях с 1-й по 5-ю. Из табл. 12.12 видим, что стоимость этого случая равна 3, плюс расстояние от 6-й деревни до ближайшего почтового отделения 2, — итого 5.

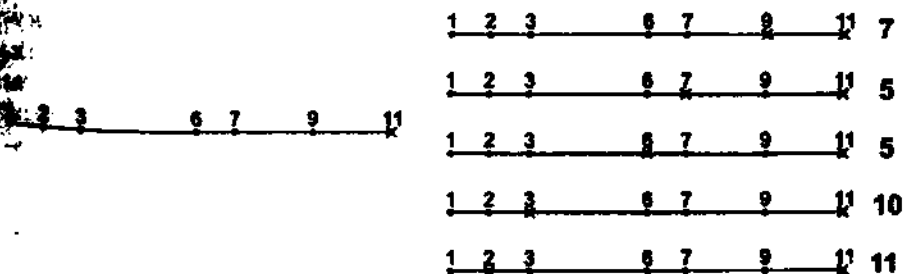


Рис. 12.3. Решение задачи для семи деревень и трех почтовых отделений

Окончательные оценки при трех, четырех и пяти отделениях приведены в табл. 12.13.

Таблица 12.13

Номер деревни	1	2	3	4	5	6	7	8	9	10
Оценка суммы расстояний	∞	∞	0	1	2	3	5	9	21	27
	∞	∞	∞	0	1	2	3	5	9	15
	∞	∞	∞	∞	0	1	2	3	5	9

Осталось сделать последний шаг. Оценка расстояний в нашей схеме проводилась до деревень, расположенных слева. К этим оценкам следует прибавить сумму расстояний до деревень, находящихся справа. Окончательные оценки суммы расстояний приведены в табл. 12.14.

Таблица 12.14

Номер деревни	1	2	3	4	5	6	7	8	9	10
Оценка суммы расстояний	∞	∞	∞	∞	101	92	85	53	11	9

Итак, общая сумма расстояний от каждой деревни до ее ближайшего почтового отделения равна 9. Это минимум в окончательном массиве оценок.

Как зафиксировать в этой схеме деревни с почтовыми отделениями? Как обычно, в алгоритмах типа Р. Флойда, при улучшении оценки, — запоминать соответствующую ситуацию.

Методическое отступление. Вооружившись микрокалькулятором, следует просчитать задачу на конкретных примерах. Полезным будет и ответ на вопрос о том, какие данные должны быть в двумерном массиве *way*, чтобы следующая процедура правильно выводила ответ задачи. Расстояния до деревень хранятся в массиве *r*[1..v].

```

Procedure OutWay(i,cnt:Integer);
Begin
    If cnt>1 Then OutWay(way[i,cnt],cnt-1);
    Write(r[i], ' ');
End;

```

Особенности реализации. Пусть начальные оценки (суммы расстояний до деревень, расположенных слева от деревни с но-

мером i , табл. 12.11) представлены в массиве *now*. Они вычисляются с помощью следующей конструкции:

```
For i:=2 To v Do
  For j:=1 To i-1 Do now[i]:=now[i]+Abs(r[i]-r[j]);
```

Как отразить работу, представленную в таблицах, в виде управляющих конструкций и структур данных? Напомним, что мы подсчитываем расстояния и делаем оценки только до деревень, расположенных слева от текущей (рассматриваемой) деревни.

Пусть задача решается для k почтовых отделений. Для меньшего количества она уже решена и оценки хранятся в массиве *now*. Запомним оценки в текущем массиве *lt* и зададим в *now* бесконечные начальные значения. Считаем, что последнее (k -е) почтовое отделение размещено в деревне с номером i ($2 \leq i \leq v$), а $i-1$ почтовое отделение размещено в деревнях с номерами от 1 до j , и наилучшая оценка для этого размещения уже получена, она является значением $lt[j]$. Предположим, что оценки сумм расстояний от деревень с номерами от $j+1$ до $i-1$ до ближайших почтовых отделений, расположенных в деревнях с номерами i и j , известны (подсчитываются один раз) — это элемент массива $A[i, j]$. Тогда новая оценка для k почтовых отделений, расположенных в деревнях с номерами от 1 до i , формируется путем выбора минимальной оценки для всех j от 1 до $i-1$. Формализованная запись изложенных соображений имеет вид:

```
For k:=2 To p Do Begin
  lt:=now; {Массивы lt и now имеют один тип}
  For i:=1 To v Do now[i] := ∞; {∞ — обозначение
    наибольшего числового значения в используемом
    типе данных}
  For i:=2 To v Do
    For j:=1 To i-1 Do
      If now[i]>A[i,j]+lt[j] Then now[i]:=A[i,j]+lt[j];
    {Если в этом месте добавить оператор way[i,k]:=j,
    то мы ответим на вопрос, заданный в методическом
    поступлении}
  End;
```

Матрица A формируется один раз и, естественно, до начала вышеприведенной логики. Приведем эти действия для полного изложения.

```

For i:=2 To v Do
  For j:=1 To v-1 Do Begin
    sum := 0;
    For t:=j+1 To i-1 Do
      If Abs(r[i]-r[t])>Abs(r[j]-r[t]) Then
        sum:=sum+Abs(r[j]-r[t])
      Else sum:=sum+Abs(r[i]-r[t]);
    A[i,j]:=sum;
  End;

```

После получения оценок в массиве *now* следует закончить его формирование путем прибавления расстояний до деревень, расположенных справа от текущей деревни *i*:

```

For i:=1 To v-1 Do
  For j:=i+1 To v Do now[i]:=now[i]+Abs(r[i]-r[j]);

```

Ответом задачи является минимальное значение в *low* и его место (*t*). Последнее значение необходимо для вывода деревень, в которых размещены почтовые отделения, с помощью процедуры *OutWay* — первый вызов *OutWay(t, p)*; при этом, естественно, используется сформированный в процессе работы массив *way*.

Методическое отступление. Здесь целесообразно в очередной раз обратиться к примерам и сделать просчет задачи, но уже с использованием представленных описаний фрагментов логики.

Временная сложность алгоритма — $O(p \cdot v^2)$, что составляет порядка $3 \cdot 10^6$ операций. При реализации задачи в среде программирования Турбо-Паскаль появляется проблема нехватки оперативной памяти, поэтому требуется использование указателей и хранение, по крайней мере, массива *A* в виде «кучи».

Методические рекомендации

Это — задача на динамическое программирование, достаточно сложная не только в обсуждении, но и в реализации. В качестве творческого задания, после того как школьником решено достаточное количество задач на динамическое программирование, она вполне применима.

Самостоятельные исследования

Обобщение задачи для ограниченной целочисленной решетки кажется естественным и порождающим, при соответствующей формулировке, интересное олимпиадное задание.

«Стены»

Условие задачи

В некоторой стране стены построены таким образом, что каждая стена соединяет ровно два города и стены не пересекают друг друга. Таким образом, страна разделена на отдельные области. Чтобы добраться из одной области в другую, необходимо либо пройти через город, либо пересечь стену. Два любых города a и b могут соединяться не более чем одной стеной (один конец стены в городе a , а другой — в городе b). Более того, всегда существует путь из города a в город b , проходящий вдоль каких-либо стен и через города.

Существует клуб, члены которого живут в городах. В каждом городе может жить либо один член клуба, либо вообще ни одного. Иногда у членов клуба возникает желание встретиться внутри одной из областей, но не в городе. Чтобы попасть в нужную область, каждый член клуба должен пересечь какое-то количество стен, возможно равное нулю. Члены клуба путешествуют на велосипедах. Они не хотят въезжать ни в один город из-за интенсивного движения на городских улицах. Они также хотят пересечь минимально возможное количество стен, так как пересекать стену с велосипедом довольно трудно. Исходя из этого, нужно найти такую оптимальную область, чтобы суммарное количество стен, которые требуется пересечь членам клуба, называемое суммой пересечений, было минимальным из всех возможных.

Города пронумерованы целыми числами от 1 до n , где n — количество городов. На рис. 12.4 пронумерованные вершины обозначают города, а линии, соединяющие вершины, обозначают стены. Предположим, что членами клуба являются три человека, которые живут в городах с номерами 3, 6 и 9. Тогда оптимальная область и соответствующие маршруты движения членов клуба показаны

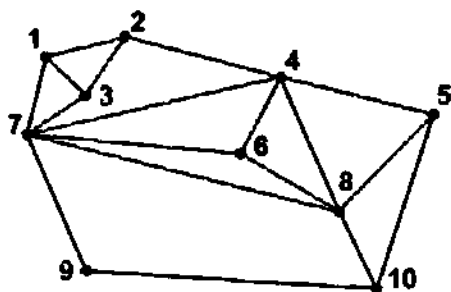


Рис. 12.4. Пример размещения городов и стен между ними

на рис. 12.5. Сумма пересечений равняется 2, так как членам клуба следует пересечь две стены: человеку из города 9 требуется пересечь стену между городами 2 и 4, человеку из города 6 требуется пересечь стену между городами 4 и 7, а человек из города 3 не пересекает стен вообще.

Требуется написать программу, которая по заданной информации о городах, областях и местах проживания членов клуба находит оптимальную область и минимальную сумму пересечений.

Входные данные. Входной файл имеет имя WALLS.IN. Первая строка содержит одно целое число: количество областей m , $2 \leq m \leq 200$. Вторая строка содержит одно целое число: количество городов n , $3 \leq n \leq 250$. Третья строка содержит одно целое число: количество членов клуба l , $1 \leq l \leq 30$, $l \leq n$. Четвертая строка содержит l различных целых чисел в возрастающем порядке: номера городов, где живут члены клуба.

Оставшаяся часть файла содержит $2m$ строк, по паре строк для каждой из m областей. Первые две строки из них описывают первую область, вторые две строки — вторую область и так далее. В каждой паре первая строка содержит количество городов i на границе области; вторая строка содержит номера этих i городов в порядке обхода границы области по часовой стрелке. Единственным исключением является последняя область — это находящаяся снаружи (внешняя) область, окружающая все города и другие области, и для нее порядок следования городов на границе задается против часовой стрелки. Порядок описания областей во входном файле задает целые номера этим областям. Область, описанная первой во входном файле, имеет номер 1, описанная второй — номер 2 и так далее. Обратите внимание, что входной файл содержит описание всех областей, образованных городами и стенами, включая находящуюся снаружи (внешнюю) область.

Выходные данные. Выходной файл имеет имя WALLS.OUT. Первая строка этого файла содержит одно целое число: минимальную сумму пересечений. Вторая строка содержит одно целое число: номер оптимальной области. Если существует несколько различных решений, вам необходимо выдать лишь одно из них.

Пример ввода и вывода. Приведенные входной и выходной файлы соответствуют рассмотренному в тексте примеру.

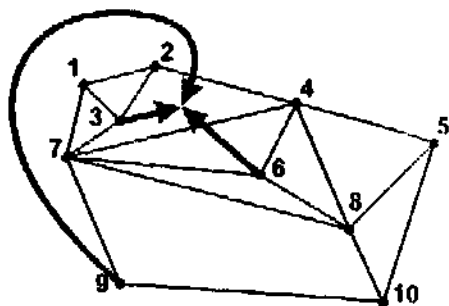


Рис. 12.5. Маршруты движения членов клуба, живущих в городах 3, 6 и 9

WALLS. IN:

10

10

3

3 6 9

3

1 2 3

3

1 3 7

4

2 4 7 3

3

4 6 7

3

4 8 6

3

6 8 7

3

4 5 8

4

7 8 10 9

3

5 10 8

7

7 9 10 5 4 2 1

WALLS OUT:

2

3

Обсуждение задачи

В формулировке задачи содержится явная подсказка о том, как ее решать, — что необычно для олимпиад по информатике. Задается граф, его описание, и очевидно, что следует искать решение, используя алгоритмы на графах.

Необходимо построить граф, вершинами которого являются области, а ребрами — стены между областями. Естественно, что должна быть вершина и для внешней области. Граф с номерами областей приведен на рис. 12.6.

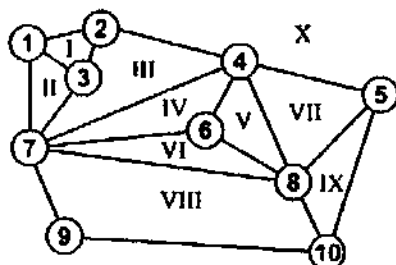


Рис. 12.6. Исходный граф.
Римскими цифрами указаны
номера областей

Построим граф областей. Он показан на рис. 12.7. Все расстояния между вершинами в графе равны единице. Используя алгоритм Р. Флойда*, подсчитаем расстояния между всеми парами вершин (матрица d) для графа на рис. 12.7. Они равны:

$$d = \begin{Bmatrix} 0 & 1 & 1 & 2 & 3 & 3 & 2 & 2 & 2 & 1 \\ 1 & 0 & 1 & 2 & 3 & 3 & 2 & 2 & 2 & 1 \\ 1 & 1 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 1 \\ 2 & 2 & 1 & 0 & 1 & 1 & 2 & 2 & 3 & 2 \\ 3 & 3 & 2 & 1 & 0 & 1 & 1 & 2 & 2 & 2 \\ 3 & 3 & 2 & 1 & 1 & 0 & 2 & 1 & 2 & 2 \\ 2 & 2 & 2 & 2 & 1 & 2 & 0 & 2 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 1 & 2 & 0 & 1 & 1 \\ 2 & 2 & 2 & 3 & 2 & 2 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 1 & 0 \end{Bmatrix}.$$

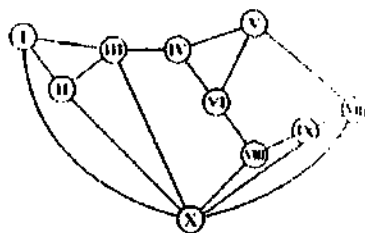


Рис. 12.7. Граф областей для примера из формулировки задачи

После этой работы остается только найти требуемую область. Логика этого поиска может иметь вид:

```
res:=∞;
For k:=1 To m Do Begin {Цикл по областям}
  rk:=0; {Рабочая переменная}
  For i:=1 To l Do Begin {Цикл по велосипедистам}
    rj:=∞; {Рабочая переменная}
    For j:=1 To m Do {Цикл по областям}
      If <город, в котором находится велосипедист,
      граничит с областью j> Then
        If rj>d[j,k] Then rj:=d[j,k]; {Выбираем граничную
        область с номером j, от которой до области k,
        количество пересечений (расстояние) минимальное}
        rk:=rk+rj; {Подсчитываем количество пересечений
        стен для области с номером k}
    End;
  If res>rk Then Begin
```

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 166.

```
res:=rk; {Количество пересечений}  
ires:=k; {Номер области}
```

```
End;
```

```
End;
```

В представленном фрагменте только одна проверка — «город, в котором находится велосипедист, граничит с областью j » — требует уточнения. Ее реализация зависит от того, как входные данные по каждой области представить в памяти компьютера. Если при вводе с каждым городом связать области, с ним граничащие, то потребуются лишь по индексам i и j проверить равенство нулю соответствующего элемента матрицы.

Методические указания

Данная задача требует конкретных знаний по алгоритмам на графах, а именно, алгоритма Р. Флойда. Кроме того, необходимо уметь преобразовывать одно описание графа в памяти компьютера в другое. Она может быть использована как обычная учебная задача.

Самостоятельные исследования

1. Задан ориентированный граф (V, E) . Дуге u множества E соответствует расстояние $c_u \geq 0$. Требуется найти все внешние центры графа, т. е. те вершины множества V , кратчайшее расстояние до которых от самой удаленной вершины минимально*.

Входные данные. Первая строка файла Input.txt содержит два целых числа: $1 \leq n \leq 150$ — количество вершин графа и $0 \leq m \leq 10000$ — количество его дуг. Последующие строки содержат тройки целых чисел: начало, конец и длину дуги (целое число в интервале от 0 до 100). Допускаются кратные дуги различной длины.

Выходные данные. Выходной файл Output.txt содержит количество внешних центров графа и номера соответствующих вершин в порядке возрастания.

Пример.

Input.txt:

```
3 4  
2 1 4  
3 1 5  
3 2 3  
1 2 5
```

* Воронин А. В., Кузнецов В. А., Корзун Д. Ж. Командные олимпиады по программированию: организация, задачи и решения. Петрозаводск: ПетрГУ, 2001. С. 178.

Output.txt:

2 1 2

Требуется написать генератор тестов и экспериментально исследовать, при каких ограничениях время работы программы превысит 1 секунду.

2. Имеется n городов, которые пронумерованы от 1 до n ($1 \leq n \leq 100$). Некоторые из них соединены двухсторонними дорогами, пересекающимися только в городах. Имеется два типа дорог — шоссейные и железнодорожные. Для каждой дороги известна базовая стоимость проезда по ней.

Необходимо проехать из города A в город B , уплатив минимальную сумму за проезд. Стоимость проезда зависит от набора проезжаемых дорог и от способа проезда. Так, если вы подъехали к городу C по шоссейной (железнодорожной) дороге $X \rightarrow C$ и хотите ехать дальше по дороге $C \rightarrow Y$ того же типа, то вы должны уплатить только базовую стоимость проезда по дороге $C \rightarrow Y$. Если же тип дороги $C \rightarrow Y$ отличен от типа дороги $X \rightarrow C$, то вы должны уплатить базовую стоимость проезда по дороге $C \rightarrow Y$ плюс 10% от базовой стоимости проезда по этой дороге (страховой взнос). При выезде из города A страховой взнос платится всегда*.

Требуется написать программу, которая находит самый дешевый маршрут проезда в виде последовательности городов и вычисляет стоимость проезда по этому маршруту.

Примечание. Требуется определить форматы входных и выходных данных, а также разработать системы тестов для проверки решения.

«Конструирование из блоков»

Условие задачи

Единичным кубом назовем куб $1 \times 1 \times 1$, вершины которого имеют целочисленные координаты x, y, z . Два единичных куба могут образовывать новый объект путем соединения их по грани. *Кубоидом* назовем непустое соединение единичных кубов (рис. 12.8). Объем кубоида равен количеству единичных кубов, из которых он составлен. *Блоком* назовем кубоид с объемом не больше четырех. Скажем, что два блока имеют одинаковый тип, если один может быть получен из другого с помощью вращений и сдвигов блока (заметьте, что отражения делать запре-

* Долинский М. С. Алгоритмизация и программирование на Turbo Pascal: от простых до олимпиадных задач. СПб.: Питер, 2005. С. 166.

щено). Всего существует 12 типов блоков (рис. 12.9). Цвета объектов на рисунке изображены лишь для иллюстрации структуры объектов, и они не несут никакой смысловой нагрузки.

Набор кубоидов d назовем *декомпозицией кубоида s* , если объединение всех кубоидов из d равно s , и никакой единичный куб из кубоида s не принадлежит одновременно двум разным кубоидам из d .

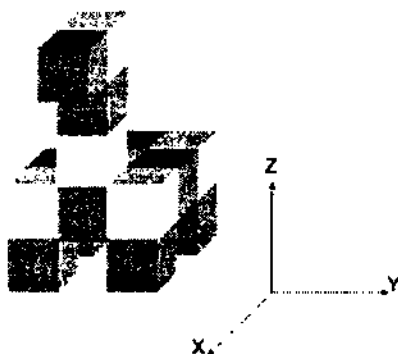


Рис. 12.8. Кубоид «Лошадь»

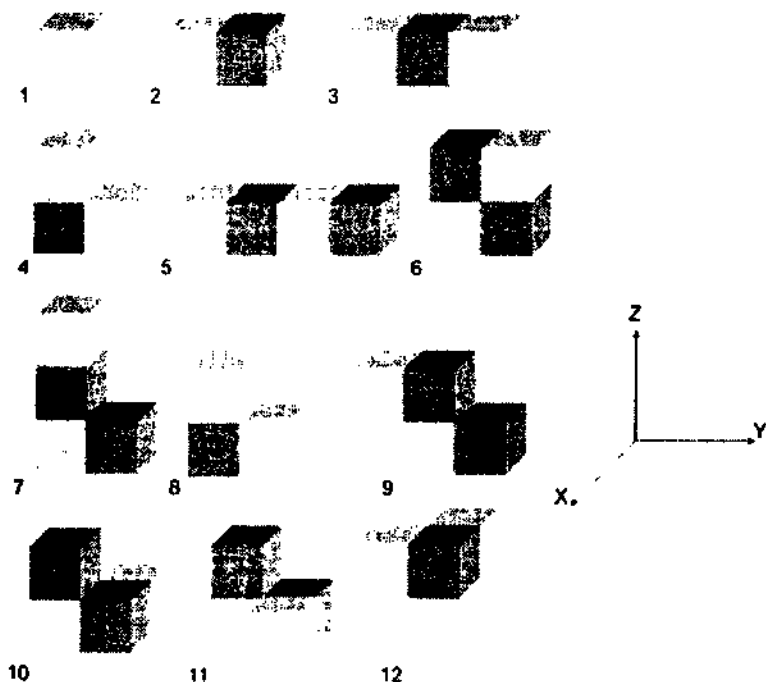


Рис. 12.9. Блоки всех 12 типов

Напишите программу, которая по заданному описанию типов блоков и кубоида s определяет наименьший по количеству элементов набор блоков, являющийся декомпозицией кубоида s . Вам нужно выдать только типы этих блоков. Каждый тип должен быть выведен столько раз, сколько блоков этого типа встречается в декомпозиции.

Входные данные. Положение единичного куба в пространстве будем задавать с помощью целочисленных координат x, y, z такой его вершины, для которой сумма $x+y+z$ минимальна.

Входной файл с описанием типов блоков имеет имя TYPES.IN. Содержимое этого файла одинаково для всех тестов. Файл содержит описание всех 12 блоков, изображенных на рис. 12.9, отсортированных по номеру типа. Каждый из блоков описывается группой идущих друг за другом строк следующим образом. Первая строка состоит из целого числа i , идентифицирующего тип блока ($1 \leq i \leq 12$). Вторая строка содержит объем v блока ($1 \leq v \leq 4$). Далее следуют v строк, описывающих положение единичных кубов, из которых состоит блок. Каждая из этих v строк состоит из трех целых чисел x, y, z ($1 \leq x, y, z \leq 4$).

Входной файл, описывающий кубоид, имеет имя BLOCK.IN. Первая строка файла содержит объем v кубоида ($1 \leq v \leq 50$). Следующие v строк описывают положение единичных кубов, из которых состоит кубоид. Каждая из этих v строк состоит из трех целых чисел x, y, z ($1 \leq x, y, z \leq 7$).

Выходные данные. Выходной файл имеет имя BLOCK.OUT. Первая строка файла содержит одно целое число m , которое равно количеству блоков в минимальном наборе, являющееся декомпозицией заданного кубоида. Вторая строка содержит список номеров типов блоков, содержащихся в наборе. Номера могут идти в любом порядке.

В случае существования нескольких решений выведите только одно из них.

Пример ввода и вывода. Файл TYPES.IN приведен на рис. 12.10, а файл BLOCK.IN — на рис. 12.11. Ответом задачи для данных из файла BLOCK.IN является информация файла BLOCK.OUT (рис. 12.12).

```

1
1
1 1 1
2
2
1 1 1
1 2 1
3
3
1 1 1
1 2 1
1 3 1
4
3
1 1 1
1 2 1
1 1 2
5
4
1 1 1
1 2 1
1 3 1
1 4 1
6
4
1 1 1
1 2 1
1 1 2
1 2 2
7
4
1 1 1
1 2 1
1 1 2
1 1 3
8
4
1 1 1
1 2 1
1 3 1
1 1 2
1 2 2
9
4
1 2 1
1 3 1
1 1 2
1 2 2
10
4
2 1 1
1 2 1
2 2 1
2 1 2
11
4
1 1 1
1 2 1
2 2 1
1 1 2
12
4
2 2 1
2 1 2
1 2 2
2 2 2

```

Рис. 12.10. Данные файла TYPES.IN

Замечание.

1. Входной файл BLOCK.IN описывает кубоид «Лошадь», изображенный на рис. 12.8.
2. Все остальные варианты содержимого второй строки выходного файла, описывающей типы используемых блоков, приведены ниже:

```

2  7  10  11  12
2  7  11  11  12
4  4   7  10  11
4  4   9  10  11

```

```

18
2 1 1
4 1 1
2 3 1
4 3 1
2 1 2
3 1 2
4 1 2
1 2 2
2 2 2
3 2 2
4 2 2
2 3 2
3 3 2
4 3 2
4 2 3
4 2 4
4 2 5
5 2 5

```

Идея решения

Решение состоит из трех частей: ввода, перебора вариантов и вывода результатов. При вводе из блока каждого типа путем поворотов строятся его модификации. Перебор вариантов укладки кубоида осуществляется как исходными блоками, так и их модификациями. При этом в кубоиде выбирается кубик с наименьшим количеством соседей (кубиков, имеющих общую грань). Затем мы получаем все блоки, в которых он может присутствовать, и исключаем используемый блок из дальнейшего рассмотрения; при этом корректируется количество соседей у оставшихся кубиков.

Рис. 12.11. Данные файла BLOCK.IN, соответствующие кубоиду «Лошадь»

```

5
7 10 2 10 2

```

Рис. 12.12. Данные файла BLOCK.OUT

Методическое отступление. Приведем здесь полный текст решения*, ибо наиболее вероятным способом изучения задачи является разбор программного кода.

```

{$DEFINE USETIME}
Program Task_Block;
Const
  types='types.in';
  block='block.in';
  out='block.out';
  CntDir=6; {Максимальное количество соседей}
  MaxTime=18*90; {Максимальное время работы}
  MaxBlock=12; {Количество различных типов блоков}
  MaxLen=4; {Максимальная длина блока}
  MaxCo=7; {Максимальное значение координаты}

```

* Приводится по программе, разработанной Дмитрием Шулятниковым.

```

MaxCntBlock=MaxBlock*30; {Общее количество
модификаций блоков}
MaxV=50; {Максимальное количество кубиков}
Dd:Array[1..3,1..6]Of ShortInt =((0,0,1,0,0,-1),
(0,1,0,0,-1,0), (1,0,0,-1,0,0)); {Приращения для
поиска соседей}

```

Type

```

TSS=Set Of 0..MaxV;
TPoint=Array[1..3] Of Integer; {Координаты кубика}
TDir=Array[1..CntDir] Of Byte; {Соседи: слева,
справа, снизу, сверху, впереди и сзади}

```

TPP=Record

```

    cc:TPoint; {Координаты}
    num:Integer; {Порядковый номер}
    dir:TDir; {Соседи}
    Nbl:Byte; {Номер блока, которому принадлежит кубик}
    CDir:Byte; {Количество соседей}

```

End;

TBlock=Record

```

    number, len:Integer;
    cc:Array[1..MaxLen] Of TPoint;

```

End;

```

TNum=Array[0..MaxLen] Of Integer;

```

Var

```

inf, ouf:Text;
AllBlk:Array[2..MaxLen,1..MaxCntBlock] Of TBlock;
{Все модификации блоков}
CntBlk:Array[1..MaxLen] Of Integer; {Количество
модификаций длины i}
Blk:Array[0..MaxV] Of TPP; {Кубоид}
Bst, V:Integer; {Количество блоков в наилучшем
решении}
BT, Now:Array[0..MaxV] Of TNum; {Лучшее и текущее
решения}
All:Array[-1..MaxCo+1, 0..MaxCo+1, -1..MaxCo+1]
Of Integer;

```

```

($IFDEF USETIME)

```

```

Timer:LongInt Absolute $0000:$046C;

```

```

Time:LongInt;

```

```

($ENDIF)

```

```

Function RavPoint(a,b:TPoint):Boolean; {Проверка
совпадения кубиков}

```

```

    Var i:Integer;

```

Begin

```

    RavPoint:=False;

```

```
For i:=1 To 3 Do
  If a[i]<>b[i] Then Exit;
RavPoint:=True;
End;
Function RavBlk(a,b:TBlock):Boolean; {Проверка
совпадения блоков}
Var i,j:Integer;
Begin
  RavBlk:=False;
  If a.len=b.len Then Begin
    For i:=1 To a.len Do Begin
      j:=1;
      While (j<=a.len) And
        Not RavPoint(a.cc[i],b.cc[j]) Do j:=j+1;
      If j=a.len+1 Then Exit;
    End;
    RavBlk:=True;
  End;
End;
Procedure PrepareBlk(Var b:TBlock); {"Приведение"
блока - уменьшение координат}
Var i, j, t:Integer;
    p:TPoint;
Begin
  With b Do
    For i:=1 To 3 Do Begin
      t:=MaxInt;
      For j:=1 To len Do
        If cc[j,i]<t Then t:=cc[j,i];
      For j:=1 To len Do cc[j,i]:=cc[j,i]-t;
    End;
  End;
Procedure AddBlk(B1:TBlock); {Добавление блока
в массив модификаций блоков}
Var i:Integer;
Begin
  With b1 Do Begin
    For i:=1 To CntBlk[len] Do
      If RavBlk(B1,AllBlk[len,i]) Then Exit;
    CntBlk[len]:=CntBlk[len]+1;
    AllBlk[len,CntBlk[len]]:=B1;
  End;
End;
Procedure RotateBlk(b:TBlock); {Выполняем все повороты
блока и "новые" блоки заносим в массив AddBlk}
```

```

Var i,j,k,w:Integer;
Procedure Rotate(x,y:Integer);
  Var i:Integer;
      p: TBlock;
  Begin
    p:=b;
    For i:=1 To b.len Do
      With p Do Begin
        cc[i,y]:=b.cc[i,x];
        cc[i,x]:=-b.cc[i,y];
      End;
      b:=p;
  End;
Begin
  For i:=1 To 4 Do Begin
    Rotate(1,2);
    For j:=1 To 4 Do Begin
      Rotate(1,3);
      For k:=1 To 4 Do Begin
        Rotate(2,3);
        PrepareBlk(b);
        AddBlk(b);
      End;
    End;
  End;
End;
Procedure Init; {Ввод данных. Вводятся описания
блоков и строятся их модификации с помощью поворотов.
Вводятся данные по кубоиду, и строится его описание}
  Var i,j,r:Integer;
      bl: TBlock;
  Begin
    {$IFDEF USETIME}
    Time:=Timer;
    {$ENDIF}
    Assign(inf, types);
    Reset(inf);
    <Элементам массивов All и CntBlk присваиваются
    нулевые значения>;
    For i:=1 To MaxBlock Do
      With bl Do Begin {Ввод описания блоков}
        Read(inf, number, len);
        For j:=1 To len Do
          Read(inf, cc[j,1], cc[j,2], cc[j,3]);
        If len>1 Then RotateBlk(bl);
      End;

```

```
Close(inf);
Assign(inf,block);
Reset(inf);
Read(inf,V);
For i:=1 To V Do
  With Blk[i] Do Begin {Ввод описания кубоида}
    Read(inf,cc[1],cc[2],cc[3]);
    All[cc[1],cc[2],cc[3]]:=i;
    NBl:=0;
    CDir:=0;
  End;
For i:=1 To V Do
  With Blk[i] Do
    For j:=1 To CntDir Do Begin {Определяем
количество "соседей" и их номера}
      Dir[j]:=All[cc[1]+dd[1,j],cc[2]+dd[2,j],
cc[3]+dd[3,j]];
      If Dir[j]<>0 Then CDir:=CDir+1;
    End;
  Close(inf);
End;
Function GetType(a:TNum):Integer; {Определяем тип
блока, так как перебор осуществляется не только
по исходным блокам, но и их модификациям.
Функция используется при выводе результата}
Var bb:TBlock;
    i:Integer;
Begin
  For i:=1 To a[0] Do
    bb.cc[i]:=Blk[a[i]].cc; {Переписываем данные
по блоку}
    bb.len:=a[0];
    PrepareBlk(bb); {Приведение блока}
    If bb.len = 1 Then GetType:=1
    Else Begin
      For i:=1 To CntBlk[bb.len] Do {Количество
модификаций блоков из определенного количества
кубиков, например трех}
        If RavBlk(bb,AllBlk[bb.len,i]) Then Begin
{Проверка на совпадение блоков}
          GetType:=AllBlk[bb.len,i].number;
          Exit;
        End;
      GetType:=0;
    End;
End;
```

```

End;
Procedure Done; {Вывод результата}
Var i:Integer;
Begin
  Assign(ouf, out);
  Rewrite(ouf);
  If Bst=MaxInt Then Begin
    WriteLn(ouf, V);
    For i:=1 To V-1 Do Write(ouf, 1, ' ');
    WriteLn(ouf);
  End
  Else Begin {Выводим лучший результат}
    WriteLn(ouf, Bst);
    For i:=1 To Bst Do Begin {Количество блоков}
      Write(ouf, GetType(BT[i])); {Номера блоков}
      If i<Bst Then Write(ouf, ' ');
    End;
  End;
  Close(ouf);
End;

Procedure AddNeib(num, dp:Integer); {Изменение
количества соседей}
Var i:Integer;
Begin
  For i:=1 To CntDir Do
    Inc(Blk[Blk[num].Dir[i]].CDir, dp);
End;

Procedure Rec(num, cnt, NotUse:Byte); {Параметрами
переборного решения являются: num — номер текущего
кубика, cnt — количество блоков в решении, NotUse —
количество неиспользованных кубиков кубоида}
Var i, j, t, ci, min:Byte;
Begin
  Now[cnt, 0]:=Now[cnt, 0]+1;
  Now[cnt, Now[cnt, 0]]:=num;
  Blk[num].Nbl:=cnt; {Помечаем кубик
как задействованный}
  AddNeib(num, -1); {Уменьшаем на 1 количество
соседей}
  {$IFDEF USETIME}
  If Timer-Time>=MaxTime Then Done; {Отсечение
по времени решения}
  {$ENDIF}

```

If cnt+((NotUse+MaxLen-1) Div MaxLen)<=Bst Then
 {Отсечение. Если количество блоков в решении,
 плюс количество возможных дополнений, меньше лучшего
 из найденных решений, то продолжаем эту "ветку"
 перебора}

If NotUse=0 Then Begin Bst:=cnt; BT:=Now End
 Else Begin

If Now[cnt,0] In [1..MaxLen-1] Then

For i:=1 To Now[cnt,0] Do

For j:=1 To CntDir Do Begin

{Цикл по соседям}

t:=Blk[Now[cnt,i]].Dir[j];

If (Blk[t].NBl=0) And (t0) Then

Rec(t,cnt,NotUse-1);

End;

min:=CntDir+1;

{Включаем в решение новый блок}

For i:=1 To V Do

With Blk[i] Do {Выбираем кубик кубоида

{с минимальным количеством соседей}

If (NBl=0) And (CDir<min) Then Begin

min:=CDir;

ci:=I;

End;

Rec(ci,cnt+1,NotUse-1);

End;

AddNeib(num, 1); {Выход из рекурсии.

Восстанавливаем количество соседей}

Blk[num].NBl:=0;

Now[cnt,0]:= Now[cnt,0]-1;

End;

Procedure Solve;

Begin

Bst:=MaxInt;

<Элементам массива Now присвоить нулевые
 значения>;

Now[0,0] := MaxLen-1;

Rec(0,0,V);

End;

Begin

Init;

Solve;

Done;

End.

Методические указания

Это задача на «Перебор с возвратом». Требуется найти все укладки кубоида блоками и выбрать минимальную укладку по количеству блоков. С логической точки зрения она проста, но с точки зрения программной реализации — сложна. Сложность заключается в выборе таких структур данных для описания задачи, чтобы программный код был простым и ясным.

Задача трудна в изложении и объяснении, использовать ее можно только при целенаправленной подготовке к олимпиадам как рассмотрение одного из возможных типов заданий.

Самостоятельные исследования

Дан кубоид. Блок — это любая комбинация t кубиков ($t > 0$ и $t \leq 4$). Определите наименьшее количество блоков и их типы, необходимые для «сборки» кубоида.

ЗАДАЧИ ПЕРВОГО ТУРА**«Мобильные телефоны»****Условие задачи**

Предположим, что в регионе Тампере базовые станции обеспечения мобильной телефонной связи четвертого поколения действуют следующим образом. Регион поделен на квадраты. Квадраты образуют матрицу размером $n \times n$, строки и столбцы которой пронумерованы от 0 до $n-1$. В каждом квадрате находится базовая станция. Количество работающих мобильных телефонов внутри квадрата может меняться, так как телефоны могут перемещаться из одного квадрата в другой и телефоны могут включаться или выключаться. В некоторые моменты времени каждая базовая станция передает головной базовой станции отчет об изменении количества работающих телефонов в свои координаты (номер строки и номер столбца, соответственно).

Напишите программу, которая получает эти отчеты и отвечает на запросы о текущем общем количестве работающих мобильных телефонов в некоторой прямоугольной области.

Ввод и вывод. Целочисленные данные вводятся из стандартного входного потока. Входные данные кодируются следующим образом. Каждая строка содержит одну команду. Команда состоит из кода и набора параметров (целых чисел) в соответствии со следующей табл. 13.1.

Таблица 13.1

Команда	Параметры	Значение
0	n	Инициализирует матрицу размера $n \times n$ нулями. Эта команда выдается только один раз и должна быть первой командой
1	$x \ y \ a$	Прибавляет к количеству работающих мобильных телефонов в квадрате (x, y) число a . Число a может быть как положительным, так и отрицательным

Команда	Параметры	Значение
2	<i>l b r t</i>	Запрашивает текущее суммарное количество работающих мобильных телефонов в квадратах (x, y) , где $l \leq x \leq r, b \leq y \leq t$
3		Завершает программу. Эта команда выдается только один раз и должна быть последней

Программа отвечает на каждый запрос (команду с кодом 2), записывая целые числа в стандартный выходной поток.

Значения всегда будут в допустимых пределах, так что нет необходимости их проверять. В частности, добавление отрицательного числа a не приведет к уменьшению количества телефонов в квадрате до значения, меньшего нуля. Индексы в матрице начинаются от 0, например, для матрицы 44 мы имеем $0 \leq x \leq 3$ и $0 \leq y \leq 3$.

Ваша программа не должна ничего выдавать в ответ на команды, отличные от команд с кодом 2. Если поступила команда 2, то ваша программа должна ответить на запрос, выдав в стандартный выходной поток строку, содержащую единственное целое число.

Инструкции для программирования. Далее в примерах целочисленная переменная *last* предполагается последней, считанной из строки, а целочисленная переменная *answer* содержит ваш ответ.

Если вы пишете программу на C++ и используете *iostreams*, то вам следует применять следующие инструкции для чтения из стандартного входного потока и записи в стандартный выходной поток:

```
cin>>last;
cout<<answer<<endl<<flush;
```

Если вы пишете программу на C или C++ и используете *scanf* и *printf*, то вам следует применять следующие инструкции для чтения из стандартного входного потока и записи в стандартный выходной поток:

```
scanf("%d", &last);
printf("%d\n", answer); fflush (stdout);
```

Если вы пишете программу на *Pascal*, то вам следует использовать следующие инструкции для чтения из стандартного входного потока и записи в стандартный выходной поток:

```
Read(last); ... Readln;
Writeln(answer);
```

Пример:

stdin	stdout	Пояснение
0 4		Инициализируется матрица размером 4×4
1 1 2 3		Значение в квадрате (1, 2) увеличивается на 3
2 0 0 2 2		Запрашивается текущая сумма значений из прямоугольника $0 \leq x \leq 2, 0 \leq y \leq 2$
	3	Ответ на запрос
1 1 1 2		Значение в квадрате (1, 1) увеличивается на 2
1 1 2 -1		Значение в квадрате (1, 2) уменьшается на 1
2 1 1 2 3		Запрашивается текущая сумма значений из прямоугольника $1 \leq x \leq 2, 1 \leq y \leq 3$
	4	Ответ на запрос
3		Программа завершается

Ограничения приведены в табл. 13.2.

Таблица 13.2

Размер матрицы	$n \times n$	$1 \times 1 \leq n \times n \leq 1024 \times 1024$
В любом квадрате матрицы в каждый момент времени может находиться v работающих телефонов	v	$0 \leq v \leq 2^{15} - 1$ (= 32767)
Изменение количества телефонов в квадрате матрицы	a	$-2^{15} \leq a \leq 2^{15} - 1$ (= 32767)
Количество команд во входном потоке	u	$3 \leq u \leq 60002$
Максимальное суммарное количество телефонов во всей матрице	m	$m = 2^{30}$

В 16 из 20 тестов размер матрицы будет не более 512×512.

Замечание. При тестировании вашей программы через web указанный вами входной файл подается в стандартный входной поток вашей программы.

Обсуждение задачи

Рассмотрим задачу для одномерного случая*. Все изменения происходят на прямой линии, координаты операций — целочисленные, изменения мы будем называть *событием*, а требование о количестве мобильных телефонов на интервале — *запросом*.

Первый вариант решения. Для хранения количества мобильных телефонов используем массив A . При событиях следует изменить соответствующий элемент массива A ; при запросах — подсчитать сумму элементов массива A при заданных значениях индексов элементов. Для наглядности рассуждений приведем текст логики:

```
Procedure UpDate (i,t:Integer);
Begin
  A[i]:=A[i]+t;
End;
Procedure Report(i,j:Integer);
  Var k:Integer;
      sum:LongInt;
Begin
  sum:=0;
  For k:=i To j Do sum:=sum+A[k];
  WriteLn(sum);
End;
```

«Прозрачность» такого решения восхищает, но нас она не устраивает, ибо время обработки запроса линейно $O(n)$. Из ограничений по времени (1 секунда на тест) следует необходимость поиска более эффективной схемы.

Время обработки запросов можно уменьшить, если хранить в A суммы элементов отрезков массива. Измененный вариант логики:

```
Procedure UpDate (i,t:Integer);
  Var k:Integer;
Begin
  For k:=i To n Do A[k]:=A[k]+t;
End;
Procedure Report(i,j:Integer);
  Var sum:LongInt;
Begin
  sum:=A[j]-A[i-1];
  WriteLn(sum);
End;
```

* Шарыгин Р. В. Об одной задаче поиска данных // Вестник ВятГГУ. Информатика. 2003. № 2. С. 230–233.

Как говорится, «в одном месте прибыло, во втором — убыло». Запросы в этой модификации решения обрабатываются быстро, но изменения — медленно. Истина, как всегда, лежит где-то посередине, и ее требуется найти. Причем возможный, и, вероятно, единственный путь здесь — усложнение структур данных!

Методическое отступление. В этом месте обсуждения целесообразно, на наш взгляд, отвлечься и сделать почти философское рассуждение о красоте как о гармоничном сочетании частей, составляющих нечто единое. А закончить рассуждение можно мыслью классика и основателя образовательной информатики Никлауса Вирта о том, что программа есть гармоничное сочетание структур данных и управляющих конструкций.

Второй вариант решения. Давайте хранить в массиве A не значения и не сумму элементов на всех предшествующих отсчетах прямой, а нечто среднее — сумму некоторых изменений. Другими словами, значением элемента массива A будет некая частичная сумма, которую можно быстро изменить и на основе которой можно быстро «собрать» результирующее значение запроса. Так как дополнительных структур данных мы не вводим, то единственным ресурсом для наших действий является индекс элемента массива A . Сам по себе индекс мало что дает, но из его двоичного представления «выжимается» необходимый ресурс для решения проблемы.

Пусть $n = 8$ и события происходят последовательно: 1, 2, 3, ..., 8, т. е. последовательно изменяются значения $A[1], A[2], \dots, A[8]$. Если бы хранение информации осуществлялось по обычной схеме, то массив A выглядел бы так, как показано в верхней части рис. 13.1. А фактический его вид приведен в нижней части рис. 13.1.

Методическое отступление. Мы пока не разъясняем действий, а только их показываем.

Пусть поступило событие: 1 5 8. Пятый элемент массива изменяется на восьмой, тип команды — первый. Но мы изменяем не только пятый элемент, но также и шестой с восьмым. Схема изменений приведена на рис. 13.2.

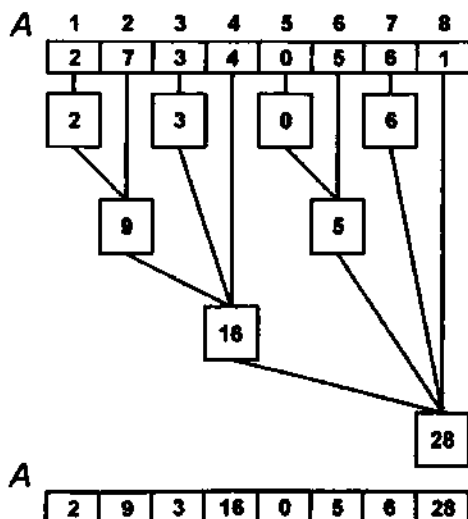


Рис. 13.1. Представление данных о событиях

Заметим, что здесь изменено только три элемента, а не четыре, как в первом варианте решения! (Мы опять умалчиваем о том, почему мы так делаем.)

Пусть теперь после произведенного изменения поступил запрос: 2 3 7 — требуется подсчитать количество мобильных телефонов в интервале [3, 7], тип команды — второй. Схема подсчета приведена на рис. 13.3. Заметьте: мы сделали не пять операций сложения, а только четыре! При небольших значениях n разница не существенна, однако при увеличении n эффект будет ощутимым.

Методическое отступление. Итак, у нас три рисунка. Можно теперь привести вариант логики обработки. Единственной «загадкой» в ее тексте является функция `Dispose(i)`. Здесь следует сделать паузу и дать возможность школьнику самому найти решение, чтобы это мини-открытие принадлежало ему*.

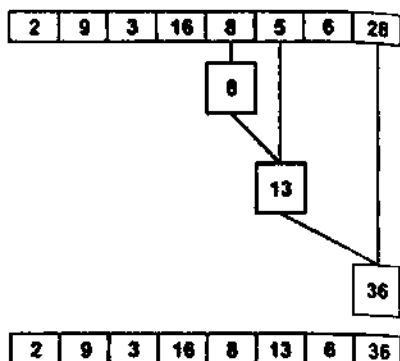


Рис. 13.2. Обработка события 1 5 8

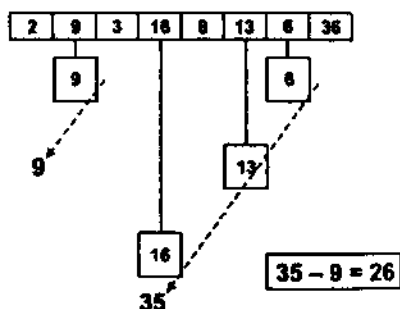


Рис. 13.3. Обработка запроса 2 3 7

```

Procedure UpDate (i,t:Integer);
Begin
  While i<=n Do Begin
    A[i]:=A[i]+t;
    i:=i+Dispose(i);
  End;
End;

Procedure Report(i,j:Integer);
Var sum:LongInt;
Begin
  sum:=0;
  While j>=1 Do Begin
    sum:=sum+A[j];
  
```

* Окулов С. М. Информатика: развитие интеллекта школьника. М.: БИНОМ. Лаборатория знаний, 2005. С. 91.

```
    j:=j-Dispose(j);  
End;  
i:=i-1;  
While i>=1 Do Begin  
    sum:=sum-A[i];  
    i:=i-Dispose(i);  
End;  
WriteLn(sum);  
End;
```

Возможны различные способы реализации функции Dispose, и поиск способов ее реализации опять же может быть предметом дискуссии. Приводится, например, следующий вариант:

```
Function Dispose(i:Integer):Integer;  
    Var q:Integer;  
Begin  
    q:=0;  
    While (i>0) And (i Mod 2=0) Do Begin  
        q:=q+1;  
        i:=i Div 2;  
    End;  
    Dispose:=1 ShL q;  
End;
```

А затем находится один из лучших способов*:

```
Function Dispose(i:LongInt):LongInt;  
    Begin  
        Dispose:=i And (i Xor (i-1));  
    End;
```

Заметим, что задача поиска количества единиц в двоичном представлении числа («созвучная» с рассматриваемым вопросом) может быть темой отдельного занятия. Материал для него есть в первой главе замечательной, но достаточно сложной книги Э. Рейнгольда и др**.

Продолжим обсуждение задачи. Итак, значением элемента массива $A[i]$ является сумма элементов из отрезка $[i-2^k+1, i]$, где k — количество нулей в младших разрядах (до первой справа единицы) двоичного представления числа i . И мы, например, видим из рис. 13.1, что для нечетных i ($k=0$) значение элемента A совпадает с характеристикой события.

* Андреева Е. В. Решения задач XIII международной олимпиады. Мобильные телефоны // «Информатика», еженедельное приложение к газете «Первое сентября». 2001. № 37. С. 19-21.

** Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы: теория и практика. М.: Мир, 1980. С. 12-17.

Пусть имеется событие 1 5 8 (рис. 13.2). Мы изменили $A[5]$, а затем — значение $5_{10} = 101_2$, $k = 0$, $2^k = 1$, $5+1 = 6$ — изменяем $A[6]$. Продолжим: $6_{10} = 110_2$, $k = 1$, $2^k = 2$, $6+2 = 8$. Изменяем элемент $A[8]$.

Пусть поступил запрос 2 3 7 (рис. 13.3). Берем элемент $A[7]$. Смотрим: $7_{10} = 111_2$, $k = 0$, $2^k = 1$, $7-1 = 6$, — добавляем к результату элемент $A[6]$. Далее: $6_{10} = 110_2$, $k = 1$, $2^k = 2$, $6-2 = 4$, — прибавляем к результату элемент $A[4]$. Затем: $4_{10} = 100_2$, $k = 2$, $2^k = 4$, $4-4 = 0$. Обработка закончена. Аналогичные действия выполняем и для второго индекса: берем $A[3-1]$, а затем $2_{10} = 10_2$, $k = 1$, $2^k = 2$, $2-2 = 0$. Обработка закончена.

После проделанной работы у учащихся уже есть готовность к пониманию исходной задачи. Как методически грамотно сделать переход на двумерный случай? На наш взгляд, следует идти от простого к сложному.

Пусть $s = 4$ и пока на обработку поступают одни события. Они, при первом варианте решения, фиксируются в двумерном

массиве $A[0..n-1, 0..n-1]$. Пусть, например, $A = \begin{pmatrix} 7 & 2 & 1 & 6 \\ 9 & 3 & 2 & 4 \\ 1 & 5 & 7 & 3 \\ 5 & 8 & 0 & 2 \end{pmatrix}$.

Очередность поступления — 7, 2, 1, 6, 9, 3 и т. д. до значений 8, 0, 2. Рассмотрим наш вариант хранения частичных сумм при той же очередности поступления событий:

$$\begin{pmatrix} 7 & 7 & 0 & 7 \\ 7 & 7 & 0 & 7 \\ 0 & 0 & 0 & 0 \\ 7 & 7 & 0 & 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 0 & 9 \\ 7 & 9 & 0 & 9 \\ 0 & 0 & 0 & 0 \\ 7 & 9 & 0 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 10 \\ 7 & 9 & 1 & 10 \\ 0 & 0 & 0 & 0 \\ 7 & 9 & 1 & 10 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 7 & 9 & 1 & 16 \\ 0 & 0 & 0 & 0 \\ 7 & 9 & 1 & 16 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 18 & 1 & 25 \\ 0 & 0 & 0 & 0 \\ 16 & 18 & 1 & 25 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 1 & 28 \\ 0 & 0 & 0 & 0 \\ 16 & 21 & 1 & 28 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 30 \\ 0 & 0 & 0 & 0 \\ 16 & 21 & 3 & 30 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 0 & 0 & 0 & 0 \\ 16 & 21 & 3 & 34 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 1 & 0 & 1 \\ 17 & 22 & 3 & 35 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 0 & 6 \\ 17 & 27 & 3 & 40 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 7 & 13 \\ 17 & 27 & 10 & 47 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 7 & 16 \\ 17 & 27 & 10 & 50 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 7 & 16 \\ 22 & 32 & 10 & 55 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 7 & 16 \\ 22 & 40 & 10 & 63 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 7 & 16 \\ 22 & 40 & 10 & 63 \end{pmatrix} \Rightarrow \begin{pmatrix} 7 & 9 & 1 & 16 \\ 16 & 21 & 3 & 34 \\ 1 & 6 & 7 & 16 \\ 22 & 40 & 10 & 65 \end{pmatrix}.$$

Мы видим, что поступление события изменяет тот элемент («квадрат» — по формулировке задачи) матрицы A , к которому оно относится. А затем осуществляется изменение элементов в строке в соответствии с логикой одномерного случая, и далее процесс продолжается по следующим строкам, опять же в соответствии с той же логикой. Заметим, что очередность поступления событий никак не влияет на конечный результат. Она оговорена для простоты и наглядности изложения.

Перейдем к обработке запросов. Пусть требуется подсчитать количество мобильных телефонов в прямоугольнике с координатами противоположных углов (i_1, j_1) и (i_2, j_2) — обозначим его как $z[(i_1, j_1), (i_2, j_2)]$. Подсчитаем количество телефонов в прямоугольниках $s_1[(0, 0), (i_2, j_2)]$, $s_2[(0, 0), (i_1-1, j_2)]$, $s_3[(0, 0), (i_2, j_1-1)]$ и $s_4[(0, 0), (i_1-1, j_1-1)]$. Результат равен $s_1 - s_2 - s_3 + s_4$. Изменение индексов i и j при этом осуществляется по схеме, рассмотренной для упрощенной задачи на прямой. Так, при запросе 2 2 3 3 для нашего примера получаем: $65 - 34 - 40 + 21 = 12$.

Методическое отступление. Казалось бы, теперь можно закончить обсуждение задачи и «вынести» ее на практическую реализацию. Однако всегда находятся ученики, у которых, несмотря на все усилия, момент озарения (возникновение собственной мысли) еще не возник. Ничего страшного в этом нет, ибо как возникает это состояние — неизвестно, и что «задержало» его возникновение в данный момент, нам не выяснить. Мы лишь создаем и поддерживаем среду усилий* для каждого интеллекта по пребыванию его в состоянии мыслительного акта, но само пребывание — это уже дело самого интеллекта.

В данной ситуации целесообразно, на наш взгляд, использовать известный методический прием — дать «набросок» логики и попросить выполнить его «ручную» трассировку при конкретных исходных данных. С этой целью приведем один из возможных вариантов реализации.

Обработка события (фрагмент):

```
...
Read(i, j, t);
i:=i+1;
j:=j+1;
While j<=n Do Begin
  ix:=i;
  While ix<=n Do Begin
```

```

    A[ix-1,j-1]:=A[ix-1,j-1]+t;
    ix:=ix+Dispose(ix);
  End;
  j:=j+Dispose(j);
End;
...

```

Обработка запроса (фрагмент):

```

...
s:=0;
i:=i+1;
j:=j+1;
While j>0 Do Begin
  ix:=i;
  While ix>0 Do Begin
    s:=s+ A[ix-1,j-1];
    ix:=ix-Dispose(ix);
  End;
  j:=j-Dispose(j);
End;
...

```

Методические рекомендации

Потенциал данной задачи с методической точки зрения значителен! В содержательном плане она охватывает темы: «Представление чисел в памяти компьютера», «Организация поиска данных», «Абстрактные типы данных» (используемую структуру данных можно рассматривать как некую модификацию двоичного дерева). С точки зрения же обучения решению задачи она — наглядный пример реализации принципа «от простого к сложному».

Самостоятельные исследования

1. Разработайте систему тестов и определите, при какой размерности входных данных время решения задачи по первому и второму вариантам будет отличаться более чем на 1 секунду.
2. Обобщение этой задачи на трехмерный случай очевидно. Её интерпретация в виде описания некой жизненной проблемы дает хорошую олимпиадную трактовку проблемы. Придумайте задачу для трехмерного случая, решите ее и разработайте систему тестов, проверяющих все возможные ветви логики решения.

«Игра *ioiware*»

Условие задачи

Игра с бусинками и лунками является одной из старейших форм развлечения. В этой задаче приводится вариант *ioiware* подобной игры, специально разработанный для международной олимпиады по информатике.

В игру играют два игрока. Для игры используется доска с семью лунками, расположенными по кругу. Кроме того, у каждого игрока есть свой банк, изначально пустой. В начале игры 20 бусинок случайно распределяются по всем семи лункам так, что в каждую лунку помещается от 2 до 4 бусинок (не менее 2 и не более 4). Два игрока ходят по очереди. Во время своего хода игрок выбирает любую непустую лунку, забирает из нее все бусинки в руку и выполняет ряд действий, пока в его руке остается хотя бы одна бусинка. Начиная с лунки, расположенной сразу за той, из которой он забрал бусинки в начале хода, он последовательно переходит по часовой стрелке от одной текущей лунки к другой, выполняя следующие операции:

- если в руке игрока более одной бусинки, тогда, если текущая лунка уже содержит 5 бусинок, то игрок вынимает из нее одну бусинку и перемещает в свой банк; если же в текущей лунке менее пяти бусинок, то игрок помещает в нее одну бусинку из своей руки;
- если же в руке игрока ровно одна бусинка, тогда, если текущая лунка содержит от одной до четырех бусинок, то все бусинки из этой лунки и единственная бусинка из руки игрока помещаются в его банк; в противном случае (то есть текущая лунка содержит 0 или 5 бусинок) бусинка из руки игрока перемещается в банк противника.

Ход считается сделанным, когда у игрока в руке не останется ни одной бусинки.

Игра закончится, когда по окончании очередного хода все лунки окажутся пустыми. Победителем считается тот из игроков, в банке которого будет больше бусинок.

Первый игрок всегда имеет выигрышную стратегию. Вы должны написать программу, которая играет в игру *ioiware* за первого игрока и выигрывает. Учтите, что второй игрок будет играть оптимально, то есть при первом же неправильном ходе со стороны вашей программы она не выигрывает.

Ввод и вывод. Ваша программа читает входные данные из стандартного потока ввода и пишет выходные данные в стандартный поток вывода. Ваша программа играет за первого игрока, а противник — за второго. В начале работы ваша про-

некую упорядоченность (меру порядка) в этом «пространстве состояний». Другими словами, нужно установить для каждого состояния, какое ему предшествует и какое за ним следует.

Пример. Семь лунок — это много. Пусть у нас будет две лунки и пять бусинок. В первой строке входных данных мы определим n (количество лунок), m (количество бусинок, общее), k (максимальное значение количества бусинок в лунке), а во второй строке — распределение бусинок по лункам. Итак, мы имеем:

```
2 5 5
2 3
```

Не поленимся и выпишем все возможные состояния лунок: (0,0), (0,1), (0,2), (0,3), (0,4), (0,5), (1,0), (1,1), (1,2), (1,3), (1,4), (1,5), (2,0), (2,1), (2,2), (2,3). Всего таких состояний шестнадцать, и мы можем их пронумеровать числами от 0 до 15, а описывать в памяти компьютера будем одномерным массивом $d[0..DMax]$, где $DMax$ — максимально возможное количество состояний игры.

Первая возникающая мысль (и она правильная!) — о том, что это ни что иное, как пара цифр чисел в системе счисления с основанием $k + 1$ ($Base = k + 1$). А если у нас n лунок, то в круглых скобках будет n цифр, n -значное число в этой системе счисления. Возникает чисто технический вопрос об определении номера, соответствующего состоянию лунок (элемента в «пространстве состояний»), и об обратном преобразовании. Для наглядности изложения приведем их.

Перевод номера в состояние:

Procedure GetState(a:Integer;Var s:TSt); {Тип данных — TSt=Array [1..NMax] Of Integer, где NMax — максимальное количество лунок}

Var i:Integer;

Begin

For i:=1 To n Do s[i]:=0; {Массивом s описывается состояние лунок}

For i:=n DownTo 1 Do Begin

s[i]:=a Mod Base;

a:=a Div Base;

End;

End;

Перевод состояния в номер:

Function GetCode(s:TSt):Integer;

Var i,a:Integer;

Begin

 a:=0;

 For i:=1 To n Do a:=a*Base+s[i];

 GetCode:=a;

End;

Представим игру для рассматриваемого примера в виде диаграммы (рис. 13.4). Верхняя цифра в кружках — это номер состояния игры, цифры в угловых скобках — состояние лунок. Возле стрелок указан номер игрока, делающего ход.

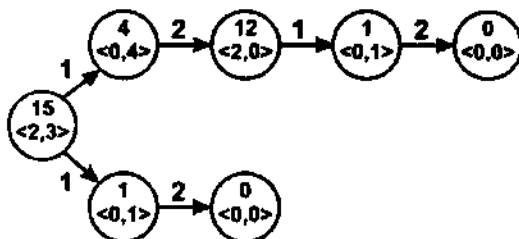


Рис. 13.4. Диаграмма игры для простого примера

Исходное состояние — 15. Первый игрок может взять две бусинки из первой лунки. По правилам игры мы переходим в состояние 4 — в первой лунке нет бусинок (нуль), во второй — четыре. У второго игрока только один ход, переводящий игру в состояние 12, затем первый игрок переводит в состояние 1, и, наконец, второй — в конечное состояние 0. Эта ветвь игры приводит к проигрышу первого игрока с результатом 1:4. Второй возможный начальный ход первого игрока — взять три бусинки из второй лунки. Этот ход переводит игру в состояние 1, и она заканчивается следующим ходом второго игрока с результатом 5:0.

Методическое отступление. Предложение о прорисовке диаграммы игры для небольших значений n и m будет вполне уместным.

Рассмотрим некоторые особенности реализации. Логика выполнения хода игры прописана в формулировке задачи. Она проста, и приводить ее текст нет смысла. Нам достаточно описать параметры процедуры `Procedure Move(s:TSt; a:Integer; Var g1,g2:integer; Var s_new:TSt)`. В массиве s представлено состояние игры, в переменной a — номер лунки, из которой берутся бусинки, в переменных $g1$ и $g2$ — количество бусинок, получаемых игроками в результате хода, и наконец, в массиве s_new — новое состояние игры.

Удержаться от того, чтобы не привести основную программу из одного оператора `WriteLn(Calc(t))` трудно (значением t является номер начального состояния игры). Конечно, требует пояснений функция `Calc`, но ее разъяснение несложно, ибо она

пишется по принципу «как говорю, так и пишу», что характеризует достигнутый синтез структур данных и управляющих конструкций (а, по утверждению классика образовательной информатики Никлауса Вирта, это и есть программа):

```
Function Calc(a:Integer):Integer; {Значением a
является номер состояния игры}
Var state,nstate:TSt; {Тип данных TSt описан ранее}
    s,i,g1,g2:Integer;
Begin
  If d[a]=-1 Then Begin {Пространство состояний
задачи описывается массивом d. Начальные значения
элементов массива равны -1, а затем мы храним
максимальное значение выигрыша первого игрока
в этом состоянии}
    d[a]:=0; s:=0;
    GetState(a,state); {По номеру состояния
определяем количество бусинок в лунках}
    For i:=1 To n Do s:=s+state[i]; {n – количество
лунок. Подсчитываем количество бусинок в лунках}
    For i:=1 To n Do {Перебор по лункам}
      If state[i]<>0 Then Begin {В лунке есть бусинки?}
        Move(state,i,g1,g2,nstate); {Делаем ход}
        d[a]:=Max(d[a],(s-g1-g2)-Calc(GetCode(nstate))+g1);
        {Рекурсивный вызов Calc дает максимальное количество
бусинок, которые может получить второй игрок
в процессе игры из состояния nstate}
      End;
    End;
  End;
  Calc:=d[a];
End;
```

Методические указания

Эта задача содержательна с точки зрения обсуждения таких фундаментальных понятий информатики, как конечная величина, управление, структуризация, отношение порядка и перебор вариантов. Их определение и обоснование приводится в книге одного из авторов*. Если же приведенного обсуждения окажется недостаточно, то можно воспользоваться текстом программного решения, ориентированного в точности на формулировку задачи**.

* Окулов С. М. Информатика: развитие интеллекта школьника. М.: БИНОМ. Лаборатория знаний, 2005. С. 52–58.

** Андреева Е. В. Решения задач XIII международной олимпиады. Игра *ioiwar* // «Информатика», еженедельное приложение к газете «Первое сентября». 2001. № 40. С. 23–24.

Самостоятельные исследования

1. Одноременные монетки разложены в стопки (в стопках может быть различное количество монет), а стопки поставлены в ряд слева направо. Двое противников по очереди делают ходы. Ход состоит в том, что один из игроков берет слева несколько стопок подряд (не меньше одной, но и не больше, чем перед этим взял его соперник). Первый игрок своим первым ходом берет не более k стопок. Игра заканчивается, когда стопок не остается. Требуется найти максимальное число монет, которое может получить первый участник после окончания игры, если второй игрок тоже старается ходить так, чтобы получить как можно больше монет*.

Входные данные. В первой строке находится сначала количество стопок n , за ним идут n чисел, задающих количество монет в стопках слева направо, а затем — число k . *Ограничения:* $1 \leq n \leq 180$, $1 \leq k \leq 80$, количество монет в стопке — не менее 1 и не более 20000, время — 1 секунда.

Выходные данные. Одно число — максимальное количество монет, которое может получить первый игрок.

Пример.

Ввод: 5 3 4 8 1 7 2.

Вывод: 18.

2. Бумажная полоска разделена на n клеток. Двое играющих по очереди выбирают и зачеркивают ровно k пустых смежных клеток. Выигрывает сделавший последний ход. Оба игрока придерживаются правильной стратегии. Дана ситуация игры. Требуется определить, кто выиграет**.

Входные данные. В первой строке содержатся числа n и k , во второй строке n символов: латинская заглавная «O» — пустая клетка, латинская заглавная «X» — зачеркнутая клетка. *Ограничения:* $1 \leq k \leq n \leq 40$.

Выходные данные. Одно число: 1, если выигрывает первый сделавший ход; 2, если выигрывает второй; 0, если ход сделать нельзя.

«Twofive»

Условие задачи

Секретные сообщения между Санта-Клаусом и его маленькими помощниками обычно кодируются так называемым

* Меньшиков Ф. В. Олимпиадные задачи по программированию. СПб.: Питер, 2006. С. 58.

** Там же. С. 64.

25-языком. В этом языке используется **25-алфавит**, который совпадает с латинским алфавитом с одним исключением — в нем отсутствует буква «Z», то есть 25-алфавит содержит 25 латинских букв от «A» до «Y» в том же порядке, что и латинский алфавит. Каждое слово в 25-языке состоит ровно из 25 различных букв. Слово записывается в таблице размера 5×5, строка за строкой. Например, слово ADJPTBEKQUCGLRVFINSWHMOXY будет записано так:

A	D	J	P	T
B	E	K	Q	U
C	G	L	R	V
F	I	N	S	W
H	M	O	X	Y

Правильным словом 25-языка считается слово, буквы которого в каждой строке и каждом столбце расположены по возрастанию их номера в алфавите. Поэтому слово ADJPTBEKQUCGLRVFINSWHMOXY является правильным словом, а слово ADJPTBEGQUCKLRVFINSWHMOXY — нет (возрастающий порядок нарушается во втором и третьем столбцах).

Лексикон Санта-Клауса состоит из всех правильных слов 25-языка. Слова расположены в возрастающем лексикографическом порядке и пронумерованы, начиная с единицы. Например, в лексиконе Санта-Клауса слово ABCDEFGHIJKLMNOPQRSTUVWXYZ имеет номер 1, а слово ABCDEFGHIJKLMNOPQRSUTVWXYZ имеет номер 2. Во втором слове, по сравнению с первым, буквы «U» и «T» поменялись местами.

К сожалению, лексикон Санта-Клауса огромен. Напишите программу, которая определяет порядковый номер по заданному слову из лексикона Санта-Клауса, а по заданному порядковому номеру определяет соответствующее слово. В лексиконе Санта-Клауса не больше 2^{31} слов.

Ввод. Входной файл имеет имя `twofive.in` и состоит из двух строк. Первая строка содержит один из символов — «w» или «n». Если первая строка содержит «w», тогда вторая строка содержит правильное слово 25-языка, то есть строку из 25 символов. Если первая строка содержит «n», то вторая строка содержит порядковый номер существующего слова 25-языка.

Вывод. Выходной файл имеет имя `twofive.out`. Если вторая строка входного файла содержит слово 25-языка, то единственная строка выходного файла должна содержать его порядковый номер в лексиконе Санта-Клауса. Если вторая строка входного файла содержит число, то единственная строка выходного файла должна содержать слово 25-языка с этим номером.

Примеры ввода и вывода.

```
twofive.in:
w
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
twofive.out:
2
```

```
twofive.in:
n
2
```

```
twofive.out:
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Обсуждение задачи

Как начать обсуждение задачи? С примеров, простых примеров!

Случай 5×5 — уже сложный. Уменьшим размер матрицы и заменим для наглядности буквы на цифры.

Язык 1×1 очевиден.

Язык 2×2 содержит только два сообщения: $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ и $\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$.

Давайте попробуем разобраться с языком 3×3 . Нам требуется определить некое *отношение порядка* на множестве сообщений данного типа. Определить — значит для любого сообщения сказать, какое сообщение ему предшествует и какое следует за ним. С этой целью попробуем выписать (чисто интуитивно) все сообщения, — их не так много: 42. Первое сообщение, как и последнее, выписать просто, так же как и понять, что первый и последний символы всех сообщений не изменяются:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 8 \\ 6 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 6 & 7 \\ 5 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 3 \\ 4 & 6 & 8 \\ 5 & 7 & 9 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 1 & 2 & 4 \\ 3 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 4 \\ 3 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 4 \\ 3 & 5 & 8 \\ 6 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 4 \\ 3 & 6 & 7 \\ 5 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 4 \\ 3 & 6 & 8 \\ 5 & 7 & 9 \end{pmatrix} \Rightarrow$$

$$\begin{aligned}
 &\begin{pmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 5 \\ 3 & 4 & 7 \\ 6 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 5 \\ 3 & 4 & 8 \\ 6 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 5 \\ 3 & 6 & 7 \\ 4 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 5 \\ 3 & 6 & 8 \\ 4 & 7 & 9 \end{pmatrix} \Rightarrow \\
 &\begin{pmatrix} 1 & 2 & 6 \\ 3 & 4 & 7 \\ 5 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 6 \\ 3 & 4 & 8 \\ 5 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 6 \\ 3 & 5 & 7 \\ 4 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 6 \\ 3 & 5 & 8 \\ 4 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 2 & 7 \\ 3 & 4 & 8 \\ 5 & 6 & 9 \end{pmatrix} \Rightarrow \\
 &\begin{pmatrix} 1 & 2 & 7 \\ 3 & 5 & 8 \\ 4 & 6 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 4 \\ 2 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 4 \\ 2 & 5 & 7 \\ 6 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 4 \\ 2 & 5 & 8 \\ 6 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 4 \\ 2 & 6 & 7 \\ 5 & 8 & 9 \end{pmatrix} \Rightarrow \\
 &\begin{pmatrix} 1 & 3 & 4 \\ 2 & 6 & 8 \\ 5 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 7 \\ 6 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 8 \\ 6 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 5 \\ 2 & 6 & 7 \\ 4 & 8 & 9 \end{pmatrix} \Rightarrow \\
 &\begin{pmatrix} 1 & 3 & 5 \\ 2 & 6 & 8 \\ 4 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 6 \\ 2 & 4 & 7 \\ 5 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 6 \\ 2 & 4 & 8 \\ 5 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 6 \\ 2 & 5 & 7 \\ 4 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 6 \\ 2 & 5 & 8 \\ 4 & 7 & 9 \end{pmatrix} \Rightarrow \\
 &\begin{pmatrix} 1 & 3 & 7 \\ 2 & 4 & 8 \\ 5 & 6 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 3 & 7 \\ 2 & 5 & 8 \\ 4 & 6 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 4 & 5 \\ 2 & 6 & 7 \\ 3 & 8 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 4 & 5 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 4 & 6 \\ 2 & 5 & 7 \\ 3 & 8 & 9 \end{pmatrix} \Rightarrow \\
 &\begin{pmatrix} 1 & 4 & 6 \\ 2 & 5 & 8 \\ 3 & 7 & 9 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}.
 \end{aligned}$$

Что мы делали? Шли с конца сообщения до первого символа, допускающего изменение. Он найден, то есть определено его место и значение. Находим среди просмотренных символов первый (он по значению должен быть больше найденного), который можно поставить (записать) на это место, а окончание сообщения (то, что просмотрено) «выстраиваем» по принципу минимально возможного элемента из оставшихся, допускающих запись на данное место.

Методическое отступление. Крайне желательно «привязать» сказанные слова к тем изменениям сообщений, которые представлены выше. Понимание отношения порядка можно проверить с помощью задания — «выписать первые десять сообщений языка 4×4 ». (Выписывание всех таких сообщений, конечно, абсурдно, — их 24024.)

Это — неформальное описание логики генерации сообщений заданного языка. Попробуем формализовать его. Разберемся вначале с частной задачей, а именно: пусть требуется, если это возможно, сформировать «хвост» сообщения (с позиции $i + 1$ до n). Сообщение хранится в массиве b , а элементы, которые можно использовать для формирования, представлены значением переменной множественного типа q . Определим константы: $NMax$ (длина сообщения — 4, 9, 16, 25, 36) и $SMax$ (количество элементов в строке — 2, 3, 4, 5, 6). В нашем случае $NMax = 9$, $SMax = 3$. Тогда формальное описание логики решения задачи имеет вид:

```

...
j:=i+1; Error:=False; {Цикл по месту в сообщении.
Если для позиции с номером j мы не можем найти
значение из q, то Error присваивается значение
True — сообщение не может быть сформировано}
While (j<=NMax) And Not Error Do Begin
    k:=1; {Поиск значения k, которое может быть
записано на место j в сообщении. Элемент k должен
быть меньше предыдущего элемента в строке —
k < b[j-1] (если он не первый). Элемент k должен
быть меньше предыдущего элемента в столбце, или
k < b[j-SMax] (мы храним сообщение в одномерном
массиве)}
    While (k<=NMax) And (Not (k In q) Or (k<b[j-SMax]))
Or ((j Mod SMax<>1) And (k<b[j-1])) Do k:=k+1;
    If k<=NMax Then Begin {Элемент найден}
        b[j]:=k;
        q:=q-[k]
    End
    Else Error:=Not Error; {Элемента нет}
    j:=j+1; {Переходим к следующей позиции сообщения}
End;
...

```

Перейдем теперь к основной логике генерации следующего сообщения:

```

Function GetNext(s:String):String;
Var i,j,k:Byte;
Find,Error:Boolean;
a,b:Array [1..NMax] Of Byte;
w,q:Set Of Byte;

```

Begin

<Преобразование строки сообщения *s* в числовой массив *a*>;

w:=[];

i:=NMax;

Find:=False;

While (*i*>0) **And Not Find Do Begin** {"Идем" с конца сообщения и находим первую позицию, значение элемента в которой может быть увеличено на единицу. Если такой позиции нет (*Find* имеет значение *False*), то сообщение является последним}

If (*a*[*i*]+1) **In w Then Begin** {Если следующий элемент уже есть в множестве *w*, то считаем, что текущий элемент (*i*) может быть увеличен на единицу. Однако если после увеличения текущего элемента на единицу мы не можем сформировать "хвост" сообщения, то считаем, что наше решение об увеличении ошибочно}

q:=*w*-[*a*[*i*]+1]+[*a*[*i*]];

b:=*a*;

b[*i*]:=b[*i*]+1;

<Формирование "хвоста" сообщения:
результат — значение переменной *b*; успешность действий — переменная *Error*>;

Find:=Not *Error*

End;

w:=*w*+*a*[*i*];

i:=*i*-1;

End;

If Find Then <Преобразование числового массива *b* в строку *s*>

Else <Строке *s* присвоить значение последнего по порядку сообщения>;

GetNext:=*s*

End;

Методическое отступление. Целесообразно, на наш взгляд, в этот момент работы над задачей сделать «ручную» трассировку приведенной логики для конкретных примеров.

В принципе, используя функцию GetNext, можно решать обе подзадачи исходной задачи: как определять порядковый номер по заданному слову, так и по заданному порядковому номеру определять соответствующее слово. Но этот подход нерационален по временным характеристикам.

Определим пространство состояний задачи, описывающее все возможные, частично заполненные, матрицы $n \times n$. Частично

заполненные — это такие матрицы, у которых для каждого уже записанного элемента все элементы выше и левее его также заполнены (первая строка и первый столбец — особый случай). Другими словами, например, матрицы с чередованием заполненных и незаполненных элементов нас не интересуют.

В каждой частично заполненной матрице есть только одно или два места, допустимых для следующего шага заполнения. Определим массив p (Array[1..SMax] Of Integer). Константа $SMax$ описана ранее. Значение $p[i]$ определяет количество заполненных элементов в строке матрицы с номером i . Оно изменяется от 0 до $SMax$, то есть имеет $Base$ значений ($Base = SMax + 1$). Количество различных значений p равно $Base^{SMax}$. Для матриц 3×3 имеем $4^3 = 64$ значения.

Понятно, что наше пространство состояний задачи является как бы подпространством всех способов заполнения матрицы. Так для рассмотренного примера 3×3 количество интересующих нас матриц равно 42, а не 64. Однако работать удобнее со всеми матрицами, и с этой целью в реализации задачи вводится массив, например с именем d (Array[0.. Base^{SMax} -1] Of Integer), описывающий наше пространство состояний. Элемент массива $d[a]$ равен количеству частично заполненных матриц для состояния a или -1, если в состоянии a мы еще не были.

Состояние a определяется по массиву p следующим образом (фрагмент кода):

```
a:=0;
For i:=1 To SMax Do a:=a*Base+p[i]; {Это ни что
иное, как перевод числа из системы счисления
с основанием Base в десятичную систему счисления}
```

Методическое отступление.

1. К работе с массивами p и d мы вернемся при обсуждении функции подсчета количества частично заполненных матриц (слов языка). В данный момент обсуждения идет их первичная «прокатка».
2. Целесообразно для конкретных значений n устроить небольшой устный тренинг по переводу чисел. Например, пусть для $n = 3$ требуется составить таблицу по типу табл. 13.4.

Таблица 13.4

p	a	p	a	p	a	p	a
100	16	211	37	310	52	322	58
200	32	220	40	311	53	330	60
300	48	221	41	320	56	331	61
210	36	222	42	321	57	332	62

Итак, пусть для нашего примера мы находимся в состоянии $a = 56$, $p = (3, 2, 0)$. Оно соответствует частично и корректно

заполненной матрице $\begin{pmatrix} * & * & * \\ * & * & 0 \\ 0 & 0 & 0 \end{pmatrix}$. Элемент $d[a]$ в этом случае ра-

вен количеству корректных (удовлетворяющих условию задачи) способов полного заполнения матрицы оставшимися буквами. Очевидно, что $d[0]$ для незаполненной матрицы есть общее количество способов заполнения матрицы, или *мощность* нашего языка.

Предположим, что у нас есть функция Calc, которая для любой частично и корректно заполненной матрицы определяет количество полностью сформированных (правильно) матриц из оставшихся букв.

Рассмотрим решение обеих подзадач. Изложение для наглядности «свяжем» со случаем $n = 3$.

Пусть требуется по номеру определить слово языка. Дан номер 12 (значение переменной s). Логика формирования слова приведена в табл. 13.5. Индексы i и j определяют место символа, индекс k — номер символа.

Таблица 13.5

i	j	k	Значение функции Calc	s	Результат (sol)	Комментарий
1	1	1	42	12	A	Количество слов с первой буквой «A» равно 42. Значение 42 больше 12
1	2	2	21	12	AB	
1	3	3	5	12	ABC	Количество слов с буквами «ABC» равно 5. Значение 5 меньше 12
				7	AB	Пропускаем букву «C», а из значения s вычитаем значение 5
1	3	4	5	7	ABD	Количество слов с буквами «ABD» равно 5. Значение 5 меньше 7
				2	AB	Пропускаем букву «D», а из значения s вычитаем значение 5

<i>i</i>	<i>j</i>	<i>k</i>	Значение функции Calc	<i>s</i>	Результат (sol)	Комментарий
1	3	5	5	2	ABE	Количество слов с буквами «ABE» равно 5. Значение 5 больше 2
2	1	3	5	2	ABEC	
2	2	4	5	2	ABECD	
2	3	6	1	2	ABECDF	Количество слов с буквами «ABECDF» равно 1. Значение 1 меньше 2
				1	ABECD	Пропускаем букву «F», а из значения <i>s</i> вычитаем значение 1
2	3	7	1	1	ABECDG	
3	1	6	1	1	ABECDGF	
3	2	8	1	1	ABECDGFH	
3	3	9	1	1	ABECDGFHI	Ответ задачи. Это слово имеет номер 12

Итак, мы размещаем в слове букву за буквой до тех пор, пока количество способов разместить оставшиеся буквы больше или равно текущему номеру слова. Если же окажется, что количество способов меньше номера слова, то очередную букву записывать нельзя, и это значение вычитается из номера слова. Формализованная запись изложенного (без очевидного описания переменных) имеет вид:

Begin

```

ReadLn(s); {Вводим номер слова}
sol:=''; {Начальное значение слова, переменная
имеет тип String}
For i:=1 To SMax Do {Циклы по таблице, которую
не формируем}
  For j:=1 To SMax Do
    For k:=1 To SMax*SMax Do {Цикл по номерам букв}
      If cx[k]=-1 Then Begin {В массиве cx
значением элемента, равным -1, фиксируем факт
неиспользования буквы в слове}
        cx[k]:=i; cy[k]:=j; {Запоминаем в массивах
sx, sy номер строки и номер столбца, в которых
задействована буква с номером k. Они используются
в функции Calc для корректности размещения
следующих букв в слове}

```

```

q:=Calc(1); {Вычисляем количество слов в
языке, имеющих сформированное начало слова
(значение переменной sol). Параметр "единица"
говорит о первом вызове функции Calc}
sol:=sol+Char(k+64); {"Приклеиваем" букву
к началу слова. Функция Char преобразует номер буквы
в соответствии с кодировкой (отсюда константа 64),
в ее символьное представление}
If s<=q Then Break; {Выходим из цикла по k}
Delete(sol,Length(sol),1); {Буква k слову
была приписана ошибочно. Удаляем ее}
s:=s-q; {Изменяем номер слова}
cx[k]:=-1; {Букву с номером k возвращаем
в разряд неиспользованных}
End;
WriteLn(sol);
End;
```

Перейдем ко второй подзадаче. Здесь по слову требуется найти его номер. Пусть при $n = 3$ дано слово «ABGCENDFI». Его номер — 21. Логика его получения представлена в табл. 13.6. Индексы i и j используются для просмотра слова. Для очередной буквы определяется ее числовой эквивалент (номер) в интервале от 1 до $SMax \times SMax$ — значение переменной q . На это место мы условно записываем незанятые буквы с меньшим номером (индекс k) и подсчитываем количество слов с таким началом, а результат прибавляем к ответу — значение переменной sol .

Таблица 13.6

i	j	q	k	sol	Комментарий
1	1	1		1	Обрабатываем первую букву, ее номер (q) — первый
1	2	2		1	Обрабатываем вторую букву, ее номер — второй
1	3	7			Обрабатываем третью букву, ее номер — седьмой. Просматриваем (индекс k) все буквы, начиная с первой. Буквы «А» и «В» заняты
			3	$6 - 1 + 5$	Буква с номером 3 могла бы быть записана на этом месте, и в этом случае есть 5 слов языка с таким началом

<i>i</i>	<i>j</i>	<i>q</i>	<i>k</i>	<i>sol</i>	Комментарий
			4	$11 = 6 + 5$	Буква с номером 4 могла бы быть записана на этом месте, и в этом случае есть 5 слов языка с таким началом
			5	$16 = 11 + 5$	Буква с номером 5 могла бы быть записана на этом месте, и в этом случае есть 5 слов языка с таким началом
			6	$20 = 16 + 4$	Буква с номером 6 могла бы быть записана на этом месте, и в этом случае есть 4 слова языка с таким началом
2	1	3		20	Обрабатываем четвертую букву, ее номер — третий
2	2	5			Обрабатываем пятую букву, ее номер — пятый
			4	$21 = 20 + 1$	Буква с номером 4 могла бы быть записана на этом месте, и в этом случае есть 1 слово языка с таким началом
2	3	8			Обрабатываем шестую букву, ее номер — восьмой
			4	21	Свободная буква с номером 4 не может быть записана на это место
			6	21	Свободная буква с номером 6 не может быть записана на это место
3	1	4		21	Обрабатываем седьмую букву, ее номер — четвертый
3	2	6		21	Обрабатываем восьмую букву, ее номер — шестой
3	3	9		21	Обрабатываем девятую букву, ее номер — девятый

Формализованная запись логики обработки (без описания типов переменных) имеет вид:

Begin

ReadLn(s); {Ввод слова}

sol:=1; {Результат, исходное присвоение}

For i:=1 **To** SMax **Do** {Циклы по буквам слова}

For j:=1 **To** SMax **Do Begin**

q:=**Integer**(s[(i-1)*SMax+j])-64; {функция **Integer**

преобразует символ (букву) в его числовой код.

Константа 64 сдвигает исходный числовой интервал

представления к интервалу от 1 до SMax*SMax}

For k:=1 **To** q-1 **Do** {Цикл по предшествующим буквам}

```

If cx[k]=-1 Then Begin {Если буква
не использована ранее, то она может быть записана
на это место}
    cx[k]:=i; cy[k]:=j; {Запоминаем в массивах
cx, cy номер строки и номер столбца, в которых
задействована буква с номером k. Они используются
в функции Calc для корректности размещения следующих
букв в слове}
    sol:=sol+Calc(1); {Подсчитываем количество
слов с буквой k на месте (i,j)}
    cx[k]:=-1; {Возвращаем букву в разряд
свободных}
End;
    cx[q]:=i; cy[q]:=j; {Фиксируем запись буквы
с номером k на месте (i,j)}
End;
    WriteLn(sol);
End;

```

Методическое отступление. Целесообразно выполнить до обсуждения функции Calc «ручную» трассировку обеих подзадач на конкретных примерах. При этом следует обратить внимание на детальное прописывание заполнения массивов *cx* и *cy* с целью прогнозирования их использования в логике функции Calc.

Итак, используя понятие «пространство состояний задачи» (массивы *d* и *p*), нам осталось обсудить функцию Calc (написание общей логики (основной программы) — чисто техническая задача).

Возьмем конкретный случай, например, четвертую строку табл. 13.6. Требуется подсчитать количество слов языка, начинающихся с букв «ABC» ($p = (3, 0, 0)$, $a = 48$).

Подсчет чисто «ручным» методом дает значение 5. Попробуем прорисовать переход из одного состояния в другое так, как это сделано на рис. 13.5. В эллипсах на рис. 13.5 записываем значение a (номер состояния) и в круглых скобках — соответствующее значение элементов массива *p*. Начинаем с $a = 0$. Единственно возможный переход в $a = 16$, а затем в $a = 32$. Переход в состояние $a = 36$ должен отсекается. Фактически обработка начинается с состояния $a = 48$.

Из каждого текущего частичного заполнения матрицы возможны один или два перехода (приписывание буквы). Один — если состояние является последним в строке (пишем первую букву в следующую строку) или если количество букв в двух последних строках совпадает (приписываем очередную букву в

предпоследнюю строку). Два перехода — если состояние позволяет приписывать букву как в ту строку, которая задается состоянием, так и в первое незанятое место следующей строки.

Следуя этой логике, на рис. 13.5 представлены все переходы из одного состояния в другое. В те состояния, в которые ведут перечеркнутые стрелки, переход не осуществляется. Для них количество возможных слов языка уже вычислено и зафиксировано в массиве d , — зачем считать дважды? Для каждого эллипса на рис. 13.5 также указано количество возможных слов языка для заданного состояния (определенного начала слова). Оно получается путем суммирования соответствующих значений у расположенных ниже эллипсов.

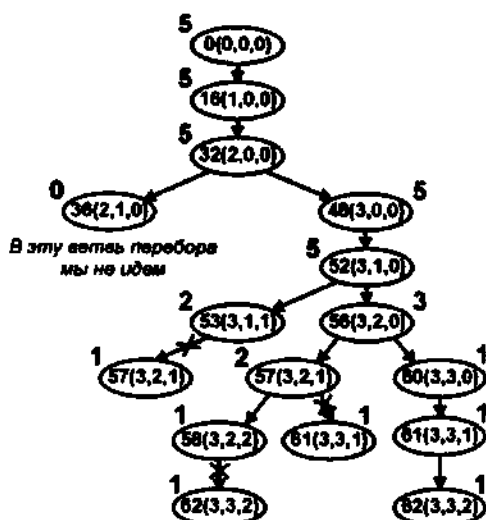


Рис. 13.5. Пример подсчета количества слов

Методическое отступление. Целесообразно сделать аналогичные рисунки для других строк табл. 13.6. Формализованная запись функции Calc после проделанной работы хотя и очевидна, но можно воспользоваться следующим приемом — дать ее частично, а в угловых скобках записать ее в словесном виде. Ее уточнение по ходу обсуждения создает атмосферу диспута.

Function Calc(t :Integer):Integer; {Параметром рекурсии является номер буквы}

Var a, i :Integer;

Begin

If $t=1$ **Then** <Элементам массива d (кроме последнего) присваивается начальное значение -1. Последнему элементу присваивается значение 1>;

$a:=0$;

<Вычисление значения a по элементам массива p >;
{ a — состояние или номер элемента в массиве d , описывающем пространство состояний задачи}

If $d[a]=-1$ **Then Begin** {Данное состояние не обработано}

$d[a]:=0$;

If $cx[t]=-1$ **Then Begin** {Буква не обработана}

```

For i:=1 To SMax Do {Цикл по элементам
массива p. По массиву p осуществляется переход
от одного состояния к другому в пространстве
состояний задачи}
    If (p[i]<SMax) And ((i=1) Or (p[i-1]>p[i]))
Then {Условие перехода в другое состояние}
        Begin
            p[i]:=p[i]+1;
            d[a]:=d[a]+Calc(t+1);
            p[i]:=p[i]-1;
        End
    End
Else {Буква обработана}
    If <Выполняются условия перехода в другое
состояние (определяется на основании данных из
массивов cx и cy)> Then Begin
        p[cx[t]]:=p[cx[t]]+1;
        d[a]:=d[a]+Calc(t+1);
        p[cx[t]]:=p[cx[t]]-1;
    End;
End;
Calc:=d[a];
End;

```

Методические рекомендации

Это задача на комбинаторику (точнее, по комбинаторному объекту требуется вычислять в соответствии с отношением порядка его номер и выполнять обратное преобразование — по номеру вычислять комбинаторный объект). В «идейном» плане она близка к классическим комбинаторным алгоритмам работы с перестановками*. Рассмотрение логики генерации следующего слова языка наиболее наглядно демонстрирует эту близость. Возможен также и другой метод работы с задачей — приведение полного текста программного кода**.

Самостоятельные исследования

1. Модифицируйте процедуру Calc для подсчета общего количества слов языка для различных значений l .

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 31–39.

** Андреева Е. В. Решения задач XIII международной олимпиады. Twofive // «Информатика», еженедельное приложение к газете «Первое сентября». 2001. №40. С. 24–26.

2. Определите значение n , при котором ваш компьютер будет работать более 1 секунды; более 10 секунд; более 1 часа, подсчитывая количество различных слов языка $n \times n$.

ЗАДАЧИ ВТОРОГО ТУРА

«Score»

Условие задачи

Score — это игра для двух игроков. Игровое поле состоит из n точек, пронумерованных от 1 до n , часть из которых соединена стрелками. Каждая стрелка идет от одной точки к некоторой другой точке. Каждая точка принадлежит одному из двух игроков. Игрок, которому принадлежит точка, называется ее владельцем. Кроме того, каждой точке приписано некоторое положительное число. Все приписанные числа различны. Точка с номером 1 является стартовой. В начале игры каждый из игроков имеет 0 очков.

Игра состоит в следующем. Будем обозначать буквой C текущую точку, то есть точку, в которой мы находимся в начале хода. В начале игры C является точкой с номером 1 (стартовой). Ход в игре делает владелец C , при этом выполняются следующие операции:

- если значение, приписанное C , больше, чем текущее количество очков у владельца C , то значение числа, приписанного C , становится новым количеством очков для владельца C . В противном случае число очков у владельца C остается прежним. Количество очков другого игрока в обоих случаях не меняется;
- далее владелец C выбирает по своему усмотрению одну из стрелок, выходящих из текущей точки, и точка, на которую эта стрелка указывает, становится новой текущей. Заметим, что один игрок может сделать подряд несколько ходов.

Игра заканчивается, когда после некоторого хода текущей становится стартовая точка. Победителем считается игрок, который имеет большее количество очков в конце игры.

Набор стрелок между точками всегда такой, что:

- из любой точки выходит по крайней мере одна стрелка;
- любая точка p достижима из стартовой точки, то есть существует такая последовательность стрелок, по которой можно добраться от стартовой точки до p ;
- гарантируется, что игру всегда можно закончить за конечное число ходов.

Напишите программу, которая играет в эту игру и выигрывает. Наборы точек и стрелок во всех играх, в которые будет предложено сыграть вашей программе для ее оценки, организованы так, что ваша программа всегда может выиграть вне зависимости от того, принадлежит ей первый ход или нет. Во время оценки вашей программы противник будет играть оптимально, то есть при первом же неправильном ходе со стороны вашей программы она проигрывает.

Ввод и вывод. Ваша программа читает входные данные из стандартного потока ввода и записывает выходные данные в стандартный поток вывода. Ваша программа — это *Игрок 1*, а противник — *Игрок 2*. Ваша программа начинает свою работу со считывания следующих входных данных из стандартного потока ввода:

- первая строка входных данных содержит одно целое число — количество точек n , $1 \leq n \leq 1000$. Каждая из последующих n строк содержит по n целых чисел (нулей или единиц), описывающих стрелки. Если на игровом поле существует стрелка, направленная от точки i к точке j , то j -е число в i -й строке (из этих n строк) равно 1, в противном случае — 0;
- следующая строка содержит n целых чисел, обозначающих владельцев каждой из n точек. Если точкой i владеет *Игрок 1*, то i -е число — единица, а если *Игрок 2*, то двойка;
- следующая строка содержит n целых чисел, обозначающих значения, приписанные точкам. Если i -е число в этой строке равно j , то i -й точке приписано число j . Все значения, приписанные точкам, различны и находятся в диапазоне от 1 до n ($1 \leq j \leq n$).

После считывания данных начинается игра, стартовой точкой при этом всегда будет точка с номером 1. Ваша программа должна играть следующим образом:

- если ход делает ваша программа (она является владельцем текущей точки), то она должна вывести в стандартный поток вывода номер точки, в которую следует перейти по выбранной вашей программой стрелке;
- если же ход делает ваш противник, то ваша программа должна считать из стандартного потока ввода номер точки, в которую следует перейти по стрелке, выбранной противником;
- когда текущей станет снова точка с номером 1, программа должна заканчивать свою работу.

Рассмотрим следующий пример для игрового поля, показанного на рис. 13.6. Точки, изображенные в виде окружностей, принадлежат *Игроку 1*, а в виде квадратов — *Игроку 2*. Значе-

ния, приписанные точкам, помещены рядом с окружностями или квадратами. Номера точек расположены внутри геометрических фигур.

Следующие входные и выходные данные описывают игру, сыгранную для изображенного на рис. 13.6 игрового поля:

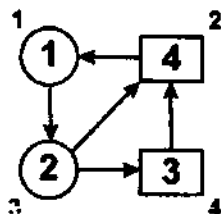


Рис. 13.6. Пример игрового поля

stdin	stdout	Объяснение
4		<i>n</i>
0 1 0 0		Информация о стрелках, выходящих из точки 1
0 0 1 1		Информация о стрелках, выходящих из точки 2
0 0 0 1		Информация о стрелках, выходящих из точки 3
1 0 0 0		Информация о стрелках, выходящих из точки 4
1 1 2 2		Владельцы точек
1 3 4 2		Числа, приписанные точкам
	2	Ход Игрока 1
	4	Ход Игрока 1
1		Игрок 2 ходит в начальную точку — игра заканчивается

В результате данной игры Игрок 1 набирает 3 очка, а Игрок 2 — 2 очка. Игрок 1 выиграл.

Указания по написанию программы. В приведенных ниже примерах переменная *target* обозначает номер точки.

Если программа на C++ использует *iostreams*, вы должны организовать ввод и вывод следующим образом:

```
cin>>target;
cout<<target<<endl<<flush;
```

Если программа на C или C++ использует *scanf* и *printf*, вы должны организовать ввод и вывод следующим образом:

```
scanf("%d",&target);
printf("%d\n",target); fflush(stdout);
```

Если ваша программа на *Pascal*, вы должны организовать ввод и вывод следующим образом:

```
Readln(target);
Writeln(target);
```

Вспомогательные программные средства. Вам дана программа (`score2` для *Linux*, `score2.exe` для *Windows*), которая читает описание начальных данных из файла с именем `score.in` и переписывает их в стандартный поток вывода в форме, в какой их должна считать ваша программа из стандартного потока ввода. После этого программа `score2` играет за *Игрока 2*, читая ходы вашей программы из стандартного потока ввода и записывая свои ходы, сделанные случайным образом, в стандартный поток вывода. Вы можете запустить вашу программу и программу `score2` в отдельных окнах и вручную организовать игру между ними.

Оценка программы. Если ваша программа при тестировании выигрывает, то вы получаете полный балл за тест, в противном случае — 0 баллов. При тестировании ваша программа сначала будет играть с другой программой, и на всю игру будет отводиться на 1 секунду больше времени, чем указано во временных ограничениях. Ввод и вывод программ будут записаны в некоторые файлы. Затем ваша программа запускается второй раз, при этом входные данные направляются ей из записанного при первом запуске файла. Ваша программа должна выдать те же самые выходные данные, что и при первом запуске. При втором запуске определяется реальное время работы вашей программы.

Обсуждение задачи

Поле игры (пространство состояний задачи) описывается связным ориентированным графом. Как следует из формулировки задачи и ограничений в памяти компьютера, граф определяется *матрицей смежности* (обозначим ее как $a[1..NMax, 1..NMax]$, где $NMax$ — константа, задающая максимальное количество точек). Из условия задачи следует, что в графе нет вершин — стоков, т. е. точек, в которые стрелки (дуги — в терминологии теории графов) только заходят, и вершин — источников, т. е. точек, стрелки из которых только выходят. Таким образом, графы подобные представленному на рис. 13.7, недопустимы.

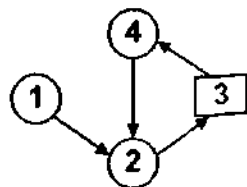


Рис. 13.7. Пример недопустимого графа

Пусть дан граф, показанный на рис. 13.8. Ясно, что второй игрок, находясь в вершине с номером 5 и видя, что переход в вершину 6, приводит к его проигрышу, будет всегда выбирать вершину 3. Это противоречит условию задачи о том, что игра всегда может быть закончена за конечное число ходов. Следовательно, все циклы в графе должны проходить через вершину с номером 1.

Методом решения задачи является традиционный обход в глубину вершин графа (граф — связный), который следует начать с первой вершины. Обход в глубину требует введения массива признаков (*new*), с элементами, равными True или False (не были / были в вершине). Приведем для наглядности логику. Описание графа в памяти компьютера представлено матрицей смежности a^* :

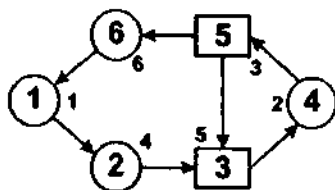


Рис. 13.8. Граф с циклами, не проходящими через вершину с номером 1, также является недопустимым

```

Procedure Pg(t:Integer); {t — номер вершины}
  Var i:Integer;
  Begin
    nnew[t]:=False; {t — вершину посетили}
    For i:=1 To n Do {n — количество вершин}
      If (a[v,i]<>0) And nnew[i] Then Pg(i);
  {Если есть ребро из v в i и в вершине с номером i
  не были, то рекурсивно вызываем Pg}
  End;
  
```

По отношению к этой логике, в решении задачи незначительно усложнятся действия на входе в рекурсию и добавится обработка данных на выходе из рекурсии.

Что нам в принципе надо получить? Для каждой вершины требуется знать, в какую вершину следует делать ход из нее (массив *p*), и для каждой вершины следует оценить максимальные выигрыши игроков (массив *d*). Так, для примера графа на рис. 13.9 они имеют значения: $p = (3, 4, 5, 6, 1, 1)$; $d = ((5,3), (4,6), (5,3), (0,6), (0,3), (0,1))$. Видим, что из вершины 1 следует делать ход в вершину 3; из нее — в вершину 5 и затем — в вершину 1. Первоначально, по логике обхода в глубину, $p[1] = 2$.

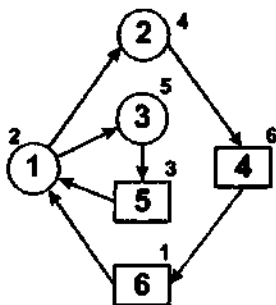


Рис. 13.9. Пример графа

Методическое отступление. «Ручной» просчет значений массивов *d* и *p* для произвольных, не очень сложных, примеров позволит лучше осознать условия задачи и сформировать жела-

емое (или, по П. Я. Гальперину, — ориентировочную составляющую деятельности).

Обсудим теперь самую сложную часть решения — действия на выходе из рекурсии. С каждой вершиной у нас связана следующая информация: принадлежность игроку (массив v , элементы которого равны 1 или 2) и вес вершины (массив w). Пусть номер игрока, делающего ход, является значением переменной Gam , т. е. на входе в рекурсию присутствует оператор $Gam:=v[t]$; Тогда, в общем, при выходе из рекурсии следует сделать:

```

If <выполняются некие условия> Then Begin
{Выполнить переоценку параметров ( $p$  и  $d$ ) для
  вершины с номером  $t$ }
   $ch:=True$ ; {Фиксируем факт оценки параметров
    для вершины  $t$ }
   $d[t, Gam]:=Max(d[i, Gam], w[t])$ ; {Оцениваем выигрыш
    игрока}
   $d[t, 3-Gam]:=d[i, 3-Gam]$ ; {Переписываем выигрыш
    противника. В вершину  $t$  мы возвращаемся из вершины
    с номером  $i$ }
   $p[t]:=i$ ; {Из вершины  $t$  следует делать ход в
    вершину  $i$ }
End;

```

Далее требуется формализовать утверждение «выполняются некие условия».

Первый вариант (рис. 13.10): если выигрыш Gam (хозяина вершины t) меньше выигрыша противника или равен ему, и выигрыш противника в t больше, чем его выигрыш в i , то следует сделать переоценку для Gam .

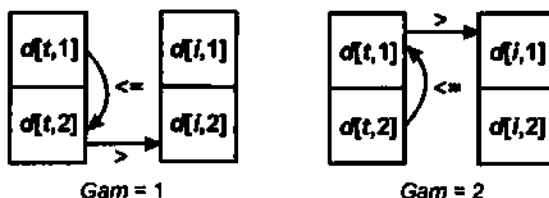


Рис. 13.10. Первый вариант переоценки выигрыша Gam

Второй вариант показан на рис. 13.11. Выигрыш Gam в вершине t больше, чем выигрыш противника, но меньше выигрыша Gam в вершине i , а выигрыш $3-Gam$ в t больше, чем выигрыш в i .

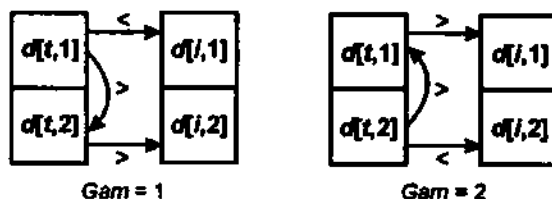


Рис. 13.11. Второй вариант переоценки выигрыша Gam

Методическое отступление.

1. При отсутствии вспомогательных программных средств разумно воспользоваться простой программой игры компьютера с человеком. После ввода данных и инициализации переменных следует вызвать процедуру просчета состояний игры (формирование массивов p и d), а затем процедуру Solve:

Procedure Solve;

Var t:Integer;

Begin

t:=1; {Начинаем с первой вершины}

Repeat

If v[t]=1 **Then Begin** {Если вершина принадлежит первому игроку (в нашем случае — компьютеру), то выводим номер вершины и делаем ход, а иначе ждем ввода номера вершины от второго игрока — человека}

WriteLn(p[t]);

t:=p[t];

End

Else ReadLn(t);

Until t=1; {До тех пор, пока не придем в первую вершину}

End;

2. Логика модификации поиска в глубину может быть предложена для самостоятельной работы учащихся, либо им может быть представлен вариант ее реализации для выполнения «ручной» трассировки на конкретных примерах:

Procedure CalcStates(t:Integer);

Var i,Gam:Integer;

ch:Boolean;

Begin

{Действия на входе в рекурсию}

If nnew[t] **Then Begin**

nnew[t]:=False; {Первое посещение вершины с номером t}

```

Gam:=v[t]; {Чья вершина?}
d[t,Gam]:=w[t]; {Определяем выигрыш игрока}
If a[t,1]=1 Then Begin {Есть ли ребро в первую
вершину?}
    ch:=True;
    p[t]:=1;
End
Else ch:=False;
For i:=2 To n Do
    If a[t,i]=1 Then Begin {Есть ребро?}
        CalcStates(i); {Рекурсивный переход
к вершине с номером i}
        {Действия на выходе из рекурсии}
        If Not ch Or ((d[t,3-Gam]<d[t,Gam]) And
            (d[i,3-Gam]<d[t,3-Gam]) And
            (d[i,Gam]>d[t,Gam])) Or
            ((d[t,3-Gam]>=d[t,Gam]) And
            (d[i,3-Gam]<d[t,3-Gam])) Then Begin
            ch:=True;
            d[t,Gam]:=max(d[i,Gam],w[t]);
            d[t,3-Gam]:=d[i,3-Gam];
            p[t]:=i;
        End;
    End;
End;
End;
End;

```

Методические рекомендации

Эту задачу, вероятно, нельзя отнести к разряду сложных, проблемных. Она может быть использована при изучении алгоритмов на графах. Методическая ценность задачи заключается скорее в возможности внесения элемента соревнования, но в этом случае ее обсуждение следует свести к минимуму. Другой вариант разбора задачи можно найти в работе Е. В. Андреевой*.

Самостоятельные исследования

1. Требуется не просто найти выигрышную стратегию, а стратегию, дающую максимальный выигрыш.
2. Предположим, что циклы разрешены (рис. 13.8), но не разрешается повторно делать ходы в ранее пройденные вершины. Как изменится решение задачи?

* Андреева Е. В. Решения задач XIII международной олимпиады. Score // «Информатика», еженедельное приложение к газете «Первое сентября». 2001. № 42. С. 7–10.

«Double Crypt»

Условие задачи

Улучшенный стандарт шифрования (УСШ) основан на новом алгоритме шифрования. Этот алгоритм работает с тремя блоками информации p , k и c , размер каждого из которых равен 128 битам. По заданному сообщению p и ключу k функция шифрования E возвращает зашифрованный блок c : $c = E(p, k)$.

Обратной функцией к функции шифрования E является функция дешифрования D , то есть справедливо следующее:

$$D(E(p, k), k) = p, \quad E(D(c, k), k) = c.$$

Назовем двойным УСШ алгоритм, в котором функция E применяется последовательно два раза, используя два независимых ключа k_1 и k_2 — сначала k_1 , затем k_2 : $c_2 = E(E(p, k_1), k_2)$.

В этой задаче также задается число s . Во всех ключах только $4*s$ левых битов являются значимыми, остальные $(128 - 4*s)$ правых битов всегда заполнены нулями.

Вам необходимо восстановить пары ключей, которые использовались при шифровании некоторого сообщения двойным УСШ. Вам дается как исходный текст сообщения p , так и соответствующий ему результат двойного шифрования — текст c_2 , а также значение s , определяющее структуру ключей шифрования.

Вам дана библиотека, в которой имеются функции шифрования и дешифрования УСШ.

Вы должны представить для оценки восстановленные ключи, а не восстанавливающую их программу.

Ввод. Вам даны десять входных текстовых файлов с именами от `double1.in` до `double10.in`. Каждый из этих файлов состоит из трех строк. Первая строка содержит целое число s , вторая строка — исходный текст сообщения p , третья строка — зашифрованный текст c_2 , полученный из p алгоритмом двойного шифрования УСШ. Тексты исходного и зашифрованного сообщений записаны в виде строк из 32 шестнадцатеричных цифр (`*0* ... *9*`, `*A* ... *F*`). В данной вам библиотеке есть функция для перевода таких строк в 128-битный блок информации. Для всех входных файлов решение задачи существует.

Вывод. Вы должны послать на проверку десять выходных файлов, соответствующих данным входным файлам. Каждый выходной файл должен состоять из трех строк. Первая строка должна содержать текст `#FILE double I`, где I обозначает номер соответствующего входного файла. Вторая строка должна

содержать ключ k_1 , записанный в виде 32-х шестнадцатеричных цифр («0» ... «9», «A» ... «F»), а третья — записанный в том же формате ключ k_2 , при этом должно выполняться $c_2 = E(E(p, k_1), k_2)$.

В данной вам библиотеке есть функции для перевода 128-ми битных блоков информации в строки из 32-х шестнадцатеричных цифр. Если задача имеет несколько решений, следует отослать для проверки одно из них.

Пример. В качестве примера ниже представлен входной файл с номером 0.

```
double0.in:
1
00112233445566778899AABBCDDDEEFF
6323B4A5BC16C479ED6D94F5B58FF0C2
```

Возможный выходной файл:

```
#FILE double 0
A0000000000000000000000000000000
70000000000000000000000000000000
```

Библиотека.

Библиотека для *FreePascal* (Linux: aeslibp.p, aeslibp.ppu, aeslibp.o; Windows: aeslibp.p, aeslibp.ppw, aeslibp.ow):

```
type
  HexStr = String [32]; {only '0'..'9', 'A'..'F'}
  Block = array [0..15] of Byte; {128 bits}
procedure HexStrToBlock (const hs: HexStr; var b: Block);
procedure BlockToHexStr (const b: Block; var hs: HexStr);
procedure Encrypt (const p, k: Block; var c: Block);
  {c = E(p, k)}
procedure Decrypt (const c, k: Block; var p: Block);
  {p = D(c, k)}
```

Программа aestoolp.pas иллюстрирует использование библиотеки для *FreePascal*.

Библиотека для *GNU C/C++* (Linux and Windows: aeslibc.h, aeslibc.o):

```
typedef char HexStr[33];
/* '0'..'9', 'A'..'F', '\0' — конец строки */
typedef unsigned char Block[16]; /* 128 бит */
```

```

void hexstr2block (const HexStr hs, /* результат */
Block b);
void block2hexstr (const Block b, /* результат */
HexStr hs);
void encrypt (const Block p, const Block k,
/* результат */ Block c);
/* c = E(p,k) */
void decrypt (const Block c, const Block k,
/* результат */ Block p);
/* p = D(c,k) */

```

Программа aestoolc.c иллюстрирует использование библиотеки GNU C/C++.

Ограничения. Для значений s справедливо соотношение $1 \leq s \leq 5$.

Подсказка. Хорошая программа может восстановить ключи не более чем за 10 секунд для любого допустимого входного файла.

Идея решения

Набросок решения очевиден. Определим по заданному значению s максимальное значение (n) ключей k_1 и k_2 . Оно равно $2^{2^s} - 1$. При $s = 5$ имеем: $n = 2^{2^5} - 1 \approx (2^{10})^2 \approx (10^3)^2 = 10^6$.

Логика обработки одного входного файла имеет в общей записи вид:

```

<преобразование s в n>;
For k1:=0 To n Do Begin
  <преобразование k1 в блок>; {Пусть блок будет
иметь то же имя - k1}
  For k2:=0 To n Do Begin
    <преобразование k2 в блок>;
    If E(E(p, k1), k2)=c2 Then <формирование результата,
завершение работы>;
  End;
End;

```

При $s = 5$ время обработки пропорционально 10^{12} , что делает дальнейшие усилия по реализации этого наброска — алгоритма с временной сложностью $O(n^2)$ — бессодержательными.

Предположим, что по зашифрованному блоку c мы умеем быстро, за время $O(1)$, вычислять адрес i в таблице $A[1..NMax]$, в которой хранится как значение c , так и значение ключа k . (Мы пока не говорим о структуре A и значении $NMax$.) Тогда

при этом допущении набросок логики можно изменить следующим образом:

```
<преобразование  $s$  в  $n$ >;
For  $k_1:=0$  To  $n$  Do Begin
  <преобразование  $k_1$  в блок>;
   $c_1:=E(p, k_1)$ ;
  <по значению  $c_1$  формирование адреса  $i$  таблицы  $A$ 
и запись значений  $k_1$  и  $c_1$ >;
End;
For  $k_2:=0$  To  $n$  Do Begin
  <преобразование  $k_2$  в блок>;
   $q:=D(c_2, k_2)$ ;
  <по значению  $q$  формирование адреса  $i$  таблицы  $A$ >;
  If <в таблице  $A$  есть такое значение  $q$ > Then
    <формирование результата, завершение работы>;
End;
```

Итак, если проблема записи и выборки из таблицы A решается за время $O(1)$, то временная сложность данного алгоритма равна $O(n)$, и его дальнейшее совершенствование и уточнение имеет право на существование.

В информатике известен способ решения данной проблемы — *метод хеширования**. Его суть заключается в том, что для некоторых данных (диапазон изменения которых значителен, — в нашем случае это зашифрованные блоки) с помощью заранее определенной функции f вычисляется адрес хранения этих данных в некоторой ограниченной (значительно меньше диапазона) области памяти компьютера (например, в таблице A). Цель функции f (она считается «хорошей») — в равномерном или почти равномерном разбрасывании исходных данных по этой ограниченной области памяти. Так, в нашей задаче в качестве f можно взять первые q битов (*хеш-значение*) зашифрованного сообщения. Мы имеем 2^q различных значений функции f , и, следовательно, значение N_{Max} должно быть не меньше, чем 2^q .

Все было бы замечательно в этом методе, если бы не возникало *коллизий*, т. е. ситуации, в которой различным значениям исходных данных соответствует одно хеш-значение. Способы разрешения коллизий в информатике также известны. Самым простым из них является увеличение размера таблицы и хранение в соседних ячейках данных, находящихся в состоянии кол-

* Окулов С. М. Основы программирования. 2-е изд., испр. М.: БИНОМ. Лаборатория знаний, 2005. С. 289–294.

лизии. Пусть, например, в качестве $NMax$ взято 2^{q+1} значений. Тогда, если первые q битов соответствуют числу i , то адресом элемента A для хранения будет 2^*i , а для данного, состоящего в коллизии — 2^*i+1 . Но в случае, когда данных, находящихся в состоянии коллизии, больше двух (функция f не очень «хорошая»), этот способ не работоспособен. Можно увеличивать значение $NMax$, но в любом случае ненулевая вероятность неработоспособности способа остается.

Другим способом разрешения коллизий является «метод цепочек». Его суть в том, что все данные, имеющие одно и то же хеш-значение, организуются в отдельный список (цепочку) в дополнительной памяти, а таблица A служит для хранения адресов первых элементов этих цепочек. Его описание можно найти в указанной литературе.

Методические рекомендации

«Олимпиадности» в этой задаче не очень много. Она достаточно успешно решается школьниками, если им известны методы хеширования. Их изучение, конечно, целесообразно при подготовке к олимпиадам. Однако использование данной задачи из-за сложности разработки «обрамления» (библиотеки, тестирующей программы, тестов), в практической работе маловероятно: предполагаемый педагогический результат не соответствует затратам труда и времени.

Самостоятельные исследования

В наброске второй логики хранить значение блока s в A не обязательно. Это снижает требование к размеру таблицы A . Модифицируйте логику. Найдите время увеличения работы программы при этом.

«Склад»

Условие задачи

У финской компании, выпускающей высокотехнологичное оборудование, имеется большой прямоугольный склад. На складе работают менеджер и рабочий. Стороны прямоугольника, изображающего склад на плане, в порядке их обхода называют левой, верхней, правой и нижней. Пол склада разделен на единичные квадраты. На плане они образуют ряды и колонки. Ряды пронумерованы целыми числами $1, 2, \dots$, начиная с верхней стороны, а колонки пронумерованы целыми числами $1, 2, \dots$, начиная с левой стороны.

Для хранения технологических устройств на складе есть контейнеры. Контейнеры имеют индивидуальные идентифика-

ционные номера. Каждый контейнер занимает один квадрат. Склад настолько велик, что количество хранящихся контейнеров всегда меньше, чем количество рядов, и меньше, чем количество колонок. Контейнеры со склада не вывозятся, но иногда на склад поступает новый контейнер. Вход на склад находится в левом верхнем углу.

Рабочий расставил контейнеры вблизи левого верхнего угла склада некоторым способом, чтобы можно было легче найти их по идентификационным номерам. Он использует следующий метод.

Пусть необходимо разместить на складе контейнер с идентификационным номером k (будем называть его *контейнер k*). Рабочий проходит по первому ряду слева направо и ищет первый контейнер с идентификационным номером, большим чем k . Если он не находит такого контейнера, то контейнер k устанавливается непосредственно после самого правого контейнера в этом ряду. Если же такой контейнер l найден, то контейнер l заменяется на контейнер k , а контейнер l размещается в следующем ряду с помощью этого же метода. Если рабочий достигает ряда, в котором нет контейнеров, то контейнер устанавливается в самом левом квадрате этого ряда.

Положим, что контейнеры 3, 4, 9, 2, 5, 1 прибыли на склад именно в таком порядке. Тогда размещение контейнеров на складе будет следующим:

```
1 4 5
2 9
3
```

Между менеджером и рабочим происходит следующий диалог:

Менеджер: Контейнер 5 прибыл раньше контейнера 4?

Рабочий: Нет, это невозможно.

Менеджер: О! Так ты можешь восстановить последовательность прибытия контейнеров по их расположению!

Рабочий: Вообще-то, нет. Например, последовательность поступления контейнеров, которые сейчас находятся на складе, могла быть 3, 2, 1, 4, 9, 5, либо 3, 2, 1, 9, 4, 5, либо еще какая-нибудь из оставшихся 14 вариантов.

Менеджер не хочет выглядеть глупым перед рабочим. Помогите ему и напишите программу, которая по заданному расположению контейнеров вычисляет все возможные последовательности поступления контейнеров.

Вход. Входной файл имеет имя `depot.in`. Первая строка этого файла содержит единственное целое число n : количество

рядов, в которых есть контейнеры. Последующие r строк содержат информацию о рядах 1, ..., r . Первым в каждой из этих строк находится число m : количество контейнеров в ряду. Затем в этой строке следуют m целых чисел: идентификационные номера контейнеров в ряду слева направо. Все идентификационные номера контейнеров i удовлетворяют ограничению $1 \leq i \leq 50$.

Обозначим общее количество контейнеров на складе через n , $1 \leq n \leq 13$.

Выход. Выходной файл имеет имя `depot.out`. Выходной файл содержит столько строк, сколько существует возможных различных последовательностей поступления контейнеров. Каждая из этих строк содержит n целых чисел — идентификационные номера контейнеров в возможной последовательности прибытия. Каждая строка описывает последовательность, которая не записана ни в одной другой строке.

Примеры входных и выходных файлов.

Пример 1.

`depot.in:`

```
3
3 1 4 5
2 2 9
1 3
```

`depot.out:`

```
3 2 1 4 9 5
3 2 1 9 4 5
3 4 2 1 9 5
3 2 4 1 9 5
3 2 9 1 4 5
3 9 2 1 4 5
3 4 2 9 1 5
3 4 9 2 1 5
3 2 4 9 1 5
3 2 9 4 1 5
3 9 2 4 1 5
3 4 2 9 5 1
3 4 9 2 5 1
3 2 4 9 5 1
3 2 9 4 5 1
3 9 2 4 5 1
```

Пример 2.

depot.in:

```
2
2 1 2
1 3
```

depot.out:

```
3 1 2
1 3 2
```

Оценка программы. Если выходной файл содержит хотя бы одну неправильную последовательность или не содержит последовательностей вообще, то за тест начисляется 0 баллов. В противном случае балл вычисляется по следующему правилу. Если выходной файл содержит все возможные последовательности, каждую в точности один раз, то начисляется 4 балла. Если в выходном файле содержится не менее половины возможных последовательностей и каждая из них встречается один раз, то начисляется 2 балла. Если в выходном файле содержится меньше половины возможных последовательностей или некоторые из них встречаются более одного раза, то за тест начисляется 1 балл.

Обсуждение задачи

Это — красивая комбинаторная задача. Заданное размещение контейнеров определяет множество допустимых последовательностей их поступления.

В первую очередь следует обратить внимание на крайние случаи. Для размещения, например (1 2 3 4 5 6), всех чисел в одной строке, существует единственная последовательность,

совпадающая с размещением. А для размещения

$$\begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} \quad \text{— также}$$

единственная последовательность — (6 5 4 3 2 1). В этом случае последовательность просто выписывается с конца.

Возьмем «на вооружение» эти закономерности, ибо логика должна работать и для крайних случаев, а затем попытаемся, образно говоря, «нащупать» отношение порядка на множестве последовательностей. *Отношение порядка на множестве ис-*

ходных данных задачи — это фундаментальное положение информатики (принцип). Только в том случае, когда оно установлено, задача подвластна компьютеру*.

Возьмем произвольный пример размещения и, отвечая на вопрос о том, какой элемент в данном размещении мог последним поступить на обработку, последовательно прописываем изменения:

$$\begin{pmatrix} 1 & 4 \\ 2 & \\ 5 & \\ 7 & \end{pmatrix} \Rightarrow \begin{pmatrix} 1 \\ 2 \\ 5 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 \\ 5 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 5 \\ 7 \end{pmatrix} \Rightarrow (7) \Rightarrow (\emptyset) \Rightarrow (7) \Rightarrow \begin{pmatrix} 5 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 \\ 5 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 \\ 2 \\ 5 \\ 7 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 1 & 4 \\ 2 & \\ 5 & \\ 7 & \end{pmatrix} \Rightarrow \begin{pmatrix} 2 & 4 \\ 5 & \\ 7 & \end{pmatrix} \Rightarrow \begin{pmatrix} 2 \\ 5 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 5 \\ 7 \end{pmatrix} \Rightarrow (7) \Rightarrow (\emptyset) \Rightarrow (7) \Rightarrow \begin{pmatrix} 5 \\ 7 \end{pmatrix} \Rightarrow \begin{pmatrix} 2 \\ 5 \\ 7 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 2 & 4 \\ 5 & \\ 7 & \end{pmatrix} \Rightarrow \begin{pmatrix} 2 & 5 \\ 7 & \end{pmatrix} \Rightarrow (2 \ 7) \Rightarrow (2) \Rightarrow (\emptyset) \Rightarrow (2) \Rightarrow (2 \ 7) \Rightarrow \begin{pmatrix} 2 & 5 \\ 7 & \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} 2 & 4 \\ 5 & \\ 7 & \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 4 \\ 2 & \\ 5 & \\ 7 & \end{pmatrix}.$$

Первый символ « $\emptyset$ » соответствует выводу последовательности 7 5 2 1 4, второй — 7 5 2 4 1, третий — 7 2 5 4 1 и четвертый — 2 7 5 4 1. Очевидно, что очередность последовательностей может быть и другой.

Пусть мы храним наше размещение в двумерном массиве $\llbracket 1..l, 0..n \rrbracket$, где n — количество контейнеров. В нулевом столбце $s[i, 0]$ мы фиксируем количество элементов в строке с номером i .

Выводы, которые явно напрашиваются, из рассмотрения примера:

1) выписывание последнего элемента из строки i в результат.

«Срабатывает» в том случае, когда количество элементов в строке с номером i больше, чем в строке с номером $i+1$;

- 2) сдвиг вверх по столбцу j . Простой случай — совпадение количества элементов в строках. В результат записывается верхний элемент. Сложный случай — количество элементов в верхней строке (i) больше, чем в тех, по которым идет сдвиг. Сдвиг идет по простому варианту, пока $s[i+1, j]$ меньше элемента $s[i, j+1]$. Если $s[i+1, j]$ больше $s[i, j+1]$, то в результат записывается элемент $s[i, j+1]$, а на его место — $s[i+1, j]$.

Таким образом, мы неформально описали отношение порядка на множестве последовательностей и, следовательно, логику перехода от одной последовательности к другой.

Методическое отступление. На этом можно закончить обсуждение. Но прежде чем приступить к соревнованию на лучший программный код, желательно вручную еще раз выписать изменения размещений для произвольного примера. При проведении соревнования можно дать готовый код* и попросить его улучшить.

Методические рекомендации

Комбинаторные алгоритмы — необходимая составляющая содержания подготовки школьников к олимпиадам по программированию и, следовательно, к профессиональной деятельности. Данную задачу можно использовать как творческое задание на поиск отношения порядка после проработки основных комбинаторных объектов: множеств, перестановок, сочетаний, размещений и т. д.**

Самостоятельные исследования

Данная задача имеет «грубое» решение — это генерация всех перестановок из n элементов (количество чисел в размещении) и проверка каждой перестановки — приводит ли она к требуемому размещению по правилам, сформулированным в задаче. Требуется экспериментально сравнить временные характеристики алгоритмов, а также установить, при какой размерности (значение n) задачи время решения по каждому алгоритму укладывается в 1 секунду.

* Андреева Е. В. Решения задач XIII международной олимпиады. Склад // «Информатика», еженедельное приложение к газете «Первое сентября». 2001. № 43. С. 22–23.

** Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 25–78.

ЗАДАЧИ ПЕРВОГО ТУРА

«Вредная лягушка»

Условие задачи

Маленькие вредные лягушки «ченгайгури» стали в Корее легендарными. По ночам они прыгают по рисовым полям, втаптывая в землю кустики риса, нанося урон посевам. Утром, заметив поврежденные кустики, вам захотелось определить путь той лягушки, которая нанесла наибольший урон. Лягушка всегда прыгает по полю по прямой линии, причем длина каждого прыжка одинакова. А разные лягушки могут иметь разную длину прыжка и разные направления (рис. 14.1).

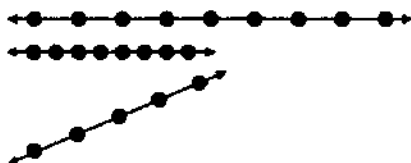


Рис. 14.1. Разные лягушки могут иметь разную длину прыжка и разные направления

Кустики риса на вашем поле посажены строго вдоль перпендикулярных линий на одинаковом расстоянии друг от друга, как это показано на рис. 14.2. Вредные лягушки проскакали через все ваше поле, причем путь лягушки начинался за пределами поля с одной его стороны, а заканчивался за его пределами с другой стороны, как это показано на рис. 14.3.

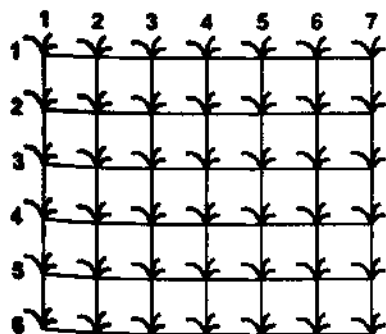


Рис. 14.2. Расположение кустиков риса на поле

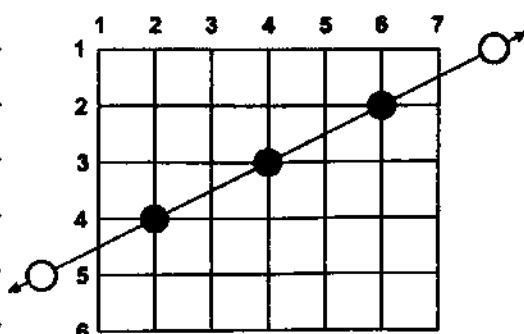


Рис. 14.3. Маршрут лягушки

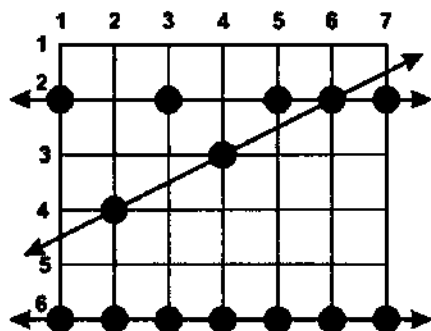


Рис. 14.4. Прямые линии, по которым лягушки затаптывали кустики

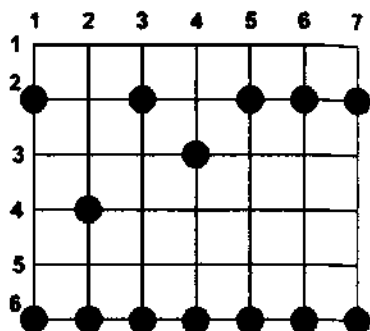


Рис. 14.5. Затапанные кустики риса без прямых линий

По полю может прыгать много лягушек, перепрыгивая от одного рисового кустика к другому. При каждом прыжке лягушка втаптывает в землю очередной кустик риса, как на рис. 14.4. Отметим, что за ночь одно и то же растение может быть затапано более чем одной лягушкой. Конечно, вы не сможете увидеть прямые линии, вдоль которых прыгали лягушки, и какие-либо следы их перемещения за пределами поля. Для примера, показанного на рис. 14.4, вы увидите картину поврежденного поля, показанную на рис. 14.5.

Исходя из рис. 14.5, вы сможете восстановить все возможные пути, по которым лягушки могли бы перемещаться через ваше поле. Нас интересуют только те лягушки, которые повредили не менее трех рисовых кустиков при своем перемещении через поле. Таким образом, мы определили понятие «путь лягушки».

Можно заметить, что три пути, показанные на рис. 14.4, являются путями лягушек (имеются также другие возможные пути лягушек). Вертикальный путь, проходящий вниз вдоль столбца 1, мог бы быть путем лягушки, имеющей длину прыжка 4, однако при этом могло быть повреждено только 2 кустика риса, и такой путь нас не интересует. Рассмотрим диагональный путь, проходящий через кустики, находящиеся на пересечении строки 2 и столбца 3, строки 3 и столбца 4, строки 6 и столбца 7. На этом пути находятся три поврежденных кустика, но путь неравномерный и содержит интервалы неравной длины; таким образом, этот путь также не является путем лягушки. Отметим также, что вдоль линии движения лягушки могут встречаться поврежденные кустики, которые не были повреждены именно этой лягушкой (например, кустик в точке (2, 6) на горизонтальном пути вдоль строки 2 на рис. 14.4), и на поле могут находиться поврежденные кустики, происхождение повреждения которых нельзя объяснить никаким путем лягушки.

Рассмотрим все возможные пути лягушек. Ваша программа должна определить количество поврежденных кустиков риса на пути той лягушки, которая повредила больше всех кустиков (то есть нанесла самый большой урон рисовой плантации). На рис. 14.5 таким мог быть путь вдоль строки 6, а ответом является число 7.

Входные данные. Ваша программа должна читать данные со стандартного ввода. Первая строка содержит два целых числа r и c , задающих соответственно количество строк и столбцов на вашем рисовом поле, $1 \leq r, c \leq 5000$. Вторая строка содержит одно целое число n — количество поврежденных рисовых кустиков, $3 \leq n \leq 5000$. Каждая из следующих n строк содержит два целых числа, номер строки ($1 \leq \text{номер строки} \leq r$) и номер столбца ($1 \leq \text{номер столбца} \leq c$), определяющие позицию поврежденного куста. Эти числа разделяются одним пробелом. Каждый поврежденный кустик описан только один раз.

Выходные данные. Ваша программа должна выводить ответ в стандартный вывод. Вывод программы должен содержать одну строку с одним числом, равным количеству кустиков, лежащих на пути лягушки, которая нанесла наибольший урон, если такой путь существует. В противном случае необходимо вывести 0.

Примеры входных и выходных данных.

Пример 1 (рис. 14.5).

Ввод:

```
6 7
14
2 1
6 6
4 2
2 5
2 6
2 7
3 4
6 1
6 2
2 3
6 3
6 4
6 5
6 7
```

Вывод:

```
7
```

Пример 2 (рис. 14.6).

Ввод:

```
6 7
18
1 1
6 2
3 5
1 5
4 7
1 2
1 4
1 6
1 7
2 1
2 3
2 6
4 2
4 4
4 5
5 4
5 5
6 6
```

Вывод:

```
4
```

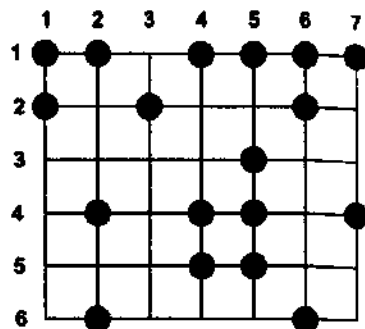


Рис. 14.6. Пример затоптанных кустиков

Идея решения

Задача имеет длинную формулировку и настолько же короткое и простое решение.

Первый и очевидный факт заключается в том, что любой путь лягушки однозначно определяется двумя соседними затоптанными кустиками (точками). Поэтому можно перебирать все пары заданных точек, считая их соседними, по этой паре строить прямую, определять, может ли она быть путем лягушки, и если да, то подсчитывать количество затоптанных кустиков. Проверка «затоптанности» осуществляется с помощью создаваемой при вводе данных матрицы $A[1..r, 1..c]$ с логическим типом элементов. Если кустик (i, j) поврежден, то $A[i, j] = \text{True}$. Временная сложность алгоритма $O(n^3)$: перебор всех пар точек — порядок n^2 и проверка точек на принадлежность прямой — порядок $O(n)$.

Необходимость улучшения решения следует из ограничений: время прохождения теста не превышает 2 секунды, а количество действий (операций) при реализации алгоритма с временной сложностью $O(n^3)$ имеет порядок 10^{11} . Линейного

алгоритма нам не найти: задача описывается двумерной структурой данных, просмотр которой обязателен; значит, единственное, что мы можем сделать — приблизить временную сложность алгоритма к $O(n^2)$.

Однако эта задача такова, что и улучшение алгоритма достаточно «прозрачно». Зачем по каждой паре точек строить путь? Достаточно его строить только для пар точек, являющихся *крайними*, т. е. если их продолжение влево, вправо, вверх или вниз дает выход за пределы поля. Так как каждый путь лягушки имеет крайние точки, то, перебирая только эти крайние точки, мы так или иначе просмотрим пути всех лягушек. Временная сложность алгоритма — $O(k*n^2)$, где k — количество возможных путей лягушек.

Теперь нам ничего не остается делать, как привести текст решения на псевдокоде, ибо получается так, что обсуждение задачи значительно компактнее ее формулировки:

```
Procedure Solve;
  Var i,j,k,tx,ty:Integer;
  Begin
    res:=0; {Ответ задачи}
    For i:=1 To n Do
      For j:=1 To n Do
        If ((x[j]+(x[j]-x[i]))>R) Or ((y[j]+(y[j]-y[i]))>C)
          Or ((x[j]+(x[j]-x[i]))<1) Or
          ((y[j]+(y[j]-y[i]))<1) Then Begin
          tx:=x[j]; {Точки — крайние}
          ty:=y[j];
          k:=0;
          While (tx<=R) And (ty<=C) And (tx>0) And
            (ty>0) And (A[tx,ty]) Do Begin
            tx:=tx-(x[j]-x[i]);
            ty:=ty-(y[j]-y[i]);
            k:=k+1;
          End;
          If (tx>R) Or (ty>C) Or (tx<1) Or (ty<1) Then
            If res<k Then res:=k
        End
      End
    End;
```

Методические рекомендации

Эту задачу можно отнести к учебным при изучении абстрактной структуры данных «матрица» (двумерный массив). Интерес в данном случае заключается в элементах формализации, присущих задаче. При работе с олимпиадной проблематикой

ее использование требует наличия тестирующей оболочки и генератора тестов максимальной размерности. Решение же этой задачи через поиск компонент связности в соответствующем графе или с помощью динамических схем так, как описывается в публикации М. Бабенко и др.^{*}, вряд ли в данном случае целесообразно.

Самостоятельные исследования

Выясните, при каких значениях размерности задачи разница по времени между решениями со сложностью $O(n^3)$ и $O(k \cdot n^2)$ становится ощутимой.

«Утопия»

Условие задачи

Однажды прекрасная страна Утопия была разорена войной. Когда враждебность утихла, страна была разделена на четыре области по долготе (линия север — юг) и широте (линия восток — запад) так, как схематично показано на рис. 14.7. Точку пересечения этих линий обозначили $(0, 0)$. Все четыре части претендовали на название Утопия, но со временем их стали называть Утопия 1 (северо — восточная), 2 (северо — западная), 3 (юго — западная) и 4 (юго — восточная). Каждая точка в любом регионе определялась расстоянием на восток и расстоянием на север от $(0, 0)$. Эти расстояния могли быть отрицательными, то есть точка в Утопии 2 обозначалась парой (отрицательное, положительное), в Утопии 3 — парой (отрицательное, отрицательное), в Утопии 4 — парой (положительное, отрицательное), в Утопии 1 — парой (положительное, положительное).

Наибольшей проблемой являлось то, что гражданам запрещалось пересекать границы. К счастью, изобретательный участник IOI из Утопии разработал простой способ телепортации. Телепортер требует кодовые числа, каждое из которых может быть использовано

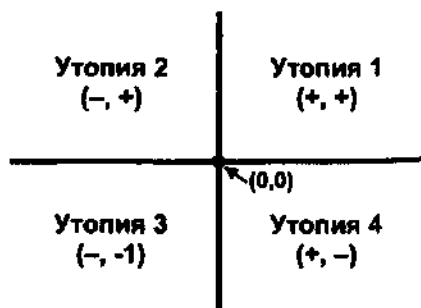


Рис. 14.7. Расположение областей Утопии

^{*} Бабенко М., Матюхин В., Митричев П. Решение задач международной олимпиады школьников по информатике 2002 года. Вредная лягушка // «Информатика», Ежедневное приложение к газете «Первое сентября». 2003. № 1. С. 15–18.

только один раз. Вам необходимо провести телепортер с его начальной позиции $(0, 0)$ через области Утопии в требуемом порядке. У вас будет последовательность из n номеров областей, в каждой из которых вы должны приземлиться, при этом не имеет значения, где именно в области вы приземлились. Вас могут попросить приземлиться в одной и той же области несколько раз подряд. После того, как вы покинете точку $(0, 0)$, вам нельзя попадать на границы областей.

Исходные $2n$ кодовых числа вы должны записать как последовательность n кодовых пар, добавляя знак «плюс» или «минус» перед каждым числом. Если вы находитесь в точке (x, y) и используете кодовую пару $(+u, -v)$, вы будете телепортированы в точку $(x+u, y-v)$. У вас есть $2n$ чисел, и вы можете использовать их в любом выбранном вами порядке, каждое со знаком «плюс» или «минус».

Допустим, у вас есть кодовые числа 7, 5, 6, 1, 3, 2, 4, 8, и вы должны провести телепортер по последовательности областей с номерами 4, 1, 2, 1. Последовательность кодовых пар $(+7, -1)$, $(-5, +2)$, $(-4, +3)$, $(+8, +6)$ позволяет это сделать, телепортируя вас из точки $(0, 0)$ последовательно в точки $(7, -1)$, $(2, 1)$, $(-2, 4)$ и $(6, 10)$. Эти точки расположены в Утопии 4, Утопии 1, Утопии 2 и Утопии 1 соответственно.

Задание. Вам дано $2n$ различных кодовых чисел и последовательность n номеров областей, указывающих, где должен приземлиться телепортер. Постройте последовательность кодовых пар из заданных кодовых чисел, которая проведет телепортер через заданную последовательность областей.

Входные данные. Ваша программа должна читать стандартный ввод. Первая строка содержит положительное целое n ($1 \leq n \leq 10000$). Вторая строка содержит $2n$ различных целых кодовых чисел ($1 \leq \text{кодовое число} \leq 100000$), разделенных одиночными пробелами. Последняя строка содержит последовательность n номеров областей, каждый из которых есть 1, 2, 3 или 4.

Выходные данные. Ваша программа должна писать в стандартный вывод. Вывод должен состоять из n строк. Каждая строка должна содержать пару кодовых чисел, перед каждым из которых указан знак («плюс» или «минус»). Эти кодовые пары будут направлять телепортер по заданной последовательности областей. Заметьте, что не должно быть пробелов между знаком и числом, но должен быть один пробел после первого кодового числа.

Если существует несколько решений, ваша программа должна вывести любое из них. Если решений нет, ваша программа должна вывести единственное целое 0.

Примеры ввода и вывода.**Пример 1.****Ввод:**

4

7 5 6 1 3 2 4 8

4 1 2 1

Вывод:

+7 -1

-5 +2

-4 +3

+8 +6

Пример 2.**Ввод:**

4

2 5 4 1 7 8 6 3

4 2 2 1

Вывод:

+3 -2

-4 +5

-6 +1

+8 +7

Система оценки. Если ваша программа выводит правильный ответ в отведенное для теста время, вы получите все баллы за этот тест; иначе вы получите за этот тест 0 баллов.

Обсуждение задачи

Первичное упрощение задачи — путь к ее реальному осознанию и, как следствие, нахождение эффективных приемов решения. Тем более, что в данном случае сама постановка задачи подталкивает на этот шаг.

Итак, пусть нам дано n чисел. Две области (числовая прямая) определяются знаками «+» и «-». Последовательность знаков описывает *знаки частичных сумм*. Требуется переставить числа и приписать им знаки, так чтобы частичные суммы $(S_j = \sum_{i=1}^j a_i)$ имели заданные знаки, т. е. были положительными или отрицательными.

Второй эффективный прием осознания задачи — «идти» через пример, через рисунок (или образ). Пусть дана произвольная последовательность из восьми чисел: 7, 5, 9, 1, 3, 11, 4, 8 и чередующаяся последовательность областей: «+», «-», «+», «-», «+», «-», «+», «-». Схематично то, что нам требуется сделать.

представлено на рис. 14.8. Сортировка чисел «напрашивается», — нам требуется прибавлять на каждом шаге к очередному числу такое, чтобы осуществлялась «переброска» в другую область. Следующее число в отсортированной последовательности больше предыдущего — «переброска» выполняется.

Отсортируем за время $O(n \cdot \log_2 n)$ числа: 1, 3, 4, 5, 7, 8, 9, 11, и припишем им знаки: +1, -3, +4, -5, +7, -8, +9, -11.

Задача решается, но не будем торопиться. Если заданная последовательность областей имеет вид: $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, то и чередование знаков в числовой последовательности должно быть другим. Но в любом случае знак последнего числа последовательности соответствует последней заданной области.

Изменим в очередной раз последовательность областей: $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$. Если числам последовательности приписать эти знаки, то получим: +1, +3, +4, -5, -7, -8, +9, +11. Действуя по старой логике, мы не получаем решения (на четвертом шаге нас не «перебрасывает» в отрицательную область). Что-то здесь не так, — но есть выводы данного шага рассуждений — требуется сортировка, ибо для частного случая логика работает, и знак последнего числа в отсортированной последовательности соответствует последней области (остальные вычисляются из принципа чередования), а вот дальнейшее следует уточнять.

В первой последовательности областей чередование ($\leftarrow + \rightarrow$ на $\leftarrow - \rightarrow$, $\leftarrow - \rightarrow$ на $\leftarrow + \rightarrow$) происходит семь раз, во второй — два раза. В первом случае мы получили решение, идя вправо по последовательности, и при каждом прибавлении знак частичной суммы чередовался. А если брать не первое число, а в середине, такое, чтобы справа от него оставалось количество чисел, равное требуемому количеству чередования областей? Итак, если не чередуется, то берем слева от выбранного числа, а если есть чередование, то справа. Время линейно — $O(n)$, и завершение логики очевидно. Попробуем: отсортированная последовательность: 1, 3, 4, 5, 7, 8, 9, 11; знаки: $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$ (последнее число должно быть положительным, а далее идет чередование). В заданной последовательности областей $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow - \rightarrow$, $\leftarrow + \rightarrow$, $\leftarrow + \rightarrow$ два раза есть чередование ($k = 2$). Выбираем $n - k$ элемент +8 (справа должно быть два элемента), а затем -7 и +5 (области не чередуется). Получили:

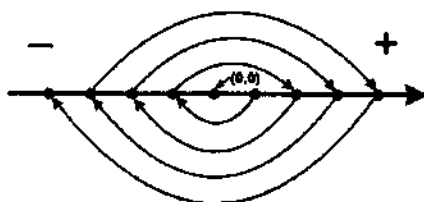


Рис. 14.8. Схематичное представление решения задачи

+8, -7, +5. Идет смена области — выбираем -9, затем -4 и 3, новая смена области — выбираем 11 и заканчиваем -1. Сформировали последовательность: +8, -7, +5, -9, -4, +3, 11, -1. Она удовлетворяет требованиям задачи.

Методическое отступление. В данный момент работы над задачей следует устроить небольшую игру — формирование результата для произвольных последовательности чисел и чередования областей.

Приведем реализацию алгоритма на псевдокоде; его сложность — $O(n)$. Таким образом, самой трудоемкой по времени операцией является сортировка элементов массива (рекомендуется использовать пирамидальную логику):

```

Procedure Solve(Var x,dx:TArr); {x — отсортированный
массив чисел; dx — последовательность областей;
тип данных: TArr=Array [1..NMax] Of Integer}
  Var a:TArr;
      i,k,l,r:Integer;
  Begin
    x[n]:=x[n]*dx[n]; {Знак последнего числа}
    For i:=n-1 DownTo 1 Do
      If x[i+1]>0 Then x[i]:=-x[i]; {Формируем
чередование знаков у последовательности чисел}
      k:=0;
      For i:=2 To n Do
        If dx[i]<>dx[i-1] Then k:=k+1; {Количество
чередований областей}
        a[1]:=x[n-k]; {Формируем первый элемент
результатирующей последовательности чисел}
        l:=n-k-1;
        r:=n-k+1;
        For i:=2 To n Do
          If dx[i]=dx[i-1] Then Begin
            a[i]:=x[l]; {Область сохраняется — идем влево}
            l:=l-1;
          End
          Else Begin
            a[i]:=x[r];
            r:=r+1;
          End; {Область изменяется — идем вправо}
        x:=a
  End;

```

Обобщение задачи на двумерный случай достаточно очевидно. Разбиваем исходную последовательность на две. Формиру-

ем для каждой последовательности очередность областей (все это можно сделать при вводе данных). Решаем задачу два раза, а при выводе формируем результат из двух полученных последовательностей.

Методические рекомендации

Это — задача на идею, оригинальную идею. С методической точки зрения она очень интересна — с точки зрения как отработки технологии решения задач, так и процесса поиска метода решения. Сформировать «рождение» идеи в процессе обсуждения задачи на простых примерах, не говоря о самой идее («делаем так-то и так-то»), а создавая условия для ее «рождения» у школьника — достаточно нетривиальная педагогическая проблема.

Самостоятельные исследования

1. Известно, что *метод сортировки Ч. Хоара* (быстрая сортировка) при некоторых исходных данных работает за $O(n^2)^*$. Разработайте генератор последовательностей. Проверьте работу алгоритма для этих исходных данных и выясните, достаточно ли в этом случае двух секунд (ограничение задачи на прохождение теста) для решения задачи.
2. После того, как задача решена с точки зрения практической информатики (а именно такой занимаются на олимпиадах), целесообразно выполнить и ее теоретическое обоснование**. Дайте определение чередующейся последовательности $X = (x_1, x_2, \dots, x_n)$, отсортированной по модулю чисел, покажите, что знак x_n совпадает со знаком $\sum_{i=1}^n x_i$ и докажите, что требуемая последовательность получается из чередующейся путем перестановки элементов последовательности X .
3. Можно предложить более сложный вариант задачи «на идею», которая созвучна рассмотренной задаче по формулировке. Даны натуральные числа p и q . Требуется разработать программу определения последовательности натуральных чисел наибольшей длины n , такой, что значение суммы любых p идущих подряд элементов этой последовательности положительно, а любых q — отрицательно***.

* Окулов С. М. Основы программирования. М.: БИНОМ. Лаборатория знаний. 2002. С. 260–262.

** Бабенко М., Матюхин В., Митричев П. Решение задач международной олимпиады школьников по информатике 2002 года. Утопия // «Информатика», Ежегодное приложение к газете «Первое сентября». 2003. № 5. С. 25–27.

*** Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний. 2002. С. 315–317.

«XOR»

Условие задачи

Вы разрабатываете программу для мобильного телефона, оснащенного черно-белым экраном, на котором x -координаты отсчитываются слева направо, а y -координаты — сверху вниз, как показано на рис. 14.9. Ваша программа использует разнообразные картинки, которые могут быть различного размера. Вместо того, чтобы хранить эти картинки, вы хотите создавать их, используя графическую библиотеку телефона. В начале рисования картинки все пиксели экрана окрашены в белый цвет. Единственная графическая операция, поддерживаемая библиотекой, — это $XOR(L, R, T, B)$, которая перекрашивает в противоположный цвет пиксели, попадающие в прямоугольник с левым верхним углом в точке (L, T) , а правым нижним — в точке (R, B) , где L обозначает левую, T — верхнюю, R — правую, B — нижнюю координаты. Заметим, что в некоторых других графических библиотеках порядок аргументов может отличаться.

Например, рассмотрим получение картинки на рис. 14.9в. Применяя $XOR(2, 4, 2, 6)$ к полностью белой картинке, получаем картинку, показанную на рис. 14.9а. Применяя $XOR(3, 6, 4, 7)$ к картинке с рис. 14.9а, получаем картинку, показанную на рис. 14.9б; наконец, применяя $XOR(1, 3, 3, 5)$ к картинке с рис. 14.9б, получаем картинку, показанную на рис. 14.9в.

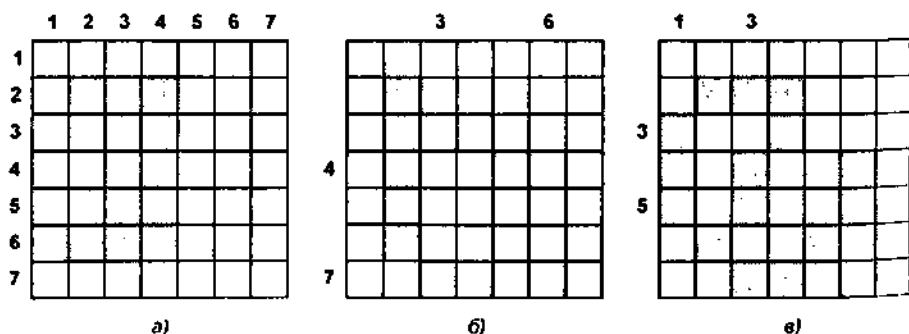


Рис. 14.9. Пример получения картинки

Ваша задача: для заданной черно-белой картинки построить по возможности кратчайшую последовательность вызовов XOR , приводящую к генерации, начиная с белого экрана, этой картинки. Вам даны входные файлы, описывающие картинки. Вы должны сдать файлы, описывающие параметры требуемых вызовов XOR , а не исходный текст программы для создания этих файлов.

Входные данные. Вам даны 10 текстовых файлов с именами с `xor1.in` по `xor10.in`. Каждый входной файл устроен следующим образом. Первая его строка содержит одно целое число n , $5 \leq n \leq 2000$, обозначающее, что картинка имеет n строк и n столбцов. Оставшиеся строки входного файла описывают строки картинки сверху вниз. Каждая строка содержит n целых чисел: цвета пикселей слева направо. Значение 0 обозначает белый пиксель, а значение 1 — черный.

Выходные данные. Вы должны сдать 10 выходных файлов, соответствующих входным файлам. Первая строка файла должна содержать текст `#FILE xor I`, где целое число I — номер соответствующего входного файла. Вторая строка должна содержать целое число K — количество вызовов XOR. В последующих K строках должны располагаться описания этих вызовов в порядке от первого к последнему. Каждая из этих K строк должна содержать четыре целых числа: параметры вызова XOR в порядке L, R, T, B .

Пример ввода и вывода.

```
xor0.in:
7
0 0 0 0 0 0 0
0 1 1 1 0 0 0
1 0 0 1 0 0 0
1 0 1 0 1 1 0
1 0 1 0 1 1 0
0 1 0 0 1 1 0
0 0 1 1 1 1 0
```

```
xor0.out:
#FILE xor 0
3
2 4 2 6
3 6 4 7
1 3 3 5
```

Система оценки. Если последовательность вызовов XOR, указанная в вашем файле, не приводит к генерации требуемой картинки, или число вызовов XOR, указанных в вашем файле, не равно K , или в вашем выходном файле $K > 40000$, или ваш выходной файл содержит вызов XOR с параметрами $L > R$ или $T > B$, или ваш выходной файл содержит вызов XOR с неположительными координатами, или ваш выходной файл содержит вызов XOR со значением координаты, превышающим n , то вы получите за данный тест 0 баллов. В противном случае ваш

балл вычисляется по формуле: $1 + 9 \times \frac{\text{ЧислоВызововВЛучшемРешенииУчастников}}{\text{ЧислоВызововВВашемРешении}}$.

Баллы округляются до первого знака после десятичной запятой по каждому тесту. Общий балл округляется до ближайшего целого. Предположим, что вы сдали решение с 121 вызовом XOR. Если это решение — лучшее среди всех участников, то вы получите 10 баллов. Если лучшее среди сданных участниками решений использует 98 вызовов XOR, то ваш балл составит $1 + 9 \times 98 / 121 (=8.289...)$ и будет округлен до 8.3.

Обсуждение задачи

Мы не будем останавливаться на различных вариантах алгоритмов: перекрашивание по одному квадрату, перекрашивание по прямоугольнику, левый верхний край которого — первая закрашенная клетка (при просмотре поля слева направо и сверху вниз), при этом его размеры могут определяться различными способами. Анализ таких решений можно вынести (при наличии соответствующего обеспечения — тестов, проверяющей программы) для самостоятельного исследования.

Заметим, что решение задачи упрощается, если идти не от белого экрана к заданной картинке, а наоборот, от картинки к белому экрану. Эквивалентность задач очевидна — найденная последовательность команд XOR, выполненная в обратном порядке, приведет нас от белого экрана к заданной картинке.

Методическое отступление. Рекомендуется в этот момент обсуждения задачи выполнить «ручной» поиск последовательности команд XOR для произвольных небольших картинок. Следует убедиться в справедливости утверждения и, более того, в том, что при любом порядке вызовов XOR, если команды найдены, мы получаем один и тот же результат. Кроме того, поиск последовательности команд следует выполнить с помощью простых алгоритмов и сравнить результаты. После такой работы дальнейшее обсуждение «ложится» на подготовленную почву.

Анализ различных картинок позволяет выделить нечто (признак, свойство), характеризующее их. А именно: углы условной сетки, на которой рисуется картинка, могут быть только четырех типов (с точностью до симметрии и поворотов). Они показаны на рис. 14.10, «жирными» отрезками выделены закрашенные области. (Здесь пропущен

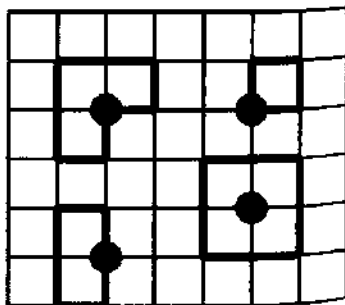


Рис. 14.10. Типы углов сетки

пятый тип, не имеющий соседних окрашенных квадратиков, но он представляет интерес только в том, что окончательная картинка (белое поле) имеет все углы этого типа.) Углы первых двух типов на рис. 14.10 имеют три или один соседний закрашенный квадратик (нечетное число); углы вторых двух типов — два (соседних по вертикали или диагонали) или четыре квадратика (четное число). Тип угла сетки на рис. 14.11 мы отнесем ко второму типу углов.

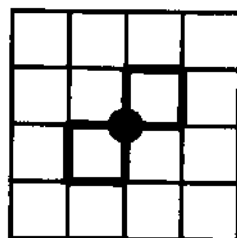


Рис. 14.11. Тип угла

Рассмотрение этих картинок наталкивает нас на утверждение о том, что любую из них можно свести к белому экрану путем применения операции XOR к прямоугольникам, имеющим в качестве всех четырех вершин углы первых двух типов. Если это так, то количество операций XOR равно количеству таких прямоугольников. Из этого предположения следует алгоритмическое решение, заключающееся в том, что следует искать прямоугольники с максимальным количеством вершин (четыре или три) первых двух типов и применять операцию XOR для них. Осталось только определиться со стратегией поиска, простой и достаточно эффективной.

Модифицируем стратегию, предложенную в публикации М. Бабенко*. Находим первую вершину (самую левую и верхнюю). Это вершина x (рис. 14.12). Продолжаем «луч» по оси Y до первой точки другого цвета — y (рис. 14.12), а затем до первой точки другого цвета по оси X — z (рис. 14.12). Нашли три вершины прямоугольника — выполняем операцию XOR и продолжаем процесс. Картинка будет сведена к белому экрану за четыре операции XOR. На самом деле минимальное количество — три. Незначительно изменим стратегию — на лучах по осям X и Y мы находим все точки нашего типа, а затем смотрим —

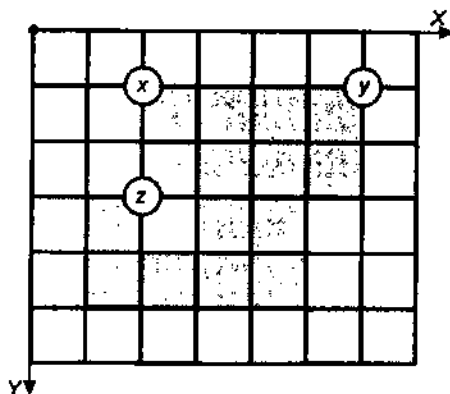


Рис. 14.12. Стратегия поиска четырехугольника

* Бабенко М., Матюхин В., Митричев И. Решение задач международной олимпиады школьников по информатике 2002 года. XOR. // «Информатика», еженедельное приложение к газете «Первое сентября», 2003. № 4. С. 18–20.

какая пара точек (нам достаточно одной) дает четвертый угол прямоугольника требуемого типа. С этим прямоугольником и выполняется операция XOR. Использование данной стратегии приводит к трем операциям XOR для картинки на рис. 14.12.

Методическое отступление. Само изложение материала требует паузы для поиска и обсуждения различных вариантов построения стратегии поиска прямоугольников с заданными характеристиками. Методическим приемом в этой работе является поиск контрпримеров — для каждого варианта стратегии нужно найти пример, когда она дает не минимальное количество операций XOR. Вариации по организации деятельности школьников в этой «игре» огромны — от работы в командах до индивидуального творчества.

Методические рекомендации

Задачи на клеточном поле (одномерном или двумерном) — излюбленная олимпиадная проблематика. Достаточно вспомнить международные олимпиады 1991, 1993, 1994, 1997 и 1998 гг. В «идейном» плане поиска алгоритмического решения она не является сложной. Заслуживает внимания система оценки: безусловно, что она «приживется» на олимпиадах, ибо ей присущ соревновательный дух и она позволяет использовать задачи, для которых неизвестны точные алгоритмы решения. Целесообразно использовать ее и в повседневной работе с одаренными школьниками.

Самостоятельные исследования

Выясните, имеет ли задача точное решение? Другими словами, надо найти и обосновать алгоритм, выполняющий за минимальное количество операций XOR сведение произвольной картинки к белому экрану.

ЗАДАЧИ ВТОРОГО ТУРА

«Пакетная обработка заданий»

Условие задачи

Имеется последовательность n заданий, предназначенных для выполнения на одной машине. Задания пронумерованы от 1 до n следующим образом: 1, 2, ..., n . Задания должны быть так сгруппированы в один или несколько пакетов, чтобы каждый пакет состоял из следующих друг за другом заданий исходной последовательности. Выполнение первого задания начинается в момент времени 0. Пакеты обрабатываются один за другим, начиная с первого, следующим образом. Если пакет b содержит задания с меньшими номерами, чем пакет c , то пакет b

поступает на обработку раньше пакета s . Задания каждого пакета выполняются на машине последовательно. Сразу же после того, как все задания пакета выполнены, машина выводит результаты выполнения всех заданий этого пакета. Время завершения выполнения j -го задания определяется временем завершения обработки всего пакета, содержащего j -е задание.

Чтобы подготовить машину к обработке каждого пакета, необходимо время S , которое назовем подготовительным. Для каждого i -го задания известен стоимостный коэффициент F_i и время T_i , необходимое для выполнения этого задания. Если пакет состоит из заданий $x, x+1, \dots, x+k$ и поступает на обработку в момент времени t , то время завершения выполнения каждого задания пакета рассчитывается по формуле $t + S + (T_x + T_{x+1} + \dots + T_{x+k})$. Заметим, что машина выводит результаты всех заданий пакета в один и тот же момент времени. Если время завершения выполнения i -го задания — O_i , то стоимость выполнения этого задания составит $O_i \times F_i$. Пусть имеется 5 заданий и известно: $S = 1$; $(T_1, T_2, T_3, T_4, T_5) = (1, 3, 4, 2, 1)$; $(F_1, F_2, F_3, F_4, F_5) = (3, 2, 3, 3, 4)$. Если задания сгруппированы в три пакета $\{1, 2\}, \{3\}, \{4, 5\}$, то можно вычислить время завершения выполнения для них $(O_1, O_2, O_3, O_4, O_5) = (5, 5, 10, 14, 14)$ и стоимости выполнения заданий $(15, 10, 30, 42, 56)$, соответственно. Общая стоимость выполнения сгруппированных в пакеты заданий определяется как сумма стоимостей выполнения каждого задания. Таким образом, общая стоимость выполнения всех заданий для предложенного примера будет равна 153.

Вам задана величина подготовительного времени S и последовательность n заданий, для каждого из которых определено время выполнения и стоимостный коэффициент. Требуется написать программу, которая вычисляет минимально возможную общую стоимость выполнения последовательности n заданий.

Входные данные. Ваша программа должна читать данные со стандартного ввода. Первая строка содержит количество заданий n ($1 \leq n \leq 10000$). Вторая строка содержит подготовительное время S , которое является целым числом ($0 \leq S \leq 50$). Следующие n строк содержат информацию о заданиях $1, 2, \dots, n$ в указанном порядке. В каждой из этих строк первым задается целое число T_i ($1 \leq T_i \leq 100$) — время выполнения задания. За ним следует целое число F_i ($1 \leq F_i \leq 100$) — стоимостный коэффициент задания.

Выходные данные. Ваша программа должна записать в стандартный вывод одну строку, содержащую одно целое число — минимальную общую стоимость выполнения последовательности n заданий.

*Примеры входа и выхода.***Пример 1.****Ввод:**

```
2
50
100 100
100 100
```

Вывод:

```
45000
```

Пример 2.**Ввод:**

```
5
1
1 3
3 2
4 3
2 3
1 4
```

Вывод:

```
153
```

Пример 2 соответствует примеру из текста задачи.

Примечание. Для каждого теста общая стоимость любой группировки не превосходит $2^{31} - 1$.

Оценка решения. Если ваша программа выводит правильный ответ на тест за отведенное время, вы получаете полное количество баллов за этот тест, иначе — вы получаете 0 баллов.

Обсуждение задачи*

Перебор всех вариантов формирования пакетов имеет временную сложность $O(2^n)$. Каждое задание включается в пакет двумя способами: добавляется в последний формируемый пакет или начинает формировать новый пакет. При максимальном значении числа заданий, $n = 10000$, время решения недопустимо (оцените значение 2^{10000}).

Однако «дорогу осилит идущий». Через анализ простых примеров попытаемся найти приемлемый (по времени) алгоритм решения.

* Авторы благодарят М. А. Корчемкина за помощь в разборе задачи.

Первое задание, естественно, войдет в первый пакет: $\{1\}$. Второе задание будет составлять пару к первому — $\{1, 2\}$ или же начнет новый пакет — $\{1\}\{2\}$. Три задания можно сформировать в один пакет $\{1, 2, 3\}$, в два пакета двумя способами — $\{1, 2\}\{3\}$ или $\{1\}\{2, 3\}$, или в три пакета — $\{1\}\{2\}\{3\}$.

Пусть значение C_{ij} означает минимальную стоимость разбиения заданий с номерами от 1 до i на j пакетов. Естественно, что задание с номером i попадает в пакет с номером j . В последнем пакете вместе с заданием i может оказаться и несколько предыдущих заданий.

Пример. Значение C_{42} выбирается из стоимостей формирования заданий следующими способами: $\{1\}\{2, 3, 4\}$, $\{1, 2\}\{3, 4\}$, $\{1, 2, 3\}\{4\}$. Для вычисления C_{42} требуется знать C_{11} , C_{21} , C_{31} (это стоимости пакетов, выделенные «жирным» шрифтом) — к соответствующим стоимостям первого пакета прибавляется стоимость объединения оставшихся заданий во второй пакет. Так, для второго варианта формирования пакетов $\{1, 2\}\{3, 4\}$ значение C_{42} рассчитывается как $C_{21} + (S + T_1 + T_2 + S + T_3 + T_4) * (F_3 + F_4)$ или $C_{21} + (2S + T_1 + \dots + T_4) * (F_3 + F_4)$. Первое слагаемое — стоимость формирования первого пакета $\{1, 2\}$, второе слагаемое — стоимость формирования пакета $\{3, 4\}$, где первый множитель — время завершения обработки этого пакета, а второй множитель — стоимостный коэффициент. Аналогичная операция продлевается и для оставшихся двух вариантов, а выбранное минимальное значение есть C_{42} .

В общем виде значение C_{ij} вычисляется следующим образом: $C_{ij} = \min\{C_{k, j-1} + W_{k+1, i}(j)\}$, при $i > k > j$, где $W_{k+1, i}(j)$ — стоимость объединения заданий, начиная с номера $k+1$ и заканчивая номером i , в один пакет, притом, что этот пакет является j -м по счету. Таким образом, $W_{k+1, i}(j) = (jS + T_1 + \dots + T_i) * (F_{k+1} + \dots + F_i)$ — первый множитель — это время завершения обработки этого пакета, а второй множитель — сумма стоимостных коэффициентов, входящих в этот пакет. Следует дополнить, что $C_{i, 1} = W_{1, i}(1)$. Действительно, значение $C_{i, 1}$ вычисляется единственным образом и равно стоимости объединения заданий начиная с 1-го и заканчивая номером i , в первый пакет. Интересующий нас результат находится среди значений $C_{n, j}$ при $j = 1..n$.

Метод достаточно понятен и прост в реализации. Формальная запись основной части алгоритма выглядит так:

```

...
For i:=1 To n Do
  For j:=1 To i Do
    For k:=j-1 To i-1 Do
      C[i, j] = min(C[k, j-1] + W(k+1, i, j))
    ...
  
```

Временная сложность алгоритма сравнима с $O(n^3)$ (оцените — за какое время ваш компьютер выполнит 10000^3 операций). Сформулируем первую цель — убрать одну конструкцию повторения из решения, т. е. найти алгоритм с временной сложностью $O(n^2)$.

Суть идеи — распределение заданий по пакетам надо начинать с последнего задания.

Очевидное замечание. Если обработка пакета началось с некоторой задержкой времени dt , то стоимость его выполнения увеличится на величину $dt \cdot (F_p + \dots + F_q)$, где $p..q$ — номера заданий, входящих в этот пакет. Это следует из того, что для любого задания с номером k от p до q время его завершения увеличится на dt , и, следовательно, его стоимость увеличится на $dt \cdot F_k$. Суммирование по $k = p..q$ дает указанную величину.

Пример. Согласно этой идее, рассмотрим динамику процесса формирования пакетов заданий для примера из формулировки задачи. Итак, пусть $n = 5$. Последнее задание сформирует пакет {5}. Его стоимость обозначим как $C_5 = (S + T_5) \cdot F_5 = (1 + 1) \cdot 4 = 8$.

На очереди — четвертое задание. Возможны два варианта распределения заданий: {4, 5} и {4}{5}. Обозначим минимальную стоимость выполнения задач с 4-го до 5-го через C_4 . Подсчитаем для пакета {4, 5} значение $C_4 = (S + T_4 + T_5) \cdot (F_4 + F_5) = (1+2+1) \cdot (3+4) = 28$. А для пакетов {4}{5} значение $C_4 = (S + T_4) \cdot F_4 + [C_5 + (S + T_4) \cdot F_5]$, где первое слагаемое — стоимость пакета {4}, а второе (в квадратных скобках) — стоимость выполнения пакета {5}, равная C_5 плюс дополнительная стоимость этого пакета, вызванная его задержкой выполнения из-за обработки пакета {4}, а именно на время $S + T_4$ со стоимостным коэффициентом F_5 . Вынесем за скобки общий множитель $(S + T_4)$ и подсчитаем стоимость выполнения пакетов {4}{5}: $C_4 = C_5 + (S + T_4) \cdot (F_4 + F_5) = 8 + (1+2) \cdot (3+4) = 29$. Из двух значений выбирается наименьшее — $C_4 = 28$.

Рассмотрим третье задание. Возможны варианты: {3, 4, 5}; {3, 4}{5}; {3}{4, 5}; {3}{4}{5}. Однако вариант {3}{4}{5} рассматривать не следует, ибо уже известно, что он заведомо хуже, чем вариант {3}{4, 5}. На предыдущем шаге установлено, что стоимость пакетов {4}{5} больше, чем пакета {4, 5}, и при смещении времени выполнения заданий, вызванного обработкой задания 3, это соотношение не изменяется (!). Остается три варианта:

- {3, 4, 5}: $C_3 = (S + T_3 + T_4 + T_5) \cdot (F_3 + F_4 + F_5) = (1+4+2+1) \cdot (3+3+4) = 80$;
- {3, 4}{5}: $C_3 = (S + T_3 + T_4) \cdot (F_3 + F_4) + [C_5 + (S + T_3 + T_4) \cdot F_5] = C_5 + (S + T_3 + T_4) \cdot (F_3 + F_4 + F_5) = 8 + (1+4+2) \cdot (3+3+4) = 78$;
- {3}{4, 5}: $C_3 = (S + T_3) \cdot F_3 + [C_4 + (S + T_3) \cdot (F_4 + F_5)] = C_4 + (S + T_3) \cdot (F_3 + F_4 + F_5) = 28 + (1+4) \cdot (3+3+4) = 78$.

При этом не принципиально, какой из вариантов, дающих минимальное значение стоимости, выбран. Главное — что на следующей итерации задания с номерами 3, 4, 5 распределяются по пакетам оптимальным образом.

Продолжим обработку — теперь на очереди задание 2. Следует рассматривать распределение по пакетам: $\{2, 3, 4, 5\}$; $\{2, 3, 4\}\{5\}$; $\{2, 3\}\{4, 5\}$; $\{2\}\{3, 4\}\{5\}$. Жирным шрифтом здесь выделены распределения по пакетам меньшего количества заданий, дающих минимальное значение стоимости. Вместо распределения $\{2\}\{3, 4\}\{5\}$ можно было бы рассмотреть вариант $\{2\}\{3\}\{4, 5\}$, но это не изменит ситуацию, ибо в обоих случаях результат будет одинаковым. Этот факт следует из того, что задания 3, 4, 5, как бы они ни формировались по пакетам — первым или вторым способом, в итоге сдвигаются во времени выполнения на одно и то же значение, а именно на время выполнения пакета $\{2\}$. Расчет для всех этих вариантов дает:

- $\{2, 3, 4, 5\}$: $C_2 = (S + T_2 + \dots + T_5) * (F_2 + \dots + F_5) = 11 * 12 = 132$;
- $\{2, 3, 4\}\{5\}$: $C_2 = C_5 + (S + T_2 + \dots + T_4) * (F_2 + \dots + F_5) = 8 + 10 * 12 = 128$;
- $\{2, 3\}\{4, 5\}$: $C_2 = C_4 + (S + T_2 + T_3) * (F_2 + \dots + F_5) = 28 + 8 * 12 = 124$;
- $\{2\}\{3, 4\}\{5\}$: $C_2 = C_3 + (S + T_2) * (F_2 + \dots + F_5) = 78 + 4 * 12 = 126$.

Результат: $C_2 = 124$ для пакета $\{2, 3\}\{4, 5\}$.

Осталось обработать первое задание. Выполняя аналогичные действия, получаем окончательную оценку $C_1 = 153$ — минимальную стоимость распределения пяти заданий по пакетам. Минимальное значение достигается на каждом из распределений: $\{1, 2\}\{3\}\{4, 5\}$ или $\{1, 2\}\{3, 4\}\{5\}$.

Запишем в общем виде все описанные выше действия:

$$C_l(j) = C_j + \left(S + \sum_{k=1}^{j-1} T_k \right) * \sum_{k=j}^n F_k + \left(S + \sum_{k=1}^{j-1} T_k \right) * \sum_{k=1}^{j-1} F_k =$$

$$= C_j + \left(S + \sum_{k=1}^{j-1} T_k \right) * \sum_{k=1}^n F_k. \quad (1)$$

Значение j в записи $C_l(j)$ указывает на последнее задание, с которого начинается формирование пакетов. Первое слагаемое в равенстве дает минимальную стоимость формирования пакетов, начиная с задания j (для заданий с номерами от j до n найдено распределение по пакетам). Второе слагаемое определяет дополнительную стоимость пакета $\{j, \dots, n\}$, вызванную задержкой из-за выполнения пакета $\{i, \dots, j-1\}$. Третье слагаемое задает стоимость пакета $\{i, \dots, j-1\}$.

Введем обозначения: $G(i) = \sum_{k=i}^n T_k$, $H(i) = \sum_{k=i}^n T_k$, тогда равенство

(1) записывается в виде:

$$C_i(j) = C_j + (S + C(i) - G(j)) \cdot H(i). \quad (2)$$

Значение $C(i)$ равно минимальному значению из величин $C_i(j+1)$, $C_i(j+2)$, ..., $C_i(n+1)$ или

$$C_i = \min_{i < j \leq n+1} (C_i(j)). \quad (3)$$

Считаем, что $C_{n+1} = 0$, поэтому приведенные формулы верны для всех $i = 1 \dots n$.

Общий вид алгоритма можно записать примерно по следующей схеме:

```

...
For i:=n DownTo 1 Do
  For j:=i+1 To n+1 Do
    C[i]:=min{C[j]+(S+G[i]-G[j])*(F[i]-F[j])}
...

```

Этот алгоритм имеет временную сложность $O(n^2)$ — порядка 10^8 операций, что при ограничениях на время менее 1 секунды (100 мс) не гарантирует 100% результата. Итак, следующей целью нашего обсуждения является такая оптимизация алгоритма, чтобы в результате этих действий был получен алгоритм с временной сложностью $O(n)$.

Попробуем оптимизировать вышеописанную схему решения, стараясь выяснить, в каком случае выгоднее обрабатываемое задание добавлять к существующему пакету, а когда лучше начинать новый пакет. Вместо перебора значения j , соответствующего номеру задания, которое является последним в формируемом пакете $\{i, \dots, j\}$, попытаемся однозначно определять этот индекс.

Если на каком то шаге у нас получилось какое то определенное значение j , то все остальные индексы, большие, чем j , можно удалить из дальнейшего рассмотрения, поскольку они уже никогда не дадут оптимального разбиения. Данное утверждение довольно «прозрачно» и выводится разбором соотношений, приведенных выше. Удаление из рассмотрения индексов выше найденного оптимального j объясняется тем, что если они не дали оптимального значения на этом шаге, то и на следующих итерациях, при добавлении новых заданий на разбор, они не окажут влияния на результат. Само значение C_j может не обеспечить минимального значения, но значения C_{j+1} , C_{j+2} , и т. д. будут заведомо больше, чем C_j , следовательно, их не стоит рас-

смагивать. Это наблюдение позволяет изменить параметры цикла по j . Время алгоритма становится несколько меньше, чем $O(n^2)$, но по-прежнему сравнимо с этим значением. Такое сокращение отобразится на алгоритме примерно так:

```
high:=n+1;  
For i:=n DownTo 1 Do Begin  
  For j:=i+1 To high Do Begin  
    c[i] = min[c[j]+(S+G[i]-G[j])*(F[i]-F[j])];  
    <Если текущее j — претендент на оптимум, то  
    запомним это значение>;  
    jHigh:=j;  
  End;  
  high:=jHigh; {Изменяем верхнюю границу перебора по j}  
End;
```

Методическое отступление. Логика обсуждения задачи требует обязательного рассмотрения школьником собственных примеров: во-первых, достижение понимания алгоритма с линейной временной зависимостью без этого затруднительно; во-вторых, возможно самостоятельное открытие алгоритма (состояние «эврики») при небольшой подсказке — на каждой итерации по i следует выписывать и анализировать список «кандидатов» на оптимальное решение.

Вернемся к обсуждению задачи. Цикл по j можно сократить не только «сверху», но и «снизу»: оказывается, что при некоторых условиях на каждой итерации можно определить числа $jLow$ и $jHigh$, задающие диапазон индексов — «претендентов» на оптимальное значение. Причем они не обязательно идут подряд, поэтому целесообразнее организовать не сплошной цикл от $jHigh$ к $jLow$, а реализовать работу с *деком* (дек — структура данных, чтение и запись элементов которого осуществляются с двух сторон — с начала и с конца), используя для его реализации список (list), и извлекать соответствующие элементы из этой структуры. Таким образом, на i -й итерации при выборе оптимального значения допустимо «выбросить» неоптимальные номера заданий с конца дека, а при добавлении индекса i в список удалить и некоторые первые элементы дека.

Рассмотрим эту схему на примере. Критерием выбора тех или иных элементов списка является соотношение: $C_p - C_q \leq (G(p) - G(q)) * H(i)$, где p и q — два элемента из списка, причем $i < p < q$. Для его вычисления используется функция $dTime$, но пусть критерий в примере используется на интуитивном уровне, а объяснение мы дадим после примера.

```

Function dTime(p, q: Integer): Double;
Begin
  dTime := (C[p]-C[q]) / (G[p]-G[q]);
End;

```

При $S = 1$; $(T_1, T_2, T_3, T_4, T_5) = (1, 3, 4, 2, 1)$; $(F_1, F_2, F_3, F_4, F_5) = (3, 2, 3, 3, 4)$. Имеем: $(G_1, G_2, G_3, G_4, G_5) = (11, 10, 7, 3, 1)$; $(H_1, H_2, H_3, H_4, H_5) = (15, 12, 10, 7, 4)$. Для первой итерации доопределяем: $C_6 = G_6 = H_6 = 0$ и добавляем индекс 6 в список.

Первая итерация: $i = 5$, $list = (6)$, $C_5 = C_6 + (S + G_5 - G_6) * H_5 = 8$, и добавляем индекс 5 в список.

Вторая итерация: $i = 4$, $list = (5, 6)$, $dTime(5, 6) = (C_5 - C_6) / (G_5 - G_6) = 8/1 = 8 > H_4 = 7$; — элемент 6 не удаляем. Далее $C_4 = C_6 + (S + G_4 - G_6) * H_4 = 0 + (1 + 3 - 0) * 7 = 28$, — добавляем индекс 4 в список, и так как $dTime(4, 5) = (28 - 8) / (3 - 1) = 10 > dTime(5, 6) = 8$, то первый элемент (5) не удаляем.

Третья итерация: $i = 3$, $list = (4, 5, 6)$, $dTime(5, 6) = 8 \leq H_3 = 10$ — можем удалить элемент 6 из $list$. Далее $list = (4, 5)$, $dTime(4, 5) = 10 \leq H_3 = 10$ — можем удалить элемент 5 из $list$, и затем $list = (4)$, $C_3 = C_4 + (S + G_3 - G_4) * H_3 = 28 + (1 + 7 - 3) * 10 = 78$, добавляем индекс 3 в список.

Четвертая итерация: $i = 2$, $list = (3, 4)$, $dTime(3, 4) = (78 - 28) / (7 - 3) = 12,5 > H_2 = 12$, — элемент 4 не удаляем, $C_2 = C_4 + (S + G_2 - G_4) * H_2 = 28 + (1 + 10 - 3) * 12 = 124$, — добавляем индекс 2 в список, и так как $dTime(2, 3) = (124 - 78) / (10 - 7) = 15,33 > dTime(3, 4) = 12,5$, то первый элемент не удаляем.

Пятая итерация: $i = 1$, $list = (2, 3, 4)$, $dTime(3, 4) = 12,5 \leq H_1 = 15$ — можем удалить элемент 4 из $list$, и далее $list = (2, 3)$, $dTime(2, 3) = 15,33 > H_1 = 15$, $C_1 = C_3 + (S + G_1 - G_3) * H_1 = 78 + (1 + 11 - 7) * 15 = 153$.

Методическое отступление. Определение (через рассмотрение собственных примеров) логики работы с деком является последним «штрихом» в понимании решения.

Итак, критерий действий с деком (реализованным через список) определен через соотношение: $C_p - C_q \leq (G(p) - G(q)) * H(i)$, где p и q — два элемента из списка, причем $i < p < q$. Если указанное соотношение выполняется, т. е. разница стоимостей формирования заданий будет меньше, чем значение дополнительной стоимости от «сдвига» времени обработки пакетов, то мы, естественно, «сдвигаем» во времени все задания, начиная с p -го. Таким образом, объединять задания в пакет следует так: $\{i, \dots, p-1\}$, а далее идет оптимальный «хвост», начиная с задания с номером p . Все значения выше p , в том числе и q , можно

отбросить. Отсюда следует, что если в качестве p и q последовательно брать предпоследний и последний элементы списка, соответственно, то при выполнении указанного условия второй из них можно отбросить. Таким образом, в результате получается, что в конце списка стоит индекс, дающий оптимальное значение. А при добавлении i -го элемента в начало списка? Если элементы p и q — первый и второй элементы списка, соответственно, и выполняется соотношение $(C_i - C_q) / (G(i) - G(q)) \leq (C_p - C_q) / (G(p) - G(q))$, то это говорит о том, что индекс p , а именно первый элемент списка, можно убрать из дальнейшего рассмотрения. Указанное сравнение следует продолжать уже с новыми первым и вторым элементами списка. Для вычисления частного от деления (назовем его «относительной стоимостью») использована функция `dTime`.

Для «наброска» программной реализации воспользуемся нотацией *Delphi*. Пусть дан класс `__List` и его методы: `GetFirst`, `GetSecond`, `GetLast`, `GetPreLast`, `GetPower`, которые возвращают значения первого, второго, предпоследнего и последнего элементов, соответственно, а также значение количества элементов в списке. Тогда фрагмент кода имеет вид:

```

...
Var list : __List;
      i, p, q : Integer;
...
list.Insert(n+1); {Добавляем дополнительный элемент}
For i := n DownTo 1 Do Begin
  p := list.GetPreLast; {Последний элемент списка}
  q := list.GetLast; {Предпоследний элемент списка}
  While (list.GetPower > 1) And (dTime(p,q) <= h[i]) Do
    Begin
      list.DeleteLast; {Удаляем с конца списка}
      p := list.GetPreLast; {Последний элемент списка}
      q := list.GetLast; {Предпоследний элемент списка}
    End;
  p := list.GetLast;
  C[i] := C[p] + (S+G[i]-G[p])*H[i];
  p := list.GetFirst;
  q := list.GetSecond;
  While (list.GetPower > 1) And (dTime(i,q) <=
dTime(p,q)) Do Begin
    list.DeleteFirst; {Удаляем с начала списка}
    p := list.GetFirst;
    q := list.GetSecond;
  End;

```

```
list.Insert(i); {Добавляем элемент в начало списка}  
End;  
WriteLn(c[1]);  
...
```

Поскольку каждый индекс добавляется в очередь, а значит, и удаляется из нее не более одного раза, то общее время работы алгоритма есть $O(n)$. Данное решение, видимо, и оценивалось на олимпиаде в полный балл.

Методические указания

С одной стороны, это задача на реализацию классической схемы динамического программирования, а с другой — поиск решения с линейной временной зависимостью переводит ее в разряд повышенной сложности. Первые варианты реализации могут быть использованы при обычном изучении темы, а последний, показывающий, откуда проистекает «олимпиадность» задач по информатике, — при подготовке к соревнованиям различных уровней. Другой вариант разбора задачи можно найти в работе М. Бабенко, В. Матюхина и П. Митричева*.

Самостоятельные исследования

Экспериментальное определение размерности задачи (n), для которой различные алгоритмы укладываются в отведенное время, является традиционным вопросом для анализа.

«Автобусные остановки»

Условие задачи

В городе Yong-In планируется создать сеть автобусных маршрутов с n автобусными остановками. Каждая остановка находится на некотором перекрестке. Поскольку Yong-In — современный город, на карте он представлен улицами с квадратными кварталами одинакового размера. Две из n остановок выбираются пересадочными станциями h_1 и h_2 , которые будут соединены друг с другом прямым автобусным маршрутом, а каждая из оставшихся $n-2$ остановок будет непосредственно соединена маршрутом с одной пересадочной станцией h_1 или h_2 (но не с обеими) и не соединена ни с одной из оставшихся остановок.

* Бабенко М., Матюхин В., Митричев П. Решение задач международной олимпиады школьников по информатике 2002 года. Пакетная обработка заданий. // «Информатика», еженедельное приложение к газете «Первое сентября». 2003. № 7. С. 24–26.

Расстояние между любыми двумя остановками определяется как длина кратчайшего пути по улицам города. Это означает, что если остановка представлена парой координат (x, y) , то расстояние между двумя остановками (x_1, y_1) и (x_2, y_2) будет равно $|x_1 - x_2| + |y_1 - y_2|$. Если остановки a и b соединены с одной и той же пересадочной станцией, например h_1 , то длина пути из a в b является суммой расстояний от a до h_1 и от h_1 до b . Если автобусные остановки a и b соединены с разными пересадочными станциями, например a с h_1 и b с h_2 , то длина пути из a в b является суммой расстояний от a до h_1 , от h_1 до h_2 и от h_2 до b .

Проектировщики города Yong-In хотят быть уверенны, что любой житель города сможет добраться до любой точки города достаточно быстро. Поэтому проектировщики хотят сделать пересадочными станциями такие две остановки, чтобы в полученной сети автобусных маршрутов максимальная длина пути между любыми двумя остановками была минимальной.

Вариант p выбора пересадочных станций и соединения остановок с ними будет лучше варианта q , если максимальная длина пути между любыми двумя остановками в варианте p будет меньше, чем в варианте q . Ваша задача — написать программу вычисления максимальной длины пути между любыми двумя остановками для наилучшего варианта p выбора пересадочных станций и соединения остановок с ними.

Входные данные. Ваша программа должна читать стандартный ввод. Первая строка содержит положительное целое число n ($2 \leq n \leq 500$) — количество остановок. Каждая из оставшихся n строк содержит пару чисел x и y — координаты автобусной остановки. Координаты x и y — положительные целые числа, не превосходящие 5000. Никакие две остановки не могут быть представлены одной и той же парой координат.

Выходные данные. Ваша программа должна записать в стандартный вывод одну строку с единственным положительным числом — минимальной возможной длиной максимального пути между остановками.

Примеры ввода и вывода.

Пример 1.

Ввод:

```
6
1 7
16 6
12 4
4 4
1 1
11 1
```

Вывод:

20

Пример 2.

Ввод:

7

7 9

10 9

5 3

1 1

7 2

15 6

17 7

Вывод:

25

На рис. 14.13 показана сеть автобусных маршрутов для первого примера. Если выбрать остановки 3 и 4 как пересадочные станции, то самый длинный путь будет или между остановками 2 и 5, или между остановками 2 и 1. Поскольку лучшего варианта выбрать пересадочные станции нет, ответом будет 20.

На рис. 14.14 показана сеть автобусных маршрутов для второго примера. Если в качестве пересадочных станций выбраны остановки 5 и 6, то самый длинный путь будет между остановками 2 и 7. Поскольку лучших вариантов нет, ответом будет 25.

Оценка решения. Если ваша программа выводит правильный ответ в отведенное для теста время, вы получите все баллы за этот тест, иначе вы получите за этот тест 0 баллов.

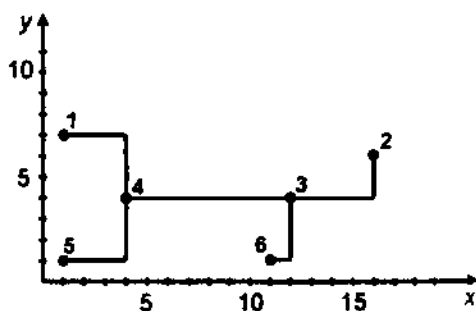


Рис. 14.13. Сеть автобусных маршрутов для первого примера

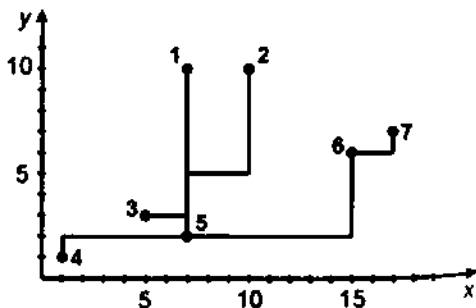


Рис. 14.14. Сеть автобусных маршрутов для второго примера

Обсуждение задачи

В качестве сквозного примера при обсуждении задачи мы будем рассматривать первый пример из формулировки задачи. Подсчитать расстояния между всеми парами остановок при вводе данных не составляет труда. Для этого примера матрица расстояний w имеет вид:

$$w = \begin{pmatrix} 0 & 16 & 14 & 6 & 6 & 16 \\ 16 & 0 & 6 & 14 & 20 & 10 \\ 14 & 6 & 0 & 8 & 14 & 4 \\ 6 & 14 & 8 & 0 & 6 & 10 \\ 6 & 20 & 14 & 6 & 0 & 10 \\ 16 & 10 & 4 & 10 & 10 & 0 \end{pmatrix}.$$

Пусть в качестве пересадочных выбраны станции с номерами i и j . Нам требуется оценить расстояние между остановками a и b (обозначим его как $r(a, b)$).

Возможны четыре способа их соединения с пересадочными станциями:

$$r(a, b) = \begin{cases} w[a, i] + w[i, j] + w[j, b]; \\ w[a, j] + w[j, i] + w[i, b]; \\ w[a, i] + w[i, b]; \\ w[a, j] + w[j, b]. \end{cases}$$

Выбираем наилучший способ соединения. Просматривая все пары (a, b) , получаем оценку для пересадочных станций i и j . «Пробегаая» по всем парам (i, j) , получаем ответ задачи.

Очевидно, что временная сложность такого алгоритма $O(n^4)$. Здравый смысл подсказывает, что на международной олимпиаде не может быть такой простой задачи, и такое решение не наберет полный балл. Действительно, $n = 500$, количество действий имеет порядок $6,25 \cdot 10^{10}$, а время на тест — 4 секунды. Следовательно, необходим поиск решения с временной сложностью $O(n^3)$.

Единственный путь уменьшения временной сложности решения — это ввести некую упорядоченность на множестве станций и уйти от рассмотрения всех пар (a, b) . От просмотра же пар (i, j) нам не избавиться при всем желании — пару для пересадочных станций мы обязаны в любом случае определять и оценивать, насколько хороша пара (i, j) в этом качестве.

Введение упорядоченности заключается в том, что для каждой станции следует отсортировать массив расстояний до остальных станций по возрастанию. «Трогать» матрицу w с этой целью нецелесообразно (в другой структуре данных потребуется хра-

нить номера столбцов в соответствии с отношением порядка в каждой строке). Достаточно ввести матрицу d , где в каждой строке будут записаны номера станций в порядке возрастания расстояний до них:

$$d = \begin{pmatrix} 1 & 5 & 4 & 3 & 6 & 2 \\ 2 & 3 & 6 & 4 & 1 & 5 \\ 3 & 6 & 2 & 4 & 5 & 1 \\ 4 & 5 & 1 & 3 & 6 & 2 \\ 5 & 1 & 4 & 6 & 3 & 2 \\ 6 & 3 & 4 & 5 & 2 & 1 \end{pmatrix}.$$

Используя один из методов сортировки с временной сложностью $O(n^* \log_2 n)$ (например, *пирамидальную сортировку*), эта работа выполняется за время, пропорциональное $O(n^2 \log_2 n)$.

Что дает нам эта предварительная работа? Рассмотрим пару станций (i, j) в качестве пересадочных и предположим, что все остальные станции соединены с j . Пусть $i = 3$ и $j = 4$. Мы описали один из способов соединения остановок с данной парой пересадочных станций, но не только. В этом случае у нас есть и оценка этого способа, причем для ее вычисления достаточно взять две самые удаленные от j станции a, b (а их номера есть в $d[4, 6]$, $d[4, 5]$) и сложить расстояния: $w[j, a] + w[j, b]$.

На рис. 14.15а показана эта ситуация. Попытаемся обобщить это положение. Предположим, что некая станция соединена не с j , а с i . Что скрывается за словом «некая»? Мы просто «идем» по строке $d[3]$ и просматриваем станции с номерами 3, 6, 2, 4, 5 и 1 (переменная t). Эти ситуации отражены на рис. 14.15. У нас больше нет временных ресурсов (выбор пары i и j — $O(n^2)$; цикл по элементам строки d — $O(n)$, общая сложность — $O(n^3)$). Другими словами, оценку максимального расстояния в этих условиях требуется выполнять за линейное время. Но так оно и получается (рис. 14.15б). В качестве значения b берется следующий номер с конца из строки $d[4]$. Так как 3 брать нельзя, то берем 1 и оцениваем максимальное расстояние (рис. 14.15б). В общем случае, когда $a \neq 0$, $b \neq 0$ и $t > 1$, оно выбирается как максимальное значение (r) из четырех чисел:

$$r = \text{Max} \begin{cases} w[j, a] + w[j, b]; \\ w[i, t] + w[i, t - 1]; \\ w[i, t] + w[i, j] + w[j, a]; \\ w[i, t] + w[i, j] + w[j, b]. \end{cases}$$

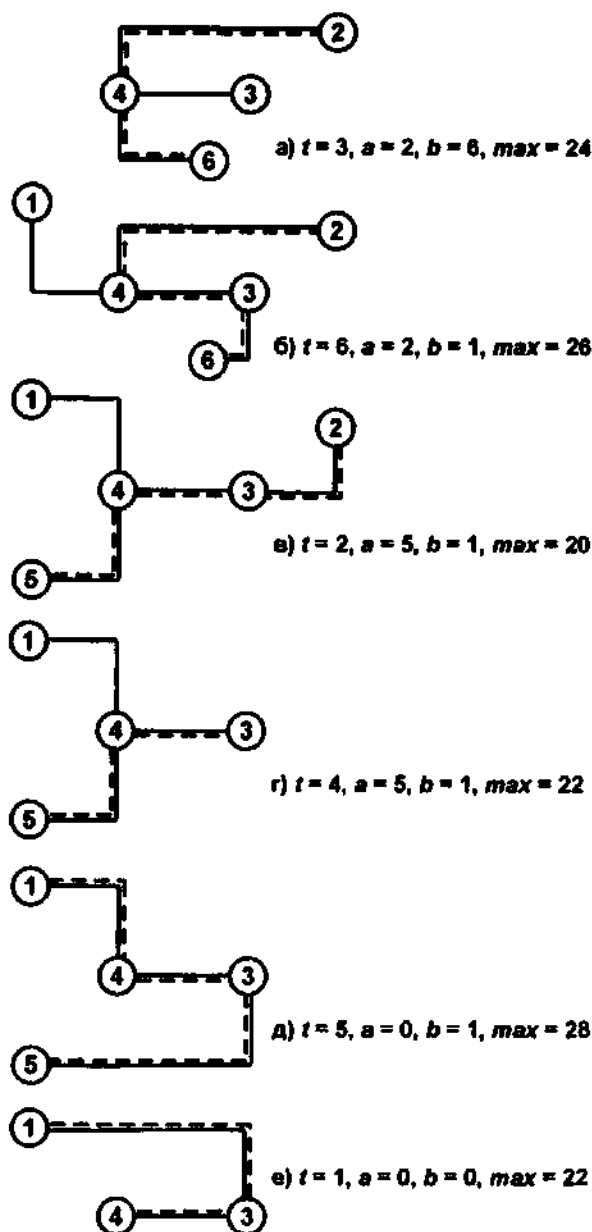


Рис. 14.15. Оценка максимального расстояния при пересадочных станциях с номерами 3 и 4

Методическое отступление. Целесообразно привести еще один пример и для некоторой выбранной пары пересадочных станций провести аналогичный расчет данных с обоснованием второй строки приведенной выше формулы.

Просмотренные номера станций (i) целесообразно отмечать признаком, например False, в дополнительном массиве. Его значение с привязкой к шагам, показанным на рис. 14.15, приведено в табл. 14.1.

Таблица 14.1

Шаг	1	2	3	4	5	6
а)	True	True	False	True	True	True
б)	True	True	False	True	True	False
в)	True	False	False	True	True	False
г)	True	False	False	False	True	False
д)	True	False	False	False	False	False
е)	False	False	False	False	False	False

Напомним, что значения a и b выбираются с конца строки с номером j матрицы d (в нашем примере $d[4]$) среди тех номеров, которые отсечены признаком True.

На этом обсуждение можно закончить и перейти к практической работе. Доказывать математические утверждения* допустимо, но только после того, как школьник «прочувствовал» все особенности задачи при проведении экспериментальной работы с реализованным решением.

Методические указания

Данная задача не является сложной по формулировке и при первом варианте решения. Следует обратить внимание на «коварство» жюри — решение с временной сложностью $O(n^4)$ получает только часть полного балла, а со сложностью $O(n^3)$ удовлетворяет требованиям задачи. Осознать эту разницу во времени при практической работе можно только при наличии достаточно полной системы тестов и проверяющей программы.

Самостоятельные исследования

Обобщение задачи для трех пересадочных станций и «уход» от решения простым перебором может оказаться достаточно нетривиальной задачей.

* *Бабенко М., Матюхин В., Митричев П.* Решение задач международной олимпиады школьников по информатике 2002 года. Автобусные остановки. // «Информатика», Еженедельное приложение к газете «Первое сентября». 2003. № 2. С. 21–22.

«Две дорожки»

Условие задачи

Дорожка представляет собой горизонтальную или вертикальную последовательность клеток сетки, состоящую не менее чем из двух последовательных клеток. Две дорожки расположены на сетке размером n на n . Одна дорожка горизонтальная, а другая — вертикальная. На рис. 14.16 эти две дорожки помечены символом x . Дорожки могут быть одинаковой или разной длины. Дорожки могут иметь общую клетку, а могут не иметь таковой. Если на диаграмме, такой, как на рис. 14.16, можно интерпретировать клетку, например (4, 4), как принадлежащую только одной дорожке или обеим дорожкам, то следует считать, что данная клетка принадлежит и той, и другой дорожке. Следовательно, на рис. 14.16 верхняя клетка вертикальной дорожки будет (4, 4), но не (5, 4).

В начале не известно, где расположены дорожки. Ваша задача — написать программу, определяющую их положение. Назовем горизонтальную дорожку **ДОРОЖКА1**, а вертикальную — **ДОРОЖКА2**. Каждая клетка сетки представлена парой чисел (r, c) , задающих номер строки и столбца, соответственно. Левая верхняя клетка сетки имеет координаты (1, 1). Каждая дорожка задается парой клеток: $\langle (r_1, c_1), (r_2, c_2) \rangle$. На рис. 14.16 **ДОРОЖКА1** задается как $\langle (4, 3), (4, 8) \rangle$, а **ДОРОЖКА2** — как $\langle (4, 4), (9, 4) \rangle$.

При решении данной задачи вы должны использовать библиотеку для получения входных данных, для поиска решения и для выдачи ответа. Размер n квадратной сетки возвращается библиотечной функцией `gridsize`, которую вы должны вызвать в начале работы. Для определения положений дорожек вы можете использовать только библиотечную функцию `rect(a, b, c, d)`, которая получает на входе координаты прямоугольной области и возвращает число 1, если в указанной области встречается хотя бы одна клетка, принадлежащая хотя бы одной дорожке, и число 0 — в противном случае. Область задается в виде $[a, b] \times [c, d]$, где $a \leq b$ и $c \leq d$, при этом необходимо обратить внимание на порядок

			$\downarrow c$			$\downarrow d$			
	1	2	3	4	5	6	7	8	9
1									
2									
A → 3									
4			X	X	X	X	X	X	
5				X					
6				X					
7				X					
B → 8				X					
9				X					

Рис. 14.16. Пример расположения двух дорожек

аргументов. Например, на рис. 14.16 проверяемая область закрашена серым цветом. От вас требуется написать программу, определяющую точное положение дорожек, используя ограниченное число вызовов функции `rect`.

Ваша программа должна выдавать ответ с помощью вызова библиотечной функции `report(r1, c1, r2, c2, p1, q1, p2, q2)`, где пара $\langle (r_1, c_1), (r_2, c_2) \rangle$ задает *ДОРОЖКУ1*, а пара $\langle (p_1, q_1), (p_2, q_2) \rangle$ — *ДОРОЖКУ2*. Вызов функции `report` приведет к завершению работы вашей программы. Обратите внимание, что *ДОРОЖКА1* — горизонтальная, а *ДОРОЖКА2* — вертикальная, и (r_1, c_1) — координаты левого конца горизонтальной *ДОРОЖКИ1*, а (p_1, q_1) — координаты верхнего конца вертикальной *ДОРОЖКИ2*. Таким образом, $r_1 = r_2$, $c_1 < c_2$, $p_1 < p_2$ и $q_1 = q_2$. Если параметры функции `report` не удовлетворяют этим ограничениям, то стандартный вывод будет содержать сообщение об ошибке.

Ограничения:

- входные данные доступны только через вызовы библиотечных функций `gridsize` и `rect`;
- размер сетки n удовлетворяет ограничению $5 \leq n \leq 10000$;
- число вызовов функции `rect` не должно превышать 400 для любого теста. Если ваша программа совершит более 400 вызовов функции `rect`, то это приведет к ее досрочному завершению;
- ваша программа должна выполнить более одного вызова функции `rect`, и ровно один раз — вызов функции `report`;
- если вызов функции `rect` будет произведен с некорректными параметрами, то это приведет к досрочному завершению вашей программы;
- ваша программа не должна считывать или записывать какие-либо файлы, а также использовать стандартный ввод/вывод.

Библиотека. В вашем распоряжении находится библиотека, включающая:

- библиотеку *FreePascal* (`prectlib.ppu`, `prectlib.o`):

```
function gridsize: LongInt;
function rect(a,b,c,d : LongInt) : LongInt;
procedure report(r1, c1, r2, c2, p1, q1, p2, q2 : LongInt);
```

Инструкции: при компиляции вашего файла `rods.pas` используйте оператор `uses prectlib;` и командную строку `fpc -So -O2 -XS rods.pas`.

Программа `prodstool.pas` представляет собой пример использования данной библиотеки *FreePascal*.

• библиотеку *GNU C/C++* (crectlib.h, crectlib.o):

```
int gridsize();
int rect(int a, int b, int c, int d);
void report(int r1, int c1, int r2, int c2, int p1,
            int q1, int p2, int q2);
```

Инструкции: при компиляции вашего файла rods.c используйте директиву `#include "crectlib.h"` и командные строки:

```
gcc -O2 -static rods.c crectlib.o -lm
g++ -O2 -static rods.cpp crectlib.o -lm
```

Программа prodstool.c представляет собой пример использования данной библиотеки *GNU C/C++*.

Для *C/C++* в среде RHIDE убедитесь, что в опциях Option->Linker указано crectlib.o.

Экспериментирование. Для проведения экспериментов с библиотекой вы должны создать текстовый файл rods.in. Файл должен состоять из трех строк. В первой строке файла должно быть записано число n , размер сетки. Вторая строка должна содержать координаты **ДОРОЖКИ1**, r_1 c_1 r_2 c_2 , где (r_1, c_1) — координаты левого конца дорожки. Третья строка должна содержать координаты **ДОРОЖКИ2** p_1 q_1 p_2 q_2 , где (p_1, q_1) — координаты верхнего конца дорожки.

После запуска вашей программы с вызовом функции report будет создан выходной файл rods.out. Этот файл будет содержать число вызовов функции rect и координаты концов дорожек, которые вы передадите при вызове функции report. Если в процессе работы программы произошли какие-либо ошибки или нарушения требований условия при обращении к библиотеке, то файл rods.out будет содержать соответствующее сообщение об ошибке.

Диалог между вашей программой и библиотекой записывается в файл rods.log. Этот файл rods.log будет содержать последовательность вызовов функции rect в форме « k : rect(a, b, c, d) = ans». Это означает, что k -й вызов функции rect(a, b, c, d) вернул число ans.

Пример ввода и вывода.

```
rods.in:
9
4 3 4 8
4 4 9 4

rods.out:
20
4 3 4 8
4 4 9 4
```

Система оценки. Если ваша программа нарушает какое-либо вышеназванное ограничение (например, число вызовов функции `rect` более 400) или если ответ вашей программы (положение дорожек) некорректный, то вы получаете 0 баллов. Если ответ вашей программы правильный, то ваши баллы будут зависеть от количества вызовов функции `rect` на каждом тесте. Если это число не превышает 100, то вы получаете 5 баллов, если оно от 101 до 200, то вы получаете 3 балла, а если от 201 до 400, то вы получаете 1 балл.

Обсуждение задачи*

Первый шаг решения заключается в уменьшении области поиска. Находим прямоугольник, содержащий обе дорожки, такой, что его размеры уменьшить нельзя. Для дорожек на рис. 14.16 таким прямоугольником является область $[4, 9] \times [3, 8]$. Использование бинарного поиска для этой цели очевидно. Последовательно «урезаем» область поиска сверху, снизу, слева и справа. Критерий — нет ли одной клетки дорожки. Количество запросов при однократном урезании (с одной стороны) совпадает с количеством сравнений при бинарном поиске, т. е. равно $\lceil \log_2 n \rceil$, а всего $4 \cdot \lceil \log_2 n \rceil$, что при $n = 10000$ составит 56 запросов. Пусть таким способом найден прямоугольник (a, b, c, d) , где (a, c) и (b, d) — координаты противоположных вершин.

Второй шаг решения заключается в поиске дорожек, но уже только в найденном прямоугольнике. В качестве критерия различных вариантов расположения дорожек в прямоугольнике определим количество занятых углов: нет углов (значение «нуль»); два угла на одной стороне; два противоположных угла; три угла. Эти варианты (рис. 14.17) с точностью до поворотов и осевых симметрий описывают все возможные случаи расположения дорожек в прямоугольнике. Выбор варианта расположения дорожек прямоугольника осуществляется за четыре запроса `rect`. Рассмотрим их.

Первый вариант (рис. 14.17а) — нет занятых углов. Требуется найти занятую клетку, которая полностью определяет вертикальную дорожку. Для этого достаточно использовать бинарный поиск, например, в прямоугольнике (a, a, c, d) . Най-



Рис. 14.17. Четыре варианта взаимного расположения дорожек

* Бабенко М., Матюхин В., Митричев П. Решение задач международной олимпиады школьников по информатике 2002 года. Две дорожки. // «Информатика», еженедельное приложение к газете «Первое сентября», 2003. № 3. С 23–25.

денное значение t (занятая клетка) определит координаты дорожки (a, t) , (b, t) . Аналогично для другой дорожки — выполнением бинарный поиск в прямоугольнике (a, c, b, c) , находим занятую клетку r и, соответственно, дорожку (r, c) и (r, d) .

Второй вариант — заняты два угла по одной из сторон прямоугольника (a, b, c, d) . В зависимости от занятых углов, возможны четыре способа размещения дорожек (рис. 14.18). Для опреде-

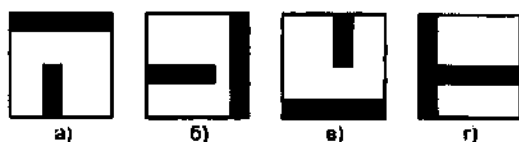


Рис. 14.18. Уточнение второго варианта расположения дорожек

ления координат второй дорожки последовательно по два раза применяем бинарный поиск. Первый случай — рис. 14.18а: поиск в прямоугольнике (b, b, c, d) — дает значение t , а затем нахождение верхнего конца вертикальной дорожки в прямоугольнике $(a+1, b, t, t)$. Аналогично задача решается и для остальных трех способов (рис. 14.18б, в, г).

Третий вариант — заняты два противоположных угла прямоугольника (a, b, c, d) . Условие задачи — одна вертикальная и одна горизонтальная дорожка — ограничивает число возможных способов их размещения. С точки зрения общей логики (занятые углы) их два: первый, третий углы (нумерация по часовой стрелке) и второй, четвертый углы. Но каждый из них имеет два различных «подспособа». Они все приведены на рис. 14.19. Пусть заняты первый и третий углы — рис. 14.19а, в. Дополнительная проверка, например, $\text{гес}(a, c+1, a, c+1)$ разделяет эти «подспособы». После этого для первого случая выполняется бинарный поиск в прямоугольнике (a, b, d, d) для определения вертикальной дорожки и бинарный поиск в прямоугольнике (a, a, c, d) — для горизонтальной дорожки; для остальных случаев — аналогично.

И наконец, последний вариант — три занятых угла. Он, по номерам углов, разбивается на четыре способа размещения дорожек. На рис. 14.20 представлен один такой способ —

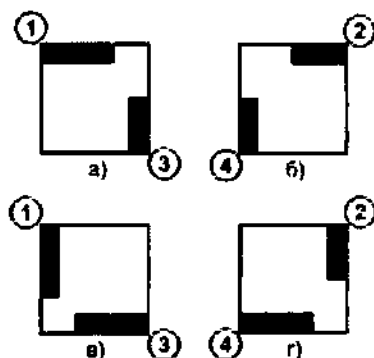


Рис. 14.19. Уточнение третьего варианта расположения дорожек

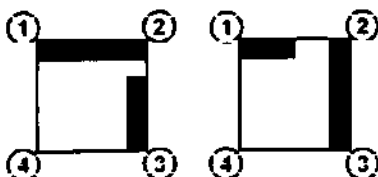


Рис. 14.20. Один из случаев четвертого варианта расположения дорожек

заняты углы 1, 2 и 3. Видим, что каждый способ разбивается на два «подспособа», определение которых требует одного дополнительного запроса, например $\text{rec}(a, a, d-1, d-1)$, а затем, как и в предыдущих вариантах, выполнения двух бинарных поисков.

Итак, оценивая количество запросов, можно утверждать, что в худшем случае при $n = 10000$ оно не превысит значения $90 (4 * \lceil \log_2 n \rceil + 4 + 1 + 2 * \lceil \log_2 n \rceil)$.

Методические рекомендации

Это задача на тему «Поиск». Она не относится к разряду сложных, но с методической точки зрения имеет ряд достоинств. Во-первых, при наличии библиотеки (а сделать ее несложно) и подготовленных тестов после обсуждения организуются соревнования на лучшее решение (на минимальность количества запросов). Во-вторых, такое соревнование можно устроить и на лучший программный код — имеется в виду не «бесконечное» использование логики бинарного поиска для всех вариантов и способов, а написание одной или двух функций бинарного поиска.

Самостоятельные исследования

1. Снимем ограничение на обязательность присутствия горизонтальной и вертикальной дорожек. Пусть обе они могут быть как горизонтальными, так и вертикальными.

2. Увеличим количество дорожек до трех, четырех и т. д.

3. «Морской бой*». На клетчатом листе бумаги размера $m \times n$ нарисованы корабли. Каждый корабль представляет собой вертикальный или горизонтальный набор подряд идущих закрашенных клеток. Разные корабли не соприкасаются по сторонам или углам и не накладываются друг на друга. В отличие от обычного «морского боя», здесь могут быть корабли более чем из четырех клеток. Необходимо найти количество кораблей.

Входные данные. Входной файл содержит в первой строке два числа, разделенные пробелом, — размеры листа бумаги m и n , $2 \leq m, n \leq 100$. Далее следует таблица из m строк по n целых чисел, разделенных пробелами, состоящая из 0 или 1 (0 — если клетка пустая, 1 — если она входит в состав какого-то корабля); одна строка таблицы располагается в одной строке файла.

Выходные данные. Выходной файл должен содержать количество кораблей на листе.

4. В приведенной выше задаче «Морской бой» найдите не только количество кораблей, но и их координаты.

* Воронин А. В., Кузнецов В. А., Корзун Д. Ж. Командные олимпиады по программированию: организация, задачи и решения / А. В. Воронин. Петрозаводск: ПетрГУ, 2001. С. 182.

ЗАДАЧИ ПЕРВОГО ТУРА

«Выбор дорог» (интерактивная задача)

Условие задачи

Коровы фермера Джона желают свободно ходить по n ($1 \leq n \leq 200$) последовательно пронумерованным от 1 до n полям фермы, даже если поля разделены лесом. Коровы выбирают систему дорог между парами полей таким образом, чтобы можно было пройти от любого поля к любому другому полю, двигаясь по дорогам. Коровы могут ходить по дорогам в любом направлении.

Для дорог коровы могут использовать только тропы диких животных. Каждую неделю они могут выбрать для использования все или некоторые тропы диких животных, о которых им известно.

Интересно, что коровы всегда обнаруживают ровно одну новую тропу диких животных в начале каждой недели. Они должны принять решение о множестве троп, которые будут использоваться в качестве системы дорог в течение текущей недели.

Коровы могут выбирать любое подмножество троп диких животных независимо от того, какие тропы выбирались на предыдущей неделе.

Коровы всегда хотят минимизировать суммарную длину дорог, которые они собираются выбрать.

Тропы диких животных не являются прямыми. Две тропы, которые соединяют одни и те же два поля, могут иметь разную длину. Более того, в случае пересечения двух троп (что возможно только вне поля) коровы не могут переходить с одной тропы на другую в точке их пересечения.

В начале каждой недели имеется информация о тропе диких животных, которую обнаружили коровы. На основании этого ваша программа должна выводить минимальную общую длину дорог, выбранных коровами на этой неделе, либо указывать, что искомая система дорог не существует.

Входные данные (стандартный вход):

- первая строка входного файла содержит два целых числа, разделенных пробелом, n и w . w — количество недель, которые должна обработать программа ($1 \leq w \leq 6000$);

- для каждой недели считывайте одну строку, содержащую информацию о вновь обнаруженной тропе диких животных. Эта строка содержит три целых числа, разделенных пробелами: концевые точки тропы (номера полей) и целочисленную длину тропы (1...10000). Не бывает тропы, у которой концевые точки находятся на одном поле.

Выходные данные (стандартный выход). Сразу после того, как ваша программа узнает о вновь обнаруженной тропе диких животных, она должна вывести одну строку, содержащую минимальную общую длину дорог, которые следует выбрать, чтобы связать все поля. Если невозможно связать все поля системой дорог, используя только тропы диких животных, то следует вывести «-1».

Ваша программа должна заканчивать работу после вывода ответа для последней недели.

Пример диалога приведен в табл. 15.1.

Таблица 15.1

Input	Output	Пояснения
4 6		
1 2 10		
	-1	Поле 4 не соединено с другими полями
1 3 8		
	-1	Поле 4 не соединено с другими полями
3 2 3		
	-1	Поле 4 не соединено с другими полями
1 4 3		
	14	Выбрать 1 4 3, 1 3 8 и 3 2 3
1 3 6		
	12	Выбрать 1 4 3, 1 3 6 и 3 2 3
2 1 2		
	8	Выбрать 1 4 3, 2 1 2 и 3 2 3
	Завершение работы программы	

Ограничения:

Время выполнения	1 second of CPU
Память	64 MB

Система оценки. За каждый тест, на котором ваша программа выведет правильный ответ, вы получите полный балл. Неоптимальный ответ на тест не оценивается.

Идея решения

Эта задача для 2003 года достаточно проста и для уровня международной олимпиады носит скорее «утешительный» характер. «Увидеть» граф из формулировки задачи не представляет особой сложности, — она просто «толкает» на это понимание. Следующий шаг, а именно нахождение каркаса минимального веса для взвешенного неориентированного графа с помощью известных алгоритмов Дж. Краскала и Р. Прима*, казалось бы, также очевиден и вытекает из формулировки задачи. Но не следует забывать о «коварстве» жюри: время прохождения теста ограничено одной секундой! Так что полный балл такое очевидное решение не получает.

Но следует ли строить на каждом шаге каркас (сеть дорог) минимального веса по всем введенным ребрам графа? В условии задачи даны *недели* как механизм «поступления» ребер графа. Каркас может быть получен, в лучшем случае, только на неделе $n-1$, если ребра вводятся хорошо. А затем следует не строить каркас, а поддерживать его в процессе ввода ребер! Каким образом? Если новое ребро соединяет ранее соединенные вершины графа и вес его больше веса старого ребра, то оно просто игнорируется, а если меньше, то новое ребро заменяет старое. Если новое ребро приводит к образованию единственного цикла в графе (цикл ищется, например, обходом в глубину), то из цикла удаляется ребро с максимальным весом. Таким образом, на каждом шаге у нас есть каркас минимального веса, а его поддержание требует незначительных усилий.

Методические рекомендации

Данную задачу можно использовать как учебную при изучении алгоритмов на графах. При наличии тестов и проверяющей программы целесообразно организовать соревнование по оценке «коварства» жюри.

«Сравнение кодов»

Условие задачи

Компания Racine Business Networks (RBN) подала в суд на компанию Heuristic Algorithm Languages (HAL), утверждая,

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 148–150.

что HAL использовала исходный код из RBN UNIX™ и внесла его в открытый код операционной системы HALinux.

Как RBN, так и HAL используют язык программирования, в котором каждый оператор располагается на отдельной строке и имеет вид:

$$\text{STOREA} = \text{STOREB} + \text{STOREC}$$

(STOREA, STOREB и STOREC — имена переменных). Каждый оператор записывается следующим образом: имя первой переменной начинается с первой позиции строки, затем — пробел, знак равенства, пробел, имя второй переменной, пробел, знак сложения, пробел и имя третьей переменной. Одно и то же имя переменной может встречаться в строке более одного раза. Имена переменных имеют длину от 1 до 8 символов и состоят из заглавных латинских ASCII букв (•A•... •Z•).

Утверждается, что HAL скопировала последовательность строк программы прямо из исходного кода RBN с минимальными изменениями:

- чтобы скрыть свое преступление, HAL изменила имена некоторых переменных. Точнее, HAL взяла последовательность строк из программы RBN и для каждой переменной в ней изменила ее имя. При этом новое имя переменной может совпасть со старым. После переименования никакие две различные переменные не могут называться одинаково;
- HAL могла изменить в некоторых строках порядок имен переменных в правой части оператора присваивания. Например, оператор вида

$$\text{STOREA} = \text{STOREB} + \text{STOREC}$$

мог быть изменен так:

$$\text{STOREA} = \text{STOREC} + \text{STOREB}$$

- утверждается также, что HAL не изменила порядок, в котором строки кода следуют в исходном тексте программы.

Необходимо по заданным исходным кодам программ RBN и HAL найти самую длинную последовательность подряд идущих строк из программы HAL, которую можно получить из последовательности подряд идущих строк программы RBN с помощью указанных выше преобразований. Заметьте, что найденная последовательность в коде HAL и соответствующая ей последовательность в коде RBN не обязательно начинаются в строках с одинаковыми номерами.

Входные данные — code.in:

- первая строка входного файла содержит два разделенных пробелами целых числа r и h ($1 \leq r \leq 1000$; $1 \leq h \leq 1000$). r задает число строк в коде программы RBN, а h — число строк в коде программы HAL;
- последующие r строк содержат программу RBN;
- последующие h строк содержат программу HAL.

Пример входных данных:

```
4 3
RA = RB + RC
RC = D + RE
RF = RF + RJ
RE = RF + RF
HD = HE + HF
HM = HN + D
HN = HA + HB
```

Выходные данные — code.out. Выходной файл должен содержать одну строку с единственным целым числом — длиной самой длинной последовательности идущих подряд строк, которую HAL могла скопировать с изменениями у RBN.

Пример выходных данных: 2.

Строки с 1-й по 2-ю из программы RBN совпадают со строками со 2-й по 3-ю из программы HAL при условии, что в программе RBN выполнены следующие замены имен переменных: $RA \rightarrow HM$, $RB \rightarrow D$, $RC \rightarrow HN$, $D \rightarrow HA$, $RE \rightarrow HB$. Решения с тремя и более соответствующими строками не существует.

Ограничения:

Время выполнения	2 seconds of CPU
Память	64 MB

Система оценки. За каждый тест, на котором ваша программа выведет правильный ответ, вы получите полный балл. Неоптимальный ответ на тест не оценивается.

Обсуждение задачи

Здесь имеет место очень сложная формулировка задачи, и увидеть без соответствующей подготовки присущую ей простоту трудно.

Пусть у нас есть не две программы, как в формулировке задачи, а две строки ($s1$ и $s2$), и требуется найти в них общую часть (подстроку) наибольшей длины. Подстрока определяется

из подряд идущих символов. Мы не будем рассматривать специальных достаточно эффективных алгоритмов Нудельмана-Вунша* или Ханта-Зиманского**, а ограничимся самым простым решением. Начиная с позиции li строки $s1$ и позиции lj строки $s2$ (относительный сдвиг строк), осуществляем посимвольное сравнение (текущие номера символов: ri в строке $s1$ и rj — в $s2$). Если $s1[ri]$ равно $s2[rj]$, то увеличиваем на единицу значения ri и rj , а также изменяем количество совпадений, иначе увеличиваем на единицу значения li и lj . Последовательно перебирая пары (li, lj) : $(1, 1)$, $(1, 2)$, ..., $(1, \text{Length}(s2))$, а затем $(2, 1)$, $(3, 1)$, ..., $(\text{Length}(s1), 1)$, находим результат $\text{Length}(s)$ — количество символов в строке s .

Пример. Пусть даны две строки: $s1 = abxxxxc$ и $s2 = dxxxxe$. При $(li, lj) = (1, 1)$ результат равен трем, при $(li, lj) = (1, 2)$ — двум, при $(li, lj) = (2, 1)$ — четырем.

Приведем для наглядности изложения фрагмент логики, выполняющей сравнение строк с заданным относительным сдвигом:

```

Function Com(li,lj:Integer):Integer;
  Var ri,rj,res,t1,t2:Integer;
  Begin
    res:=0;
    ri:=0;
    rj:=0;
    t1:=Length(s1);
    t2:=Length(s2);
    While (ri<=t1) And (rj<=t2) And (li<=t1)
      And (lj<=t2) Do Begin
        ri:=Max(ri,li); {Max — функция вычисления
максимального из двух чисел}
        rj:=Max(rj,lj);
        If s1[ri]=s2[rj] Then Begin
          ri:=ri+1;
          rj:=rj+1;
          res:=Max(res,ri-li);
        End
        Else Begin li:=li+1;lj:=lj+1;End;
      End;
    Com:=res;
  End;

```

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2002. С. 100.

** Кипров С. Г. Алгоритм Ханта-Зиманского // Вестник Вятского государственного гуманитарного университета. Информатика. № 2. Киров, 2003. С. 186–189.

Пришло время сформулировать основной вопрос: можно ли преобразовать тексты программ так, чтобы логика поиска общей подстроки двух строк стала работоспособной и в нашей задаче? Конечно, она претерпит некие изменения, но суть — основная идея — останется неизменной.

Вчитаемся в формулировку задачи и попытаемся осознать, насколько важны, имеют принципиальное значение сами имена переменных. Какая нам разница — имя «Петя» или «Ваня»? Главное — в какой части оператора — левой или правой — оно встречается и было ли оно ранее в тексте программы. И если да (было), то опять же где — в левой или правой части оператора? Заметим, что если имя встречается в правой части, то, следуя формулировке задачи, не имеет значения, первым оно записано или вторым. Поэтому каждое имя мы будем характеризовать двухкомпонентным вектором (num_left, num_right) , где num_left — номер последнего предшествующего оператора, в левой части которого записано данное имя, а num_right — номер последнего предшествующего оператора, в правой части которого записано данное имя. Если имя встречается первый раз, то вектор равен $(0, 0)$.

Сформируем для первой программы матрицу $p[1..r, 1..3]$, элементами которой являются значения векторов, а для второй программы — $q[1..h, 1..3]$. Они, для примера из формулировки задачи, имеют вид:

$$p = \begin{pmatrix} (0,0) & (0,0) & (0,0) \\ (0,1) & (0,0) & (0,0) \\ (0,0) & (3,0) & (0,0) \\ (0,2) & (3,3) & (3,4) \end{pmatrix} \text{ и } q = \begin{pmatrix} (0,0) & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) \\ (0,2) & (0,0) & (0,0) \end{pmatrix}.$$

Мы формируем эти матрицы только по тексту самих программ (при построчном вводе каждого оператора, если говорить о технической реализации) — обращаться к тексту другой программы нет необходимости. А после формирования матриц и сами тексты программ нам больше не требуются! Вы еще не уловили смысл? Добавим в текст второй программы из формулировки задачи еще две строки. Он имеет вид:

```

HD = HE + HF
HM = HN + D
HN = HA + HB
A = A + B
HB = A + A

```

$$\text{Тогда матрица } q = \begin{pmatrix} (0,0) & (0,0) & (0,0) \\ (0,0) & (0,0) & (0,0) \\ (0,2) & (0,0) & (0,0) \\ (0,0) & (4,0) & (0,0) \\ (0,3) & (4,4) & (4,5) \end{pmatrix}.$$

Вас не удивляет некая похожесть матрицы p и нового варианта матрицы q (точнее, ее части со второй по пятую строки)? Ответом задачи в этом случае является значение «четыре» — «самая длинная последовательность подряд идущих строк из программы HAL, которую можно получить из последовательности подряд идущих строк программы RBN с помощью указанных преобразований равна четырем.

Методическое отступление. Момент «эврика» наступил, его надо закрепить! Школьнику предлагается написать произвольные тексты программ, составить для них матрицы p и q (саму программу при этом разрабатывать не следует — это отдельное задание, чисто техническое для нашего уровня обсуждения) и найти ответ задачи.

Вернемся к обсуждению. Пусть матрицы p и q сформированы. Как изменится функция Com для данного случая — когда нам необходимо осуществлять не посимвольное сравнение строк, а построковое сравнение матриц с учетом их относительного сдвига?

Во-первых, требуется функция сравнения двухкомпонентных векторов a и b с учетом их местоположения в матрицах p и q . Один из простых способов ее реализации имеет вид:

```
Function Eq(a,b:TVar):Boolean; {Тип данных -
TVar=Array [1..2] Of Integer;}
  Var i:Integer;
      pp:Boolean;
Begin
  pp:=True;
  For i:=1 To 2 Do
    pp:=pp And (((a[i]<l1) And (b[i]<lj))
      Or ((ri-a[i])=(rj-b[i])));
  Eq:=pp;
End;
```

Во-вторых, в функции Com требуется реализовать те сравнения, которые указаны в формулировке задачи. С этой целью введем две логические переменные $f1$ и $f2$. Значение $f1$ — это

результат сравнения левых частей операторов программ или $f1 := \text{Eq}(p[ri, 1], q[rj, 1])$, а значение $f2$ — правых: $f2 := (\text{Eq}(p[ri, 2], q[rj, 2]) \text{ And } \text{Eq}(p[ri, 3], q[rj, 3])) \text{ Or } (\text{Eq}(p[ri, 2], q[rj, 3]) \text{ And } \text{Eq}(p[ri, 3], q[rj, 2]))$. В результате сравнение символов $\text{If } s1[ri] = s2[rj]$ в тексте `Com` замещается проверкой условия истинности значений переменных $f1$ и $f2$ — `If f1 And f2`.

Методические указания

Эту задачу можно отнести к теме «Поиск данных». После изучения различных алгоритмов поиска вхождения подстрок в строку и поиска общих частей строк данная задача или творческое задание в виде рассмотренной задачи будет уместно и своевременно. В этом случае можно изменить обсуждение — не начинать с функции `Com`, а дать только основную идею сведения программы к числовой матрице.

Самостоятельные исследования

Данная задача допускает различные модификации. Если добавление знака умножения ($\ast$) не изменяет сути, то допустимость операции деления приводит к необходимости внесения корректив на уровне структур данных. Еще одно направление изменений связано с увеличением (уменьшением) количества операндов в правой части операторов программ. Например, предположим, что допускается оператор $A = 0$ — когда переменной с именем A присваивается нулевое значение. Тогда добавление в правых частях операндов $A + \dots + A$ не изменяет значения переменной, имя которой указано в левой части оператора. Решение задачи и для этого случая является хорошим заданием. Модификации задачи можно продолжить — тема неисчерпаема, и каждая из них достойна быть представленной на олимпиадах.

«Reverse»

Условие задачи

Рассмотрим вычислительную машину, выполняющую две операции (назовем ее «МДО»). Она имеет девять регистров, пронумерованных числами от 1 до 9. Каждый регистр хранит неотрицательное целое число в диапазоне от 0 до 1000 включительно. МДО способна выполнять две операции:

S i j	Записать значение регистра i , увеличенное на 1, в регистр j (i может быть равным j)
P i	Напечатать значение регистра i

Программа для МДО состоит из начальных значений каждого регистра и последовательности операций. Для заданного целого числа N ($0 \leq N \leq 255$) необходимо написать для МДО программу, которая печатает убывающую последовательность целых чисел $N, N-1, N-2, \dots, 0$. Максимальное количество подряд идущих S -операций должно быть как можно меньше.

Пример программы для МДО и ее исполнения при $N = 2$:

Операция	Новые значения регистров	Напечатанное значение
	1 2 3 4 5 6 7 8 9	
Начальные значения	0 2 0 0 0 0 0 0 0	
R 2	0 2 0 0 0 0 0 0 0	2
S 1 3	0 2 1 0 0 0 0 0 0	
R 3	0 2 1 0 0 0 0 0 0	1
R 1	0 2 1 0 0 0 0 0 0	0

Варианты входных данных, пронумерованные от 1 до 16, расположены на сервере соревнования.

Входные данные:

- первая строка входного файла содержит одно целое число k , обозначающее номер теста;
- вторая строка содержит число n .

Пример входных данных:

1
2

Выходные данные.

Первая строка выходного файла должна содержать текст «FILE reverse K », где k обозначает номер теста.

Вторая строка выходного файла должна содержать девять разделенных пробелами чисел, являющихся начальными значениями регистров и расположенных в следующем порядке: регистр 1, регистр 2, ..., регистр 9.

Остальные строки выходного файла должны содержать последовательность операций в порядке их исполнения, по одной операции в строке. Таким образом, третья строка должна содержать первую операцию, четвертая — вторую и так далее. Последняя строка должна содержать операцию, печатающую 0. В каждой строке должна содержаться только допустимая операция. Операции должны быть отформатированы, как указано в примере:

Пример выходного файла № 1 (неполный балл)	Пример выходного файла № 2 (полный балл)
FILE reverse 1 0 2 0 0 0 0 0 0 P 2 S 1 3 P 3 P 1	FILE reverse 1 0 2 1 0 0 0 0 0 P 2 P 3 P 1

Система оценки. Оценка каждого теста осуществляется с учетом корректности и оптимальности полученной программы для МДО.

Корректность: 20%.

Программа для МДО является правильной, если она выполняет не более 131 подряд идущей *S*-операции и последовательность напечатанных значений является правильной (содержит ровно $N+1$ целое число, начиная с числа N и заканчивая 0). Если какая-то *S*-операция вызывает переполнение регистра, программа для МДО считается неправильной.

Оптимальность: 80%.

Оптимальность корректной программы оценивается по максимальному количеству подряд идущих *S*-операций в программе, которое должно быть как можно меньше. Количество баллов будет определяться отличием вашей программы от наилучшей известной программы.

Обсуждение задачи*

Рассмотрим конкретные простые примеры с целью поиска метода решения этой сложной *NP*-полной задачи. Система оценки решения позволяет предположить, что точного решения нет (полный перебор вариантов не укладывается во временные рамки), и требуется нахождение ряда эвристик, дающих возможность получить приближенное решение, наиболее близкое к наилучшему варианту.

Очевидные соображения. Предположим, что мы умеем заполнять начальные регистры. Для определенности считаем, что они образуют убывающую последовательность. Число 0 должно быть выведено, и оно не может быть получено с помощью *S*-операций, поэтому для его хранения следует выделить последний регистр. Первый регистр предназначен для числа N , ибо нет смысла выполнять *S*-операции для его получения. Содержимое регистров не изменяется до тех пор, пока соответствующее значение не будет выведено программой для МДО.

*

Авторы благодарят Р. А. Веснина за помощь в разборе данной задачи.

Пример 1: значение N меньше, чем R ($N < R$) (будем говорить — задача (N, R)). Это самый простой случай, в качестве начальных значений регистров задается весь диапазон от N до 0, тогда количество подряд идущих S -операций (t) равно нулю.

Пусть $N = 3$ и $R = 5$. Начальные значения регистров: 3 2 1 0 0. Программа для МДО представлена в табл. 15.2:

Таблица 15.2

Команда	Результат
P 1	3
P 2	2
P 3	1
P 4	0

Для решения задачи требуется всего $N+1$ регистр. В данном примере один из регистров оказался неиспользуемым, его значение может быть произвольным.

Пример 2. Произвольное значение N и $R = 2$. Начальные значения регистров определяются однозначно: $(N, 0)$. Для получения из нуля числа $N-1$ требуется $t = N-1$ S -операций. Остальные числа требуют меньшее количество подряд идущих S -операций. Этот вариант программы — и есть оптимальное решение.

Пусть $N = 4$ и $R = 2$ ($t = N-1 = 3$). Начальные значения регистров: 3 0. Программа для МДО представлена в табл. 15.3.

Таблица 15.3

Команда	Регистры	Результат
P 1	4 0	4
S 2 1	1 0	
S 1 1	2 0	
S 1 1	3 0	
P 1	3 0	3
S 2 1	1 0	
S 1 1	2 0	
P 1	2 0	2
S 2 1	1 0	
P 1	1 0	1
P 2	1 0	0

Для диапазона чисел от N до 1, выводимых с первого регистра командой P 1, регистр с номером 2, хранящий 0, назовем *отсчетным* — все последовательности S-операций начинаются с команды S 2 1. Каждое очередное число для вывода начинается «формироваться с нуля».

Пример 3. Произвольное значение N и $R = 3$. Начальные значения регистров: $(N, x, 0)$. Экспериментальным путем убеждаемся, что:

- при $N = 3*m$ значение x равно $2*m-1$, или $2*m$, или $2*m+1$;
- при $N = 3*m+1$ значение x равно $2*m$ или $2*m+1$;
- при $N = 3*m+2$ значение x равно $2*m+1$.

При этом $t = m$.

Пусть $N = 7$ и $R = 3$ ($m = 2$, $t = m = 2$, $N = 3*m+1$, считаем, что $x = 2*m = 4$). Начальные значения регистров: 7 4 0. Программа для МДО представлена в табл. 15.4.

Таблица 15.4

Команда	Регистры	Результат
P 1	7 4 0	7
S 2 1	5 4 0	
S 1 1	6 4 0	
P 1	6 4 0	6
S 2 1	5 4 0	
P 1	5 4 0	5
S 3 1	1 4 0	
S 1 1	2 4 0	
P 2	2 4 0	4
S 1 2	2 3 0	
P 2	2 3 0	3
P 1	2 3 0	2
S 3 1	1 3 0	
P 1	1 3 0	1
P 3	1 3 0	0

В решении для этого примера выделяются следующие основные блоки:

- вывод диапазона чисел от 7 до 4+1 с использованием двух регистров — 1-го и 2-го (регистр 2 — отсчетный);
- вывод диапазона чисел от 4 до 2+1 с использованием двух регистров — 1-го и 2-го (регистр 1 — отсчетный);
- вывод диапазона чисел от 2 до 0+1 с использованием двух регистров — 1-го и 3-го (регистр 3 — отсчетный);
- вывод нуля с регистра 3.

Вывод. Итак, задача для трех регистров была сведена к трем подзадачам для двух регистров (!). Перед решением второй подзадачи решается вспомогательная задача подготовки значений регистров (выделена в табл. 15.4 жирным шрифтом) — с использованием регистра 3 как отсчетного в регистре 1 формируется значение 2. После этого регистр 1 используется как отсчетный при решении подзадачи.

Пример 4. Значения N и R удовлетворяют неравенству $N < \sum_{i=1}^R i$. Экспериментальным путем устанавливаем факт: начальное значение i -го регистра определяется по формуле $a_i = \max\left(N + 1 - \sum_{j=1}^i j, 0\right)$, при этом $t = 1$.

Утверждение. Решение является оптимальным, если в программе нет S -операции вида $S \ i \ i$. Это связано с тем, что после команды $P \ i$ значение в регистре i становится ненужным и на его место можно записать новое значение из другого регистра, увеличенное на единицу. Данное условие является достаточным, но не необходимым. В связи с этим может существовать более чем одно оптимальное решение.

Пусть $N = 9$ и $R = 4 \left(9 < \sum_{i=1}^4 i = 10 \right)$. Начальные значения регистров: 9 7 4 0. В табл. 15.5 представлены две программы для МДО — два возможных решения задачи. Первая — оптимальный вариант, вторая — с командой $S \ i \ i$ (выделена жирным шрифтом), которая не влияет в данном случае на оптимальность решения.

Таблица 15.5

1-я программа для МДО с соблюдением утверждения			2-я программа для МДО без соблюдения утверждения		
Команда	Регистры	Результат	Команда	Регистры	Результат
P 1	9 7 4 0	9	P 1	9 7 4 0	9
S 2 1	8 7 4 0		S 2 1	8 7 4 0	
P 1	8 7 4 0	8	P 1	8 7 4 0	8
S 3 1	5 7 4 0		S 3 1	5 7 4 0	
P 2	5 7 4 0	7	P 2	5 7 4 0	7
S 2 1	5 6 4 0		S 2 1	5 6 4 0	
P 2	5 6 4 0	6	P 2	5 6 4 0	6
S 4 2	5 1 4 0		S 4 2	5 1 4 0	
P 1	5 1 4 0	5	P 1	5 1 4 0	5
S 2 1	2 1 4 0		S 2 2	5 2 4 0	
P 3	2 1 4 0	4	P 3	5 2 4 0	4
S 1 3	2 1 3 0		S 2 3	5 2 3 0	
P 3	2 1 3 0	3	P 3	5 2 3 0	3
P 1	2 1 3 0	2	S 4 3	5 2 1 0	
P 2	2 1 3 0	1	P 2	5 2 1 0	2
P 4	2 1 3 0	0	P 3	5 2 1 0	1
			P 4	5 2 1 0	0

Пусть $N = 14$ и $R = 5 \left(14 < \sum_{i=1}^5 i = 15 \right)$. При начальных значениях регистров ($14 = 9+5$, $12 = 7+5$, $9 = 4+5$, $5 = 0+5$, $0 = (14 \ 12 \ 9 \ 5 \ 0)$) первая программа (табл. 15.5) выведет диапазон чисел от 14 до 5. Ее небольшая модификация в конце программы обеспечивает вывод всего диапазона от 14 до 0. При этом полное решение является оптимальным ($t = 1$).

Модификация программ из табл. 15.5 для $N = 14$ и $R = 5$ приведена в табл. 15.6. Для первой (левой) программы жирным шрифтом выделены модифицированные команды. Описание второй программы (табл. 15.6) прервано, ибо дальнейший вывод убывающей последовательности чисел с соблюдением условия $t = 1$ невозможен.

Таблица 15.6

1-я программа для МДО с соблюдением утверждения			2-я программа для МДО без соблюдения утверждения		
Команда	Регистры	Результат	Команда	Регистры	Результат
P 1	14 12 9 5 0	14	P 1	14 12 9 5 0	14
S 2 1	13 12 9 5 0		S 2 1	13 12 9 5 0	
P 1	13 12 9 5 0	13	P 1	13 12 9 5 0	13
S 3 1	10 12 9 5 0		S 3 1	10 12 9 5 0	
P 2	10 12 9 5 0	12	P 2	10 12 9 5 0	12
S 2 1	10 11 9 5 0		S 2 1	10 11 9 5 0	
P 2	10 11 9 5 0	11	P 2	10 11 9 5 0	11
S 4 2	10 6 9 5 0		S 4 2	10 6 9 5 0	
P 1	10 6 9 5 0	10	P 1	10 6 9 5 0	10
S 2 1	7 6 9 5 0		S 2 2	10 7 9 5 0	
P 3	7 6 9 5 0	9	P 3	10 7 9 5 0	9
S 1 3	7 6 8 5 0		S 2 3	10 7 8 5 0	
P 3	7 6 8 5 0	8	P 3	10 7 8 5 0	8
S 5 3	7 6 1 5 0		S 4 3	10 7 6 5 0	
P 1	7 6 1 5 0	7	P 2	10 7 6 5 0	7
S 3 1	2 6 1 5 0		S 5 2	10 1 6 5 0	
P 2	2 6 1 5 0	6	P 3	10 1 6 5 0	6
S 1 2	2 3 1 5 0		S 2 3	10 1 2 5 0	
P 4	2 3 1 5 0	5	P 4	10 1 2 5 0	5
S 2 4	2 3 1 4 0		S 4 4	10 1 2 3 0	
P 4	2 3 1 4 0	4			
P 2	2 3 1 4 0	3			
P 1	2 3 1 4 0	2			
P 3	2 3 1 4 0	1			
P 5	2 3 1 4 0	0			

Если даже предположить, что вторая программа (табл. 15.5) также дает оптимальное решение при $N = 9$ и $R = 4$, то ее модификация, как для первой программы, невозможна для $N = 14$ и $R = 5$. Операция $S i i$ стирает из регистра i предыдущее значение, и его придется получать повторно, при этом есть возможность использовать свободный регистр для записи нового значения.

Вывод. В примере 4 показано, что решение задачи при $N = 14$ и $R = 5$ разбивается на две подзадачи:

- решается задача при $N = 9$ и $R = 4$; при этом выводится диапазон чисел от 14 до 6 с использованием отсчетного регистра с номером 4 со значением 5 (вместо нуля);
- решается задача при $N = 5$ и $R = 5$ с использованием отсчетного регистра 5 со значением 0.

Методическое отступление. Самостоятельное составление программ для МДО и их обсуждение уместно перед рассмотрением общей логики решения задачи.

Решение полным перебором вариантов заключается в генерации всех возможных наборов начальных значений регистров, образующих убывающие последовательности чисел. Количество таких наборов равно C_{N-1}^{R-2} . Для каждого набора из всех возможных последовательностей операций, приводящих к выводу убывающей последовательности от N до 0, следует выбрать оптимальный вариант, содержащий наименьшее количество идущих подряд S -операций.

Оценим количество различных последовательностей операций для произвольного набора начальных значений регистров. Пусть количество подряд записанных S -операций равно t . Максимальное значение, которое может принимать t , равно $N-1$. Программа для МДО должна содержать $N+1$ P -операцию, и между любыми двумя соседними P -операциями может располагаться 0 ... $N-1$ S -операций. Таким образом, количество последовательностей (программ), состоящих из P и S -операций, равно N^N . При этом большая часть количества всех последовательностей операций не дает корректного решения, и их следует «отсекать» каким-либо способом. Полученная оценка $O(C_{N-1}^{R-2} \cdot N^N) = O(N^N)$

достаточно приближительна, но четко дает понять, что простым перебором эта задача не решается.

Одним из приемов поиска приближенного решения NP -полных задач является введение дополнительного параметра t . В данном случае исходная задача (N, R) переходит в задачу (N, R, t) . Для новой задачи выясняется — разрешима она или нет при заданных параметрах. Найдя минимальное значение параметра t , при неизменных значениях N и R , при котором новая задача разрешима, мы одновременно получаем и решение исходной задачи.

Задача (N, R, t) имеет на одну известную величину больше и требует лишь проверки разрешимости. Как же проверить, что данная задача имеет решение? Ответ — на двух регистрах при заданном значении t известно, что задача решается для

$N \leq t+1$. Далее работает следующая схема: если количество задействованных регистров меньше R , то добавляем еще один регистр и ищем максимальное число N' , для которого разрешима задача (при этом используются найденные ранее решения для меньших значений N' и количества регистров); в противном случае проверяем: если $N' \geq N$, то задача разрешима.

Описываемый метод позволяет использовать рекурсивную схему решения задачи. Предположим, что при заданном значении t определено, что на r регистрах задача имеет решение (и известно, как его получить) для всех чисел из диапазона от 0 до n . Увеличиваем количество регистров на единицу. Естественно, что по-прежнему существует решение для диапазона от 0 до n , так как новый регистр можно не задействовать, а использовать уже существующие решения для r регистров. Этот вариант мало интересен, ибо хранить одинаковые решения нет смысла.

Пусть требуется проверить, существует ли решение $(n, r+1, t)$. Теоретически в регистр r может быть записано любое число от 0 до $\max(0, n-r+1)$ так, чтобы значения во всех регистрах образовывали убывающую последовательность. Пусть в регистр r записано некоторое значение b . Программа для МДО не должна изменять это значение до тех пор, пока не будут выведены все числа из интервала от n до $b+1$ ($k = n-b$ значений). Если разрешимы задачи (k, r, t) (отсчетный регистр r содержит значение b вместо 0) и $(b, r+1, t)$, то вместе они выводят числа из интервала от n до 0. Остается в программу для МДО задачи (k, r, t) внести дополнение, которое к завершению этой программы в первые $r-1$ регистров запишет начальные значения для задачи $(b, r+1, t)$ (данный прием рассмотрен в примере). Если это возможно, то обе программы для указанных задач становятся «единым целым» — задача $(n, r+1, t)$ разрешима. В противном случае в регистр r следует записать другое значение b и повторить проверку.

С помощью данного метода для произвольной задачи (N, R, t) находится единственный набор начальных значений регистров и единственная последовательность операций. «Отсечение» вариантов перебора значительно, поэтому вероятно, что и оптимальное решение окажется нерассмотренным. Следовательно, метод является приближенным.

Оценим временную сложность решения. При использовании бинарного поиска в интервале от 0 до $n-1$ для определения значения параметра t требуется $O(\log N)$ операций. При заданном значении t требуется проверить разрешимость всех задач (n, r, t) ($1 \leq n \leq N$, $2 \leq r \leq R$), их количество — $O(N \cdot R)$. Для проверки одной задачи требуется подобрать значение b — количество

операций $O(N)$ — и проверить осуществимость вывода последовательности чисел — количество операций $O(N^2)$ (столько операций в программе для МДО). Объединяя все результаты, получаем оценку $O(N^4 R \log(N))$.

Проверка разрешимости задачи и вывод оптимального решения выполняются с помощью одного алгоритма (в первом случае — без отображения данных). Алгоритмически задача (n, r, t) разбивается на две подзадачи $(k, r-1, t)$ и (b, r, t) . Введем обозначения: b — база числа n , $k = n - b$ — остаток числа n . Затем каждая из подзадач также разбивается на более мелкие подзадачи до тех пор, пока количество регистров не станет равно двум. В каждой очередной первой подзадаче количество регистров уменьшается на единицу, а во второй — количество регистров уменьшается, если b станет меньше r (для ее решения достаточно b регистров), — а это обязательно произойдет на каком-то из шагов. Как только получена подзадача с двумя регистрами (один из них — отсчетный), ее решение известно (рассмотрено в примере).

После изложения основной идеи метода решения задачи остается только одна еще не рассмотренная проблема: как сделать так, чтобы алгоритм мог находить решение, наиболее близкое к оптимальному? Для устранения этой неопределенности требуется ввести дополнительный критерий, позволяющий определять, какое из разбиений на подзадачи (по параметру b) наиболее целесообразно для задачи (N, R, t) при ее использовании как подзадачи для задачи большей размерности.

Методическое отступление. Приведенные здесь рассуждения достаточно трудно воспринимаются школьником. Вероятно, методически более грамотно будет увеличить количество рассматриваемых примеров и из них «вывести» (резюмируя) основные идеи метода.

Перейдем теперь к особенностям реализации метода. Во-первых, рассмотрим вопрос о формировании начальных значений регистров МДО. Введем двумерный массив $A[1..Nmax, 2..Rmax]$. У элемента $A[i, j]$ индекс i — число, для которого решается задача ($1 \leq i \leq n$), j — количество регистров ($2 \leq j \leq r$). Для него хранятся параметры b, r, c :

- b — база числа i (остаток k числа i можно не хранить);
- r — количество регистров, необходимых для решения задачи с количеством b при использовании не более t операций;
- c — количество S -операций, которые не были использованы при выводе («резервные»).

Смысл параметра s поясним с помощью примера.

Пример 5. Требуется проверить разрешимость задачи $(N, R, t) = (7, 4, 2)$. Текст программы с пояснениями приведен в табл. 15.7.

Таблица 15.7

№	Команда	Регистры	Результат	№	Команда	Регистры	Результат
1	P 1	7 4 0	7	13	P 2	2 3 0	3
3	S 1 1	6 4 0		15	SC		
4	P 1	6 4 0	6	16	P 1	2 3 0	2
5	S 2 1	5 4 0		17	S 3 1	1 3 0	
6	SL			18	SC		
7	P 1	5 4 0	5	19	P 1	1 3 0	1
8	S 3 1	1 4 0		20	SC		
9	S 1 1	2 4 0		21	SC		
10	P 2	2 4 0	4	22	P 3	1 3 0	0
11	S 1 2	2 3 0		23	SC		
12	SL			24	SC		

Пояснение. SL — S-операция, которая не может быть выполнена, так как отсутствуют свободные регистры для записи новых значений. В данном примере таких операций две. SC — S-операция, которая сейчас не выполняется, но может быть использована, когда данная задача будет выполняться как подзадача. В данном примере таких операций семь.

В данном примере $t = 2$; следовательно, после каждой команды P i может быть выполнено две S-операции. В табл. 15.7 приведены два типа S-операций — SL и SC. Количество SC-операций (s) подсчитывается и является критерием выбора разбиения — выбирается то разбиение, при котором s принимает максимальное значение (дополнительный критерий).

Задание параметров b и r позволяет однозначно разбить задачу (N, R) на две подзадачи: $(N-b, R-1)$ и (b, r) . В свою очередь, каждая подзадача также имеет однозначное разбиение.

Таким образом, алгоритм находит единственное приближенное решение задачи (N, R) .

Если используется не более t операций, то автоматически формируется следующая часть массива A :

- при $j = 2$ — элементы $A[i, 2]$ ($i = 1 \dots t+1$), для них $b = i, r = 1, c = t$;
- при $i = 1$ — элементы $A[1, j]$ ($j = 2 \dots r$), для них $b = 1, r = 1, c = (j-1) \cdot t$.

Основываясь на этих данных, элементы $A[i, j]$ ($i = 2 \dots N, j = 3 \dots R$) определяются путем поиска параметров, удовлетворяющих решению задачи (i, j, t) . Для проверки разрешимости задачи (i, j) требуется проверить разрешимость каждой пары задач (b, r, t) и $(k, j-1, t)$ при $b = 1 \dots i-1, k = i-b, r = 2 \dots j$. При положительном результате в обоих случаях имеет смысл проверить разрешимость задачи (i, j, t) , т. е. применить к ней найденное разбиение и убедиться в корректности вывода. Если в процессе вывода действительно получилась убывающая последовательность целых чисел $i, i-1, i-2, \dots, 0$, то найдено решение задачи (i, j, t) .

Существует условие, при невыполнении которого задача (i, j, t) при заданном разбиении будет неразрешима: $b-t-1 = A[k, j-1] \cdot c$. Число $b-t-1$ представляет собой максимальную величину из всех неизвестных начальных значений регистров при решении задачи (b, r, t) . Число $A[k, j-1] \cdot c$ — это количество резервных S -операций, которые могут быть использованы при решении задачи $(k, j-1, t)$. Данное условие является лишь необходимым требованием, но позволяет отсеять большую часть решений, заведомо приводящих к неоптимальному результату.

Пример 6. Формирование элемента $A[4, 4]$ массива A при $t = 1$ (табл. 15.8). Пусть элементы первых трех строк A уже определены.

Таблица 15.8

$i \backslash j$	2			3			4		
	b	r	c	b	r	c	b	r	c
1	1	1	1	1	1	1	1	1	1
2	2	1	1	1	1	2	1	1	2
3	—	—	—	1	1	2	1	1	3
4	—	—	—	2	1	2	?	?	?

Возможные варианты разбиения задачи на подзадачи приведены в табл. 15.9.

Таблица 15.9

Разбиение, ($k \ j-1$) ($b \ r$)	Задача разре- шима	Значе- ние s	Разбиение, ($k \ j-1$) ($b \ r$)	Задача разре- шима	Значе- ние s
(3 3) (1 2)	да	3	(2 3) (2 4)	да	3
(3 3) (1 3)	да	3	(1 3) (3 2)	нет	—
(3 3) (1 4)	да	3	(1 3) (3 3)	да	2
(2 3) (2 2)	да	3	(1 3) (3 4)	да	2
(2 3) (2 3)	да	3			

При первых шести разбиениях параметр s принимает максимальное значение. Из этих вариантов оставим те, где b максимально — чем больше «нагрузка» сейчас, тем меньше «нагрузка» будет потом. Среди оставшихся выбираем разбиение с минимальным значением r — чем меньше требуется «ресурсов» останется для выполнения данной задачи, тем больше свободных «ресурсов» останется для выполнения последующих задач. Заметим, что приведенный способ выбора разбиения не является единственно возможным. Можно рассматривать различные стратегии выбора разбиения на подзадачи, руководствуясь другими признаками и критериями.

После формирования A становится возможным указать начальные значения регистров. Для хранения значений регистров введем массив $Reg[1..Rmax]$. Ранее было оговорено, что значения в регистрах мы будем располагать в убывающем порядке. В задаче (N, R) два значения известны: регистр $Reg[1]$ содержит значение N , регистр $Reg[R]$ содержит 0. В данной задаче регистр с номером R является отсчетным, так как он хранит минимальное значение, которое должно быть выведено в процессе решения задачи. Элемент массива $A[N, R]$ содержит значение b . Разбиваем задачу (N, R) на подзадачи $(N - A[N, R] \cdot b, R-1)$ и $(A[N, R] \cdot b, A[N, R] \cdot r)$. Первая подзадача использует на один регистр меньше, т. е. первые $R-1$ регистров, последний из которых является отсчетным и содержит значение $Reg[R] + A[N, R] \cdot b$. Аналогичным образом продолжаем заполнять регистры для задачи $(N - A[N, R] \cdot b, R-1)$. Обозначим $k = N - A[N, R] \cdot b$ и $r = R-1$. Задача (k, r) разбивается на подзадачи: $(k - A[k, r] \cdot b, r-1)$ и $(A[k, r] \cdot b, r)$. Видим, что первая подзадача решается на $r-1=R-2$ регистрах, последний из которых ($Reg[r-1]$) должен содержать значение $Reg[r] + A[k, r] \cdot b$. Тем самым определено

значение еще одного регистра. Процесс продолжается до тех пор, пока не будут определены значения всех регистров.

Последнее, что еще не рассмотрено, — это процесс вывода команд, осуществляющих вывод убывающей последовательности. Для решения этой задачи все исходные данные уже подготовлены:

- известны начальные значения регистров;
- известно разбиение любой задачи на подзадачи, решение которой может потребоваться;
- известен способ решения задачи на двух регистрах.

Каждая подзадача, получаемая при разбиении, характеризуется своим *состоянием*: набором регистров, которые ей выделены, и их значениями. Понятно, что при разбиении задачи на каждой подзадаче будут выделены все регистры сразу. Из алгоритма получения начальных значений регистров следует, что для всякой первой подзадачи при разбиении задачи (n, r) известны значения всех регистров, а для второй подзадачи — значения только первого регистра с номером $r-1$ и последнего с номером r ($Reg[r] = \text{«отсчетное значение»}$, $Reg[r-1] = Reg[r] + b$).

Состояние задачи (n, r) описывается следующей структурой: $(n, r, rc, nums)$, где:

- rc — текущее количество регистров, выделенных подзадаче (до начала решения подзадачи должно выполняться равенство: $rc = r$);
- $nums$ — массив, хранящий номера индексов регистров задачи (n, r) в порядке убывания значений, записанных в регистры.

Итак, суть решения:

Шаг 1. Заполнить регистры $Regs$ начальными значениями по данным из массива A .

Шаг 2. Поместить в стек $(N, R, R, (1, 2, \dots, R))$.

Шаг 3. Извлечь из стека $(n, r, rc, nums)$; если $rc < r$, то задача не может быть решена.

Шаг 4. Если $r = 2$, то решить задачу на двух регистрах — $(n, 2)$.

Шаг 5. Иначе — разбить задачу на 2 подзадачи.

Шаг 6. Если стек не пуст, то перейти к пункту 3, иначе — к пункту 7.

Шаг 7. Вывести 0 из регистра $Regs[R]$.

Решение задачи $(n, 2)$ сводится к следующим действиям:

- а) выполнить $P(nums[1])$;
- б) получить последовательность от $Regs[nums[2]] + n-1$ до $Regs[nums[2]] + 1$ — в цикле для i от $n-1$ до 1:

- выполнить одну S -операцию ($nums[2]$) ($nums[1]$),
- выполнить $i-1$ раз S -операцию ($nums[1]$) ($nums[1]$),
- выполнить $t-i$ раз S -операцию над регистрами, где это возможно,
- выполнить P ($nums[1]$);

в) освободить регистр с номером $nums[1]$:

- начиная с вершины стека, найти подзадачу, для которой $rc < r$,
- освобождаемому регистру присвоить индекс регистра, в котором будет храниться минимальное из еще не полученных начальных значений для подзадачи,
- увеличить значение rc на 1;

г) выполнить t раз S -операцию над регистрами там, где это возможно.

Логика разбиения задачи на подзадачи:

- а) объявить массив индексов регистров $nums2$ для второй подзадачи и указать в нем индексы первого и последнего регистров: $nums2[1] = nums2[r-1]$, $nums2[A[n, r].r] = nums[r]$, а затем поместить в стек вторую подзадачу ($A[n, r] \cdot b$, $A[n, r] \cdot r$, 2, $nums2$);
- б) поместить в стек первую подзадачу ($n - A[n, r] \cdot b$, $r-1$, $r-1$, $nums$).

Таким образом, первоначально решается первая подзадача, а затем вторая.

В описании алгоритма присутствуют слова: «выполнить S -операцию над регистрами там, где это возможно». Их смысл сложен в следующем. Начиная с вершины стека находится подзадача, для которой регистр с последним присвоенным индексом, равным $last$, хранит значение $Regs[nums[last]]$, отличающееся от положенного начального значения. Если значение $Regs[nums[last]]$ меньше требуемой величины, то должна быть выполнена S -операция ($nums[last]$) ($nums[last]$), иначе — S -операция ($nums[last-1]$) ($nums[last]$) — это случай, когда регистр был освобожден другой подзадачей, выводящей диапазон из больших чисел, и после этого его значение не изменялось. Именно эти S -операции используются для подготовки начальных значений регистров подзадач, находящихся в стеке и ожидающих выполнения.

Пример 7. Пусть $N = 15$, $R = 4$, $t = 2$. Сформированный массив A представлен в табл. 15.10.

Таблица 15.10

N	2			3			4		
	b	r	c	b	r	c	b	r	c
1	1	2	2	1	2	2	1	2	2
2	2	2	2	1	2	4	1	2	4
3	3	2	2	1	2	5	1	2	6
4				1	2	5	2	3	7
5				2	2	5	3	3	8
6				3	2	5	3	3	9
7				4	3	5	4	3	9
8				5	3	5	3	3	10
9							3	3	10
10							3	3	11

Определим начальные значения регистров:

$Regs[1] = N = 10$;

$Regs[4] = 0$;

$Regs[1] = Regs[0] + A[10, 4] \cdot b = 3$;

$Regs[2] = Regs[1] + A[10-3, 3] \cdot b = 7$.

Итого: $Regs = (10, 7, 3, 0)$.

Работа алгоритма представлена в табл. 15.11. В первом столбце цифрой указан номер шага, а буквой — действия на этом шаге.

Таблица 15.11

Шаг	Регистры	Стек (n, r, rc, (литы))	Извлечено из стека	Команда	Результат
2	(10 7 3 0)	(10 4 4 (1 2 3 4))			
3	(10 7 3 0)		(10 4 4 (1 2 3 4))		
5 а)	(10 7 3 0)	(3 3 2 (3 0 4))	(10 4 4 (1 2 3 4))		
5 б)	(10 7 3 0)	(3 3 2 (3 0 4)) (7 3 3 (1 2 3))	(10 4 4 (1 2 3 4))		

Шаг	Регистры	Стек (п, г, гс, (лима))	Извлечено из стека	Команда	Результат
3	(10 7 3 0)	(3 3 2 (3 0 4)) (3 3 2 (3 0 4))	(7 3 3 (1 2 3))		
5 а)	(10 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(7 3 3 (1 2 3))		
5 б)	(10 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3)) (3 2 2 (1 2))	(7 3 3 (1 2 3))		
3	(10 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))		
4 а)	(10 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))	P 1	10
4 б)	(8 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))	S 2 1	
	(9 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))	S 1 1	
	(9 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))	P 1	9
	(8 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))	S 2 1	
	(8 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 0 3))	(3 2 2 (1 2))	P 1	8
4 в)	(8 7 3 0)	(3 3 2 (3 0 4)) (4 3 2 (2 1 3))	(3 2 2 (1 2))		
4 г)	(4 7 3 0)	(3 3 2 (3 0 4)) (4 3 3 (2 1 3))	(3 2 2 (1 2))	S 3 1	
3	(4 7 3 0)	(3 3 2 (3 0 4))	(4 3 3 (2 1 3))		
5 а)	(4 7 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(4 3 3 (2 1 3))		
5 б)	(4 7 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3)) (3 2 2 (2 1))	(4 3 3 (2 1 3))		
3	(4 7 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))		
4 а)	(4 7 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	P 2	7
4 б)	(4 5 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	S 1 2	
	(4 6 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	S 2 2	

Шаг	Регистры	Стек (n, r, rc, (lits))	Извлечено из стека	Команда	Результат
	(4 6 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	P 2	6
	(4 5 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	S 1 2	
	(4 5 3 0)	(3 3 2 (3 0 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	P 2	5
4 в)	(4 5 3 0)	(3 3 3 (3 2 4)) (1 2 2 (1 3))	(3 2 2 (2 1))		
4 г)	(4 1 3 0)	(3 3 3 (3 2 4)) (1 2 2 (1 3))	(3 2 2 (2 1))	S 4 2	
3	(4 1 3 0)	(3 3 3 (3 2 4))	(1 2 2 (1 3))		
4 а)	(4 1 3 0)	(3 3 3 (3 2 4))	(1 2 2 (1 3))	P 1	4
4 б, в, г)					
3	(4 1 3 0)		(3 3 3 (3 2 4))		
5 а)	(4 1 3 0)	(1 2 2 (2 4))	(3 3 3 (3 2 4))		
5 б)	(4 1 3 0)	(1 2 2 (2 4)) (2 2 2 (3 2))	(3 3 3 (3 2 4))		
3	(4 1 3 0)	(1 2 2 (2 4))	(2 2 2 (3 2))		
4 а)	(4 1 3 0)	(1 2 2 (2 4))	(2 2 2 (3 2))	P 3	3
4 б)	(4 1 2 0)	(1 2 2 (2 4))	(2 2 2 (3 2))	S 2 3	
	(4 1 2 0)	(1 2 2 (2 4))	(2 2 2 (3 2))	P 3	2
4 б, в, г)					
3	(4 1 2 0)		(1 2 2 (2 4))		
4 а)	(4 1 2 0)		(1 2 2 (2 4))	P 2	1
4 б, в, г)					
7	(4 1 2 0)		(1 2 2 (2 4))	P 4	0

Методические указания

Эта задача может использоваться в качестве метода изучения приближенных решений NP-полных задач, а как творческое задание она поистине неисчерпаема!

Самостоятельные исследования

Рассмотренную задачу трудно назвать «задачей» в общепринятой олимпиадной трактовке. Это скорее *проблема* для серьезного исследования, и нами затронута только «верхняя часть айсберга». Решение — приближенное (следует найти контрпри-

мер, для которого оно не даст наилучшего ответа). Математическая составляющая проблемы не затронута в обсуждении. А дальнейший поиск эвристик, улучшающих предложенный вариант решения, может быть предметом гордости как школьника, так и многих специалистов по информатике.

ЗАДАЧИ ВТОРОГО ТУРА

«Угадайте корову» (интерактивная задача)

Условие задачи

В стаде фермера Джона находятся n пронумерованных от 1 до n ($1 \leq n \leq 50$) и похожих коров. Когда фермер Джон ставит корову в стойло, он должен знать, какую корову он туда ставит.

Коровы различаются по p признакам, пронумерованным от 1 до p ($1 \leq p \leq 8$), каждый из которых имеет по три возможных значения. Например, цвет метки на ухе коровы может быть желтым, зеленым или красным. Значения каждого признака определяются одной из букв «X», «Y» или «Z». Любая пара коров фермера Джона отличается по крайней мере одним признаком.

Напишите программу, которая по заданным признакам коров поможет фермеру Джону определить номер коровы, которую он ставит в стойло. Ваша программа может задать Джону не более 100 вопросов вида: «Принадлежит ли значение некоторого признака T у коровы некоторому множеству значений $S?$ ».

Задайте как можно меньше вопросов для определения номера коровы.

Входной файл признаков — guess.in:

- первая строка входного файла содержит два целых числа n и p , разделенных пробелом;
- каждая из последующих n строк описывает признаки коровы, используя p букв, разделенных пробелами. Первая буква каждой строки — значение признака 1 и так далее. Вторая строка во входном файле описывает корову с номером 1, третья строка — корову с номером 2 и так далее.

Пример входного файла признаков:

```
4 2
X Z
X Y
Y X
Y Y
```


Интерактивность: стандартный ввод и стандартный вывод. Шаг «вопрос/ответ» осуществляется через стандартный ввод и стандартный вывод. Ваша программа задает вопрос о корове посредством записи в стандартный вывод строки следующего вида: буква «Q», номер признака, значения признака (одно или более), разделенные пробелами. Например, строка Q 1 Z Y соответствует вопросу: «Имеет ли первый признак коровы значение «Z» или «Y»?». Признак может быть целым числом в пределах от 1 до p. Значения признака должны быть только «X», «Y» или «Z» и не должны повторяться в одной строке.

После задания вопроса ваша программа должна читать одну строку, содержащую одно из целых чисел — 0 или 1. Число 1 обозначает, что корова обладает одним из указанных значений признака. Число 0 означает, что ни одним из указанных значений признака корова не обладает.

Последняя строка вывода программы должна содержать букву «C», пробел и одно число (номер коровы).

Пример диалога (для вышеуказанного примера):

Ввод	Вывод	Комментарий
	Q 1 X Z	
0		Может быть, корова 3 или корова 4
	Q 2 Y	
1		Должна быть корова 4!
	C 4	
Завершение работы программы		

Ограничения:

Время выполнения	1 секунда CPU
Память	64 MB

Система оценки.

Корректность: 30% от баллов. Программа, которая выводит правильный номер коровы, получит полный балл за корректность, если существует единственная корова, соответствующая полученным ответам. Программа, которая задает больше 100 вопросов, баллов за тест не получит.

Зависимость от количества вопросов: 70% от баллов. Оставшиеся баллы будут определены в зависимости от количества вопросов, заданных для корректного определения коровы.

Система оценки устроена так, чтобы оценить минимальное количество вопросов, которые задает ваша программа в худшем случае. Близкие к оптимальному решения получают частичные баллы.

Обсуждение задачи

Самый простой вариант решения — последовательно задавать вопросы типа «равен ли признак с номером i значению X ». Если ответ — «да», то признак определен, иначе следует вопрос — «равен ли признак с номером i значению Y ». Очевидно как то, что при этом потребуются в худшем случае $2p$ вопросов, так и то, что это решение не наберет полный балл. Ведь по условию задачи требуется задавать как можно меньше вопросов для определения номера коровы!

Второй очевидный факт заключается в том, что вопрос типа $Q\ 1\ Z\ Y$ равносильен вопросу $Q\ 1\ X$. Действительно, ответ «нет» на первый вопрос означает равенство признака значению X , ответ «да» — равенство Z или Y . Для второго вопроса имеем: ответ «да» — признак равен X , ответ «нет» — признак равен Z или Y . Таким образом, можно ограничиться вопросами второго типа.

Мы не тронемся с места до тех пор, пока не определим некое пространство решения задачи и логику перехода из одного состояния в другое в этом пространстве. Первым шагом в этой работе является кодировка признаков и то, как вопросы и ответы на них отражаются в изменении кодировки.

Признак имеет три значения. Следовательно, трех битов достаточно для их представления в памяти. Считаем, что «0» (в бите записан нуль) является допустимым для значения признака, а «1» говорит о невозможности равенства признака этому значению. Тогда последовательность «000» означает, что признак может принимать значения X , Y и Z , а «001» — только Y и Z . Последовательность из $3 \cdot p$ битов служит для описания всех признаков и определяет конкретное состояние в пространстве решения задачи. Если задано t признаков, то последовательность имеет вид $P_1 P_{t-1} \dots P_1$; другими словами, первый признак описывается тремя младшими битами последовательности, затем второй признак и т. д. В этом случае конструкция:

```
For i:=0 To p-1 Do
  For j:=0 To 2 Do ...
```

обеспечивает побитовый просмотр последовательности (начиная с младших разрядов).

Обратимся к примеру в формулировке задачи. Описание конечных состояний для всех коров (первая строка таблицы), следующих из формулировки задачи, представлено в табл. 15.12.

Таблица 15.12

1						2						3						4					
Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X
0	1	1	1	1	0	1	0	1	1	1	0	1	1	0	1	0	1	1	0	1	1	0	1
P_2			P_1			P_2			P_1			P_2			P_1			P_2			P_1		

Начальным состоянием является последовательность «000000». Пусть задан вопрос: $Q \ 1 \ X$ — равен ли первый признак коровы значению X ? Если ответ — «нет», то из начального состояния осуществляется переход в состояние «000001», а если «да», то в «000110». Другими словами, при ответе «нет» это значение признака запрещается, а остальные — не изменяются; ответ «да» запрещает все другие значения признаков.

Формализованная запись логики перехода из одного состояния в другое имеет вид:

Function ChangeY(s,i,j:Integer):Integer; {s — номер состояния, i — номер признака, j — номер значения признака}

Var t,c:Integer;

Begin

c:=s;

For t:=0 **To** 2 **Do** {Добавляем "1" в оставшиеся нулевые значения признака}

If (t<>j) **And** ((c **And** (1 **ShL** (i*3+t)))=0) **Then**

c:=c+ (1 **ShL** (i*3+t)); {Командой ShL

осуществляется сдвиг единицы на заданное количество разрядов}

ChangeY:=c;

End;

Function ChangeN(s,i,j:Integer):Integer;

Begin

ChangeN:=s+1 **ShL** (i*3+j); {Запрещаем это значение признака}

End;

Методическое отступление. Здесь целесообразно, на наш взгляд, сделать паузу в обсуждении и выполнить небольшой

тренинг по «ручному» просчету перехода из одного состояния в другое при разных вариантах ответов на вопросы. При этом следует состояния описывать как в двоичном представлении, так и в десятичном.

Вторым шагом нашей работы является описание всего пространства состояний (количество состояний 2^{3^p}). Достаточно ли для такого описания одного одномерного массива? Чтобы ответить на этот вопрос, требуется ответить на другой вопрос: какие бывают состояния? Например, недопустимые. Так, состояние 000011 (3_{10}) является недопустимым (недостижимым) для примера из формулировки задачи — первый признак коров не может быть равен не X и не Y. Состояние 010100 (20_{10}) является допустимым для первой и третьей коров и недопустимым для второй и четвертой коров (второй признак у коров не может быть не равен единице). Наконец, есть состояния, допустимые только для одной коровы. Так, в примере состояние 011100 (28_{10}) допустимо только для первой коровы.

Введем два массива $w[0..2^{3^p}-1]$ и $d[0..2^{3^p}-1]$. При инициализации элементам d присвоим значение 0, а элементам w — -1. На начальной стадии «разметим» пространство состояний:

$$w[i] = \begin{cases} t, \text{ номер коровы (первой), для которой состояние } i \\ \text{допустимо;} \\ 255, \text{ если состояние } i \text{ недопустимо ни для одной} \\ \text{коровы;} \end{cases}$$

$$d[i] = \begin{cases} 255, \text{ состояние } i \text{ допустимо для нескольких коров;} \\ 0, \text{ состояние } i \text{ недопустимо или допустимо только} \\ \text{для одной коровы.} \end{cases}$$

Например, $w[9] = 4$, $d[9] = 0$ — состояние 001001 (9_{10}) — допустимо только для коровы с номером 4; $w[12] = 1$, $d[12] = 255$ — состояние 001100 (12_{10}) допустимо для нескольких коров (больше одной), в частности для коровы с номером 1; $w[15] = 255$, $d[15] = 0$ — состояние 001101 (15_{10}) недопустимо ни для одной коровы.

Как разметить пространство состояний? Возьмем признаки первой коровы XZ и зафиксируем нули в соответствующих позициях — 0****0. (В позициях, отмеченных символом «*», допустима запись как 0, так и 1.) Другими словами, нам требуется сгенерировать все последовательности, у которых в позициях, отмеченных «*», записаны 0 или 1. И это необходимо сделать

для каждой коровы. В табл. 15.13 приведены допустимые последовательности для всех коров.

Таблица 15.13

1		2		3		4	
s		s		s		s	
2	011110	2	101110	1	110101	1	101101
4	000010	4	000010	1	000001	1	000001
4	000100	4	000100	4	000100	4	000100
6	000110	6	000110	5	000101	5	000101
8	001000	8	001000	16	010000	8	001000
10	001010	10	001010	17	010001	9	001001
12	001100	12	001100	20	010100	12	001100
14	001110	14	001110	21	010101	13	001101
16	010000	32	100000	32	100000	32	100000
18	010010	34	100010	33	100001	33	100001
20	010100	36	100100	36	100100	36	100100
22	010110	38	100110	37	100101	37	100101
24	011000	40	101000	48	110000	40	101000
26	011010	42	101010	49	110001	41	101001
28	011100	44	101100	52	110100	44	101100
30	011110	46	101110	53	110101	45	101101

Анализируя данные из табл. 15.13, видим, что некоторые состояния допустимы только для одной из коров. Назовем их *конечными*. В табл. 15.14 приводятся конечные состояния для каждой коровы.

Таблица 15.14

s	1	s	2	s	3	s	4
18	010010	34	100010	17	010001	9	001001
22	010110	38	100110	21	010101	13	001101
24	011000	42	101010	48	110000	41	101001
26	011010	46	101110	49	110001	45	101101
28	011100			52	110100		
30	011110			53	110101		

Примечание. Для удобства работы опишем признаки коров с помощью числовой матрицы ($r[1..n, 1..p]$), формируемой при вводе данных. Для примера из формулировки задачи она имеет вид:

$$r = \begin{pmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}.$$

Значение 0 соответствует X, 1 — Y и 2 — Z.

Один из вариантов выполнения действий по начальной разметке пространства состояний задачи может быть следующим. В процедуре решения задачи организуем цикл по i (коровам) с вызовом рекурсивной процедуры разметки `For i:=1 To n Do GenEq(0, 0, 0)`. При $s = 0$ (в начальный момент времени) может быть задан любой из $3 \cdot p$ вопросов. Вопрос опишем парой (a, b) , где a — номер признака, а b — номер вопроса. Тогда пары $(0, 1), (0, 2), (0, 3), \dots, (p-1, 1), (p-1, 2), (p-1, 3)$ соответствуют состоянию $s = 0$ и мы их назовем *начальными*. Для нашего примера они равны $(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)$.

Procedure GenEq($a, b, s: \text{Integer}$); { a — номер признака, b — номер значения признака, s — номер состояния. Начинаем для каждой коровы с нулевого состояния}

Begin

If $b=3$ **Then Begin** $a:=a+1$; $b:=0$; **End**; {Если значение признака равно трем, то переходим к следующему признаку}

If $(a < p)$ **And** $(b < 3)$ **Then Begin**

GenEq($a, b+1, s$); {Переход к следующему значению признака}

If $r[i, a+1] \neq b$ **then** GenEq($a, b+1, s+(1 \text{ ShL } (a*3+b))$); {Если у коровы с номером i признак a не равен значению b , то можно выполнить переход в другое состояние. Его номер вычисляется путем сдвига единицы на количество разрядов, соответствующих значениям a и b . Плюс единица выравнивает нумерацию признаков (с нуля или с единицы) в матрице r и в процедуре GenEq}

End

Else

If $w[s]=255$ **Then** $w[s]:=i$ **Else** $d[s]:=255$; {Если в состоянии s мы не были ($w[s] = 255$), то присваиваем значение номера коровы; иначе помечаем, что состояние s допустимо для нескольких коров}

End;

Методическое отступление. Теперь у нас появилась замечательная работа, способствующая пониманию задачи, — просчитать разметку пространства состояний как с использованием процедуры GenEq, так и без нее. Конечно, выходить за значение $p > 2$ (количество признаков) неразумно, но значения признаков можно изменять.

После разметки пространства мы имеем начальное, конечные, промежуточные и недопустимые (отмечены в w значением 255) состояния. Для рассматриваемого примера в табл. 15.14 даны конечные состояния (в w записан номер коровы, а в d — значение 0). Если в табл. 15.13 убрать конечные состояния, то в ней останутся только промежуточные состояния. Их для нашего примера — 17 (различных). Промежуточные состояния отмечены в массиве d значением 255 (после разметки). Осталось просчитать для каждого промежуточного состояния в зависимости от заданных вопросов возможные переходы в другие состояния и зафиксировать в соответствующем элементе массива d минимальное количество вопросов в худшем случае. Другими словами, требуется составить *таблицу переходов* для всех промежуточных состояний.

Элементом массива w для состояния s является код признака и его значение, они определяют новое состояние. Эта работа выполняется с помощью следующей рекурсивной функции.

```
Function Calc(s:Integer):Integer; {Находим минимальное
количество вопросов, необходимых для достижения
одного из конечных состояний из состояния s}
  Var q,i,j,s1,s2:Integer;
  Begin
    If d[s]=255 Then Begin {Конечные и недопустимые
состояния не рассматриваем. Нас интересуют только
промежуточные}
      For i:=0 To p-1 Do {Цикл по признакам}
        For j:=0 To 2 Do {Цикл по значениям признака}
          If s And (1 ShL (i*3+j))=0 Then Begin {Если
значение j признака i в состоянии s допустимо, то
моделируем переходы из него при ответах "да" и "нет"}
            s1:=ChangeY(s,i,j); {Переход в состояние
s1 при ответе "да"}
            s2:=ChangeN(s,i,j); {Переход в состояние
s2 при ответе "нет"}
            If (s1<>s) And (s2<>s) Then Begin
              q:=max(Calc(s1),Calc(s2)); {Находим
максимальное количество вопросов (худший вариант
Задания вопросов)}
```

```

      If (q+1)<d[s] Then Begin {Если с учетом
заданного вопроса этот вариант лучше ранее
рассмотренных, то фиксируем его}
          d[s]:=q+1; {Количество вопросов}
          w[s]:=(i+1)*10+j; {Код — из номера
признака и его значения (в десятичном виде), при
котором из состояния s реализуется вариант
с наименьшим количеством вопросов}
      End;
    End;
  End;
  Calc:=d[s]
End;

```

Практически мы закончили обсуждение решения задачи, обеспечивающего сформулированные в условии требования.

Методическое отступление. Промоделировать работу проверяющей программы путем организации диалога компьютера с человеком можно с помощью следующей процедуры:

```

Procedure Solve;
  Var s,c,i,j:Integer;
  Begin
    <элементам массива d присвоить значение 0>;
    <элементам массива w присвоить значение 255>;
    For i:=1 To n Do GenEq(0,0,0); {Начальная
разметка}
    Calc(0); {Формирование таблицы переходов}
    s:=0;
    While d[s]<>0 Do Begin
      WriteLn('Q ',w[s] Div 10,' ',CnvI[w[s] Mod 10]);
      {Задаем вопрос. Массив констант CnvI имеет вид:
CnvI:Array[0..2] Of Char=('X','Y','Z');}
      ReadLn(c); {Человек отвечает}
      If c=0 Then s:=ChangeN(s,w[s] Div 10-1,w[s]
Mod 10)
      Else s:=ChangeY(s,w[s] Div 10-1,w[s] Mod 10);
      {Переход в другое состояние. Значение "-1" требуется
для выравнивания нумерации}
    End;
    WriteLn('C ',w[s]);
  End;

```

Если в $d[s]$ хранится наименьшее количество вопросов для перехода из состояния s в одно из конечных состояний, то

структура значения элемента массива w хотя и просматривается, но, возможно, не до конца ясна. Ее можно пояснить с помощью рис. 15.1, на котором приводятся три из шести возможных вопросов для начального состояния. Рядом с вершинами в угловых скобках приводится номер признака и номер его значения, представленные в вопросе, или, если это конечное состояние, — номер коровы.

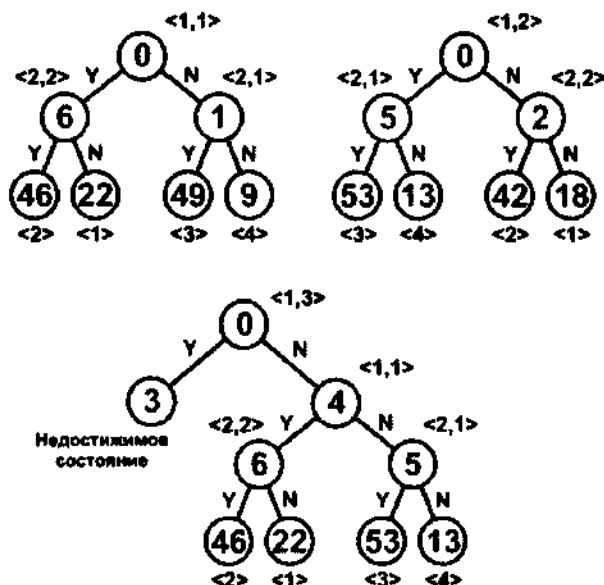


Рис. 15.1. Примеры переходов из начального состояния

Методические рекомендации

Это — задача на *«метод решета»*. Предварительно, до ее разбора, следует, помимо традиционного *решета Эратосфена*, рассмотреть более простые задачи. Например, задачу о *«быках и коровах»**. В этой задаче пространство решения строится более просто. Другой вариант разбора, с большей долей математических рассуждений, можно найти в статье М. Бабенко и С. Дятлова**.

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 109–112

** Бабенко М., Дятлов С. Решение задач международной олимпиады школьников по информатике 2003 года. Угадайте корову // «Информатика», еженедельное приложение к газете «Первое сентября». 2003. № 43. С. 13–15.

Самостоятельные исследования

Размерность массивов d и w определяется значением p ($2^{3 \cdot p}$). Очевидно, что при увеличении p требования к памяти становятся неприемлемыми, но при этом используется не вся память. Так, в рассмотренном примере из 2^6 состояний 20 являются конечными, 18 — промежуточными, а остальные, в принципе, не требуются. Используя другую организацию данных, можно попытаться увеличить допустимое значение p .

«Роботы в лабиринте»

Условие задачи

Вы являетесь счастливым обладателем двух роботов, которые расположены в двух разных прямоугольных лабиринтах. Квадраты с координатами (1, 1) расположены в верхних левых (северо-западных) углах лабиринтов. В лабиринте с номером i ($i = 1, 2$) находится g_i охранников ($0 \leq g_i \leq 10$), пытающихся поймать роботов. Для этого они патрулируют лабиринты вперед и назад по своему участку патрулирования, представляющему собой несколько последовательных клеток лабиринта на одной прямой. Ваша задача — определить последовательность команд, приводящую к выходу обоих роботов из лабиринтов без их поимки охранниками.

В начале каждой минуты обоим роботам посылается одна и та же команда, которая представляет собой одно из четырех направлений (север, юг, восток или запад). Получив команду, робот перемещается на одну клетку в указанном направлении. Если выполнение команды приводит к столкновению робота со стеной, то вместо выполнения этой команды робот в течение данной минуты остается на месте. Считается, что робот выходит из лабиринта, когда оказывается за его пределами. Выйдя из лабиринта, робот игнорирует все последующие команды.

Для каждого охранника в начальный момент известна его позиция в лабиринте и направление, в котором он начнет двигаться. В начале каждой минуты одновременно с роботами охранники перемещаются на одну клетку. Охранник движется вперед со скоростью одна клетка в минуту до тех пор, пока он не достигнет конца своего участка патрулирования (то есть пока он не сделает на одно перемещение меньше, чем число клеток в его участке патрулирования). По достижении этого момента охранник мгновенно разворачивается и в дальнейшем продолжает движение в обратном направлении к своей началь-

ной позиции. Достигнув начальной позиции, он снова разворачивается и повторяет свое движение по участку патрулирования до тех пор, пока оба робота не выйдут из лабиринтов.

Участки патрулирования охранников таковы, что они не проходят через стены и не выходят за пределы лабиринта. Участки патрулирования охранников могут пересекаться, однако никакие два охранника никогда не столкнутся, то есть они никогда не будут занимать одну и ту же позицию к концу одной и той же минуты. Охранники также не будут одновременно находиться в соседних клетках и двигаться навстречу друг другу в течение этой минуты. Начальное положение охранника не будет совпадать с начальным положением робота.

Считается, что охранник поймал робота, если охранник и робот оказываются в одной и той же позиции к концу минуты, либо если охранник и робот находятся в соседних клетках и движутся навстречу друг другу.

По заданному описанию двух лабиринтов (размер каждого из лабиринтов не больше 20×20), известным начальным позициям каждого из роботов и участкам патрулирования охранников определите последовательность команд, приводящую к выходу из лабиринтов обоих роботов без поимки их охранниками. При этом нужно минимизировать время, которое потребуется для того, чтобы оба робота вышли из лабиринтов. Если роботы выходят из лабиринтов в разные моменты времени, то время того робота, который вышел раньше, не существенно (учитывается лишь время того робота, который вышел позднее).

Входные данные — robots.in.

Первая часть строк во входном файле описывает первый лабиринт и тех, кто в нем находится. Соответственно, вторая часть описывает второй лабиринт и тех, кто в нем находится.

- Первая строка входного файла содержит два разделенных пробелами целых числа r_1 и c_1 — количества строк и столбцов в первом лабиринте соответственно.
- В каждой из последующих r_1 строк содержится по c_1 символов, описывающих конфигурацию лабиринта. Начальное положение робота обозначается символом «X». Символ «.» обозначает пустую клетку, а символ «#» — стену. В лабиринте всегда ровно один робот.
- После описания лабиринта следует строка, содержащая единственное целое число g_1 — количество охранников в первом лабиринте ($0 \leq g_1 \leq 10$).
- Каждая из последующих g_1 строк описывает участок патрулирования охранника в виде трех целых чисел и символа, разделенных пробелами. Первое и второе числа указывают

строку и столбец начального положения охранника. Третье число (2...4) представляет собой количество клеток в участке патрулирования данного охранника. Символ «N», «S», «E» или «W» (север, юг, восток и запад, соответственно) указывает начальное направление движения охранника.

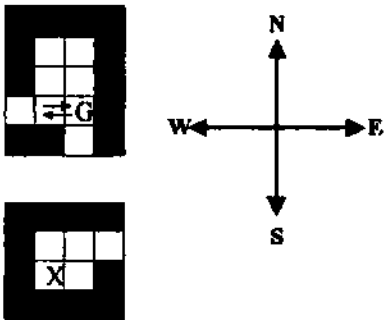
Описание второго лабиринта и тех, кто в нем находится, следует за описанием первого лабиринта в том же формате.

Пример входных данных:

```

5 4
#####
#X.#
#..#
...#
##.#
1
4 3 2 W
4 4
#####
#...
#X.#
#####
0

```



Выходные данные — robots.out.

Если искомое решение существует, то первая строка выходного файла должна содержать единственное целое число k ($k \leq 10000$) — количество команд для обоих роботов, приводящих к их выходу из лабиринтов без поимки охранниками. Гарантируется, что в этом случае кратчайшая последовательность команд имеет длину не больше 10000. В последующих k строках должны располагаться команды, каждая из которых представляет собой один из символов «N», «S», «E» или «W». Если решения не существует, то необходимо вывести единственную строку, содержащую «-1».

После выполнения представленной последовательности команд оба робота должны оказаться за пределами лабиринтов. Последняя команда должна приводить к выходу из лабиринта хотя бы одного из роботов. Если существует несколько решений с минимальным временем, то любое из них считается правильным.

Пример выходных данных:

8
E
N
E
S
S
S
E
S

Ограничения:

Время выполнения	2 секунды CPU
Память	64 MB

Система оценки.

В тех тестах, где решения не существует, будет оцениваться только правильный ответ. Частичные баллы за другие тесты будут начисляться следующим образом.

Корректность: 20% баллов.

Выходной файл считается корректным, если он оформлен в соответствии с требованиями условия задачи, содержит не более 10000 команд и их последовательность приводит к выходу обоих роботов из лабиринтов. При этом последняя команда должна приводить к выходу хотя бы одного из роботов.

Минимальность: 80% баллов.

Считается, что решение, представленное в выходном файле, минимальное, если оно корректно и не существует более короткой последовательности команд, которая также корректна. Решение, в котором последовательность команд не минимальна, не получает баллов за минимальность*.

Обсуждение задачи

Задача о лабиринте — это классика информатики. Известно, что *метод волны* как модификация предложений Э. Люка (1891 г.) дает кратчайший путь для выхода из лабиринта, если

* Бабенко М., Орлов Д. Решение задач международной олимпиады школьников по информатике 2002 года. Роботы в лабиринте // «Информатика», еженедельное приложение к газете «Первое сентября». 2003. № 41. С. 22–24.

последний существует*. Методические особенности преподавания темы можно найти в работе С. Ю. Иванова**.

Первый вывод, следующий из ограничений задачи, — что нам требуется нерекурсивная схема реализации логики метода волны при поиске выхода из лабиринта. В этом случае необходимо введение очереди как абстрактной структуры данных для хранения очередности просмотра квадратов лабиринта, а «поле» лабиринта описывается двумерным массивом $A[n*n]$, где n — размерность задачи. Для примера из формулировки задачи результат использования метода волны (мы пока не рассматриваем присутствие охранников) приведен на рис. 15.2. В свободных полях лабиринта записывается значение метки — оно на единицу больше значения, записанного в квадрате, из которого мы пришли в текущий квадрат.

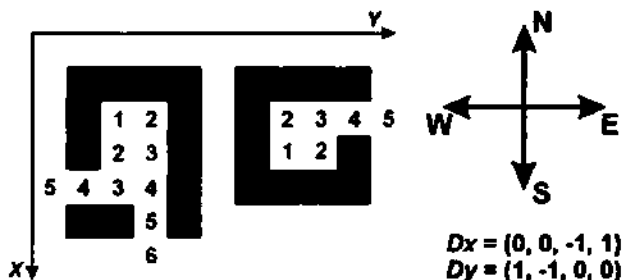


Рис. 15.2. Использование метода волны для примера из формулировки задачи. Массивами констант Dx , Dy определяется очередность просмотра соседних квадратов («E», «W», «N», «S»)

В данном случае (рис. 15.2) задача решалась отдельно для каждого лабиринта. В очереди, — это двумерный массив (назовем его *Тигп*), например для второго лабиринта, — хранились координаты: $Тигп[x_1, y_1] = ((3, 2), (3, 3), (2, 2), (2, 3), (2, 4), (2, 5))$.

Методическое отступление. В этот момент обсуждения задачи целесообразно, на наш взгляд, рисование произвольных лабиринтов и «ручное» заполнение матрицы A и очереди *Тигп*. Это позволит как проверить знание метода волны, так и актуализировать его.

Давайте ответим на вопрос — что вносит в задачу совместное выполнение команд роботами? Дается команда, и оба робота ее выполняют. Первое, и очевидное, изменение — поле лабиринта

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2002. С. 89–92.

** Иванов С. Ю. Метод волны: от простого к сложному // Вестник ВятГГУ. Информатика. № 2. 2004. С. 177–180.

та $A[x_1, y_1, x_2, y_2]$ и очередь $Turn[x_1, y_1, x_2, y_2]$ становятся четырехмерными массивами. Анализируя пример из формулировки задачи, мы видим, что для данной пары лабиринтов решения нет. В чем дело? В формулировке задачи есть ключевая фраза (мы ее выделили курсивом) — «если выполнение команды приводит к столкновению робота со стеной, то вместо выполнения этой команды робот в течение данной минуты остается на месте». Следствие: в очередь и в поле лабиринта требуется вводить пятый параметр — время! Только в этом случае нахождение робота в течение нескольких минут в одном квадрате становится различным с точки зрения логики использования метода волны.

Итак, мы имеем: $A[x_1, y_1, x_2, y_2, t]$ и $Turn[x_1, y_1, x_2, y_2, t]$. Однако принципы работы с этими структурами данных не изменились, они остались прежними, как и в простейшем случае поиска выхода из обычного лабиринта, ибо они определяются сутью метода волны, и только!

Рассмотрим пример. Последовательность команд: «Е», «N», «Е», «Е», «S», «S», «S», «S» выводит роботов из лабиринтов. Изменение координат роботов с привязкой ко времени приведено в табл. 15.15.

Таблица 15.15

x_1	y_1	x_2	y_2	t	Примечание
2	2	3	2	0	Начальное состояние
2	3	3	3	1	Выполнена команда «Е»
2	3	2	3	2	Выполнена команда «N». Первый робот находится в том же квадрате. Изменяется только время
2	3	2	4	3	Выполнена команда «Е»
2	3	2	5	4	Выполнена команда «Е»
3	3	2	5	5	Выполнена команда «S». Второй робот вышел из лабиринта — последующие команды игнорируются
4	3	2	5	6	Выполнена команда «S»
5	3	2	5	7	Выполнена команда «S»
6	3	5	5	8	Выполнена команда «S»

Приведем модификацию известной логики поиска выхода из лабиринта на псевдокоде. Изменения, с учетом сделанных за-

мечаний, касаются только структур данных (оператор проверки условия встречи с охранниками мы пока не рассматривали, но вставили в псевдокод, чтобы определить его место в схеме решения задачи):

```

...
up:=1;down:=1; {Указатели очереди}
A[x1,y1,x2,y2,0]:=1; {Начальное положение роботов}
Write_Turn(x1,y1,x2,y2,0); {Запись в очередь}
While down<up Do Begin {Пока очередь не пуста}
  Read_Turn(a1,b1,a2,b2,t); {Чтение из очереди.
Изменяется значение down}
  For i:=1 To 4 Do Begin {Просмотр соседних квадратов}
    <Изменить координаты роботов a1, b1, a2, b2 на
значения dx[i] и dy[i]. Новые значения координат
nx1, ny1, nx2, ny2. Если квадрат с координатами
(nx1,ny1) или (nx2, ny2) занят стеной, то новые
значения координат соответствующего робота совпадают
с прежними>;
    If <(первый робот не встречается с охранниками
в первом лабиринте) And (второй робот не встречается
с охранниками во втором лабиринте)> Then
      If A[nx1, ny1, nx2, ny2,t+1]=0 Then Begin
        A[nx1, ny1, nx2, ny2,t+1]:=A[a1,b1,a2,b2,t]+1;
        <Записать положение роботов, требуемое для
вывода результата, в дополнительную структуру данных>;
        Write_Turn(nx1, ny1, nx2, ny2,t+1); {Запись
в очередь. Изменяется значение up}
      End;
    End;
  End;
End;
...

```

Методическое отступление. Рекомендуется для двух произвольных лабиринтов (не очень большой размерности) «вручную» найти выход путем составления таблицы по типу табл. 15.15. Трассировка работы с массивами *A* и *Turn* в данном случае очень трудоемка.

Вернемся к нашему примеру и попытаемся учесть наличие охранника. Добавим в табл. 15.15 столбец, в котором будем фиксировать положение охранника в различные моменты времени — (*x*, *y*). Изменения положения роботов и охранника приведены в табл. 15.16.

Таблица 15.16

x_1	y_1	x_2	y_2	t	Положение охранника	Примечание
2	2	3	2	0	(4,3)	Начальное состояние
2	3	3	3	1	(4,2)	Выполнена команда «Е»
2	3	2	3	2	(4,3)	Выполнена команда «N». Первый робот находится в том же квадрате. Изменяется только время
2	3	2	4	3	(4,2)	Выполнена команда «Е»
2	3	2	5	4	(4,3)	Выполнена команда «Е»
3	3	2	5	5	(4,2)	Выполнена команда «S». Второй робот вышел из лабиринта — последующие команды игнорируются
4	3	2	5	6	(4,3)	Выполнена команда «S»
5	3	2	5	7	(4,2)	Выполнена команда «S»
6	3	5	5	8	(4,3)	Выполнена команда «S»

Мы видим, что в момент времени $t = 6$ первый робот и охранник находятся в одном квадрате. Этот выход из лабиринта не удовлетворяет условиям задачи, но ответ есть, и он приведен в формулировке. Как реализовать функцию «робот не встречается с охранниками»? Во-первых, условие задачи — «считается, что охранник поймал робота, если охранник и робот оказываются в одной и той же позиции к концу минуты, либо если охранник и робот находятся в соседних клетках и двигаются навстречу друг другу» — накладывает ограничения. Требуется передавать нашей функции не только текущие координаты робота и время, но и предыдущее его положение. Иначе движение робота и охранника навстречу друг другу не удастся зафиксировать. Во-вторых, трассу движения каждого охранника хранить и просчитывать не следует, в этом нет необходимости. Знание начального положения охранника, длины участка его патрулирования и текущего значения t позволяет определить местонахождение охранника как в текущий, так и предыдущий моменты времени.

Методические рекомендации

Это не первая задача на международных олимпиадах, связанная с тематикой поиска выхода из лабиринта (см. задачи про «прямоугольники» 1993 года, «замок» 1994 года). Программная реализация рассмотренной логики трудоемка, также как и тестирование, и требует продуманных структур данных,

поэтому подобной работе должно предшествовать изучение классических вариантов задачи.

Самостоятельные исследования

1. Сделайте рекурсивную реализацию задачи. Сравните временные характеристики решений для одних и тех же входных данных.

Примечание. Исследование потребует написания генератора тестов для максимальных значений размеров лабиринтов.

2. Снимем ограничение на участки патрулирования охранников. Пусть есть один лабиринт и один охранник, свободно перемещающийся по лабиринту; его начальное положение известно. Требуется найти выход из лабиринта для робота (он не простаивает по времени в одном квадрате) при тех же условиях. (Дальнейшее усложнение задачи очевидно — несколько охранников, два лабиринта и т. д.)

«Разглядывание забора»

Условие задачи

Фермер Дон осматривает забор, которым окружено его квадратное плоское поле размером n на n метров ($2 \leq n \leq 500\,000$). Один угол забора расположен в точке $(0, 0)$, противоположный ему угол находится в точке (n, n) . Стороны забора параллельны осям X и Y .

Столбы, к которым крепится забор, стоят не только по углам, но и через метр вдоль каждой стороны поля, всего в заборе $4 \cdot n$ столбов. Столбы стоят вертикально и не имеют толщины (радиус равен нулю). Фермер Дон хочет определить, сколько столбов он увидит, если встанет в определенной точке поля.

Проблема заключается в том, что на поле фермера Дона расположено r огромных камней ($1 \leq r \leq 30\,000$), за которыми не видны некоторые столбы, поскольку Дон недостаточно высок, чтобы смотреть поверх этих камней. Основание каждого камня представляет собой выпуклый многоугольник ненулевой площади, вершины которого имеют целочисленные координаты. Камни стоят на поле вертикально. Камни не имеют общих точек между собой, а также с забором. Точка, где стоит фермер Дон, лежит внутри, но не на границе поля, а также не лежит внутри и на границе камней.

По заданным размеру поля фермера Дона, положению и форме камней на поле, месту, где стоит фермер Дон, вычислите количество столбов, которые может видеть фермер Дон. Если вершина основания камня находится на одной линии с местом расположения Дона и некоторым столбом, то Дон не видит этот столб.

Входные данные — boundary.in:

- первая строка входного файла содержит два целых числа n и r , разделенных пробелом;
- вторая строка входного файла содержит два целых числа — координаты X и Y места, где стоит фермер Дон на поле;
- последующие строки входного файла описывают положение r камней:
 - описание i -го камня начинается со строки, которая содержит целое число p_i ($3 \leq p_i \leq 20$), определяющее количество вершин в основании камня,
 - каждая из последующих p_i строк содержит пару разделенных пробелом целых чисел X и Y , которые являются координатами вершины. Вершины основания камня различны между собой и перечислены в направлении против часовой стрелки.

Пример входных данных (рис. 15.3):

```
100 1
60 50
5
70 40
75 40
80 40
80 50
70 60
```



Обратите внимание на то, что основание камня имеет три вершины на одной прямой: $(70, 40)$, $(75, 40)$ и $(80, 40)$.

Рис. 15.3. Пример из формулировки задачи

Выходные данные — boundary.out:

Выходной файл должен содержать одну строку с одним целым числом, обозначающим количество столбов, которые будет видеть фермер Дон.

Пример выходных данных: 319.

Ограничения:

Время выполнения	1 секунда CPU
Память	64 MB

Система оценки. За каждый тест, на котором ваша программа выведет правильный ответ, вы получите полный балл. Неправильный ответ на тест не оценивается.

Обсуждение задачи

Упростим задачу. Пусть имеется один камень, и он расположен так, как показано на рис. 15.4. Координаты фермера — (45, 40), координаты углов камня — (50, 80), (85, 60), (90, 95).

В геометрических задачах при их относительной логической простоте сложность заключается в написании компактного кода. В данном конкретном случае требуется искать точки пересечения прямых линий с четырьмя границами. Желательно сделать одну процедуру. В вызывающей логике должно быть определение абсциссы и ординаты точки (угла многоугольника) в зависимости от границы, с которой ищется пересечение.

Многое в этом случае зависит от выбора структур данных. Введем массив координат углов многоугольника — `s:Array[1..30000,0..20,0..1] Of Integer` (может быть, потребуется длинное целое число — 64 бита); координат фермера — `pos:Array[0..1] Of Integer`; количество вершин у камней — `v:Array[1.. 30000] Of Integer`; массив видимых столбов — `a: Array[0..4*500000+1] Of Integer`.

В основной логике нет сложностей — она оформляется по принципу «как говорю, так и пишу» (один из критериев того, что между структурами данных и управлением вычислительным процессом достигнуто единство, называемое программой):

Procedure Solve;

Var `i,j,k,l,r:Integer`; {Для переменных `l` и `r` возможно потребуется другой тип данных}

Begin

For `i:=1 To c Do` {Цикл по камням}

For `j:=0 To v[i]-1 Do` {Цикл по граням камня.
Координаты первой вершины камня требуется записать дважды, в начало и конец}

If <Векторное произведение (`pos[0]-s[i,j,0]`, `pos[1]-s[i,j,1]`) и (`s[i,j+1,0]-s[i,j,0]`, `s[i,j+1,1]-s[i,j,1]`) больше нуля> **Then**

For `k:=0 To 3 Do Begin` {Цикл по сторонам поля}

`x:=k Mod 2; y:=1-x;` {Этот прием в сочетании со структурами данных позволяет сделать дальнейшую обработку практически независимой от границы.
Глобальные переменные}

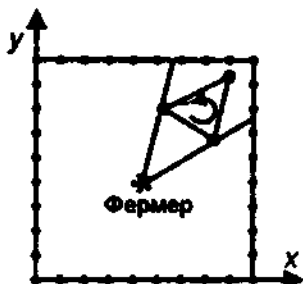


Рис. 15.4. Простой случай расположения камня на поле

```

    GetInvisibleSegment(k, s[i,j,x], s[i,j,y],
s[i,j+1,x], s[i,j+1,y], l, r); {В переменных r и
l получаем значения границ невидимого участка прямой}
    If (r>=0) And (l<=n) And (r<=l) Then Begin
{Помечаем начало (+1) и конец (-1) невидимого
участка границы)
        a[n*k+r]:=a[n*k+r]+1; {Начальное значение
элементов массива a равно нулю}
        a[n*k+l+1]:=a[n*k+l+1]-1;
    End;
End;
End;

```

После этой работы подсчет количества видимых столбов осуществляется за линейное время:

```

Procedure Done;
Var i,cnt,k:Integer;
Begin
    k:=0; cnt:=0; {cnt – счетчик количества видимых
столбов}
    For i:=0 To 4*n-1 Do Begin {n – размер поля}
        k:=k+a[i];
        If k=0 Then cnt:=cnt+1; {Увеличиваем значение
cnt только на видимом участке забора}
    End;
    WriteLn(cnt);
End;

```

Уточняющая логика не содержит циклических конструкций, поэтому время решения задачи есть $O(n+20*c)$.

Методическое отступление. Прежде чем перейти к обсуждению деталей логики, желательно «прорисовать» параметры процедуры GetInvisibleSegment для четырех сторон забора.

Напомним* суть векторного произведения $v \cdot w$. Результат его есть вектор, такой, что:

- длина вектора $v \cdot w$ равна $|v| * |w| * \sin(\varphi)$, где φ — угол между векторами;
- вектор $v \cdot w$ перпендикулярен векторам v и w , т. е. перпендикулярен плоскости этих векторов;
- вектор $v \cdot w$ направлен так, что из конца этого вектора *кратчайший поворот* от v к w виден происходящим против часовой стрелки.

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2002. С. 224–225.

Для нас в данной задаче важно третье свойство. Направление вектора определяется знаком выражения $v_x w_y - v_y w_x$. Для видимых сторон многоугольника он положительный, а для невидимых — отрицательный. У треугольника на рис. 15.4 видима только одна сторона — отрезок $(50, 80) - (85, 60)$.

Методическое отступление. Рекомендуется вычислить значения векторных произведений (их знаков) для различных расположений на плоскости фермера и камня, в частности для данных на рис. 15.4. Иллюстрация векторного произведения для этого случая приведена на рис. 15.5.

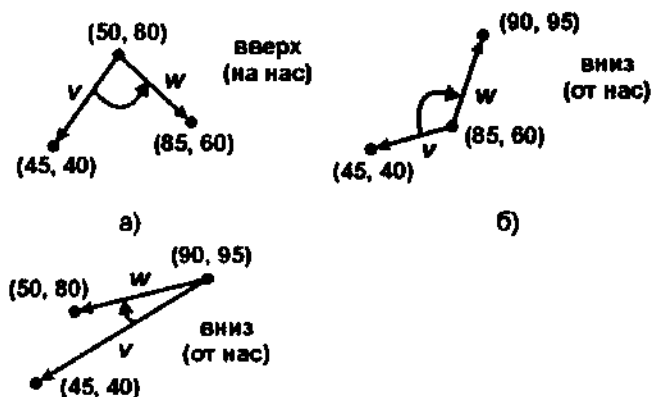


Рис. 15.5. Иллюстрация векторного произведения

Перейдем к обсуждению процедуры `GetInvisibleSegment`. Требуется найти границы невидимого отрезка (значения l и r) конкретной стороны забора (определяется значением переменной k). При этом $r \geq 0$, $l \leq n$ и $r \leq l$. Если эти условия не выполняются, то точки пересечения находятся за пределами границы поля или вся граница видима.

Рассмотрим случай $k = 0$ (эта ситуация показана на рис. 15.6). По большому счету, требуется только найти две точки пересечения прямых, но прежде необходимо учесть, какая из границ рассматривается:



Рис. 15.6. Поиск границ невидимого отрезка границы поля (случай $k = 0$)

Procedure `GetInvisibleSegment` (k :Integer; x_0, y_0, x_1, y_1 :Work; var l, r :Work); {Тип Work — целый тип данных, 64 бита}
 Var q :Work;

Begin

If (k=1) **Or** (k=2) **Then** q:=n **Else** q:=0;

{Определяем тип границы поля}

If (Abs(y0-q)>=Abs(pos[y]-q)) **And** (Abs(y1-q)>=Abs(pos[y]-q)) **Then** **Begin** {Вся граница поля видима. Проекция координат фермера и стороны многоугольника на прямую не пересекаются}

r:=-1;

l:=-1;

End

Else **Begin**

l:=Coord(k, q, x0, y0, **True**); {Вычисляем первую точку пересечения — левый крайний столб невидимого участка}

r:=Coord(k, q, x1, y1, **False**); {Вычисляем вторую точку пересечения — правый крайний столб невидимого участка}

End;

End;

Мы опять «зарыли» суть в следующий фрагмент логики (технология проектирования программ «сверху вниз») — функцию Coord (ее заголовок — Function Coord(k:Integer; q, rx, ry:Work; pp:Boolean):Work), ибо ситуация осложняется тем, что нам бы хотелось «остаться» в целочисленной области вычислений (не использовать вещественный тип данных).

Из рис. 15.6 следует, что значение r после целочисленного деления необходимо увеличить на единицу. Для всех границ поля правила изменения значений r и l показаны на рис. 15.7. Для их учета в параметры Coord введена логическая величина. Ее значение, а также значение k , позволяют учесть все возможные варианты.

Методическое отступление. Желательно, используя рис. 15.7, составить таблицу изменения r и l в зависимости от значения k и логической величины, определяющей номер точки пересечения.

Обозначим через $p(p_x, p_y)$ вершину многоугольника (p_x и p_y — это не обязательно абсцисса и ордината точки — они зависят от

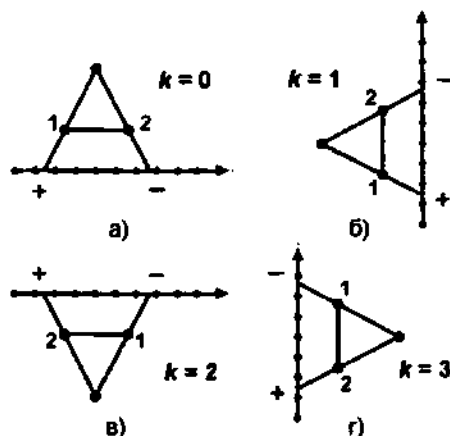


Рис. 15.7. Правило изменений значений r и l

типа границы поля). Зная координаты фермера $f(f_x, f_y)$, мы можем провести луч (fp) и найти его пересечение с прямой (ее тип определен в процедуре `GetInvisibleSegment` — $q = n$ или $q = 0$). Значение $f_x = pos[x]$, а $f_y = pos[y]$, где x и y (0 или 1) определяются k .

Уравнения луча в параметрической форме имеют вид:

$$\begin{cases} x = p_x + t(f_x - p_x); \\ y = p_y + t(f_y - p_y). \end{cases}$$

Параметр t в этих уравнениях принимает неотрицательные значения. Решая уравнения, находим координату точки пересечения: $u := p_x - ((p_y - q) * (pos[x] - p_x)) \text{ Div } (pos[y] - p_y)$. Следует не забыть при $k > 1$ о необходимости изменения значения $u := n - u$ и учесть возможность равенства нулю знаменателя дроби.

Рассмотрим сквозной пример в обсуждении.

Случай $k = 1$ (прямая $x = n$):

- для первой точки $u = 67$; после учета целочисленного деления $u = 68$;
- для второй точки $u = 480$ — угол невидим, поэтому меняем на n , равное 100.

Случай $k = 2$ (прямая $y = n$):

- для первой точки $u = 165$, изменяем на $u = 100$ и, с учетом k , на $u = 0$;
- для второй точки $u = 52$, после учета целочисленного деления $u = 53$ и, с учетом k , $u = 47$.

Ответом задачи является число 320.

Методическое отступление. «Ручной» просчет для конкретного примера — необходимая составляющая такой работы. К кодированию решения следует приступать только после уточнения всех деталей, что позволит избежать многочисленных ошибок, присущих «быстрой» реализации геометрических задач.

Методические рекомендации

Если для олимпиад России задачи по вычислительной геометрии традиционны, то на международных олимпиадах школьников по информатике это одна из первых задач. Несмотря на традиционность тематики на российском уровне, нельзя сказать, что с методической точки зрения она проработана. Немно-

гочисленная литература* является скорее «руководством к действию», чем действительным помощником учителю. Причины, видимо, кроются в том, что здесь очень значительна математическая составляющая темы, а также объемность программных решений и сложность их тестирования. Элементарная (с логической точки зрения) задача о рассмотрении всех случаев пересечения двух произвольных треугольников на плоскости использовалась лет тридцать тому назад как тест на профессиональную пригодность при приеме программистов на работу.

Другой вариант разбора описываемой задачи можно найти в статье М. Бабенко и Д. Орлова**.

Самостоятельные исследования

1. Оцените упрощение программного кода (если оно есть) при использовании вещественной арифметики.
2. Усложняет ли задачу снятие условия выпуклости многоугольника в основании камней?

* Андреева Е. В., Босова Л. Л., Фалина И. Н. Математические основы информатики. М.: БИНОМ. Лаборатория знаний, 2005. С. 283–318; Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики. СПб.: БХВ-Петербург, 2003. 560 с.; Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2002. С. 222–265.

** Бабенко М., Орлов Л. Решение задачи XV международной олимпиады школьников по информатике. Разглядывание забора // Информатика, еженедельное приложение к газете «Первое сентября». 2004. № 13. С. 23–26.

ЗАДАЧИ ПЕРВОГО ТУРА

«Гермес»

Условие задачи

В современном городе греческих богов улицы расположены в виде целочисленной решетки и параллельны осям X и Y . Для каждого целого числа Z существуют горизонтальная улица с $Y = Z$ и вертикальная улица с $X = Z$. В каждой точке с целочисленными координатами находится уличный перекресток. Таким образом, каждая пара целых чисел задает некоторый перекресток. В жаркие дни боги отдыхают в кафетериях, расположенных на уличных перекрестках. Посыльный Гермес должен послать световые сообщения богам, отдыхающим в кафетериях, перемещаясь только по улицам города. Каждое сообщение предназначено только для одного бога, но ничего не случится, если его увидят другие боги.

Сообщения должны быть посланы богам строго в заданном порядке, поэтому Гермесу даны координаты кафетериев именно в этом порядке. Гермес стартует из точки с координатами $(0, 0)$. Для того, чтобы послать сообщение в кафетерий с координатами (X_i, Y_i) , Гермесу достаточно посетить некоторую точку на этой же горизонтальной улице (с Y -координатой Y_i) или на этой же вертикальной улице (с X -координатой X_i). После отправки всех сообщений Гермес останавливается.

Вы должны написать программу, которая по заданной последовательности кафетериев находит минимальную суммарную длину пути, который должен пройти Гермес для доставки всех сообщений.

Входные данные. Первая строка входного файла с именем `hermes.in` содержит одно целое число n — количество сообщений, которые должны быть отправлены. Следующие n строк содержат координаты n уличных перекрестков, куда должны быть посланы сообщения. Эти n строк определяют порядок, согласно которому должны быть посланы сообщения. Каждая из этих n строк содержит два целых числа: первое — X -координата и второе — Y -координата уличного перекрестка.

Выходные данные. Выходной файл с именем `hermes.out` должен содержать только одну строку с одним целым числом — минимальной суммарной длиной пути, необходимого Гермесу для посылки всех сообщений.

Пример.

```
hermes.in:
5
8 3
7 -7
8 1
-2 1
6 -5

hermes.out:
11
```

Ограничения: для всех тестов $1 \leq n \leq 20000$, $-1000 \leq X_i$, $Y_i \leq 1000$. В 50% тестов: $1 \leq n \leq 80$.

Обсуждение задачи

Упростим задачу. Пусть Гермесу разрешено перемещаться только вдоль оси X (в горизонтальном направлении); при этом он посылает только вертикальные лучи. Тогда, чтобы отправить сообщения, он должен двигаться так, как показано на рис. 16.1. Суммарная длина пути (S) при этом равна 28.

Введем второго Гермеса и разрешим ему двигаться (одновременно с первым) только в вертикальном направлении, по оси Y , и посылать горизонтальные лучи (рис. 16.2). Здесь $S = 27$.

Пока наша идея не конструктивна, ибо Гермесы не встречаются, если рас-

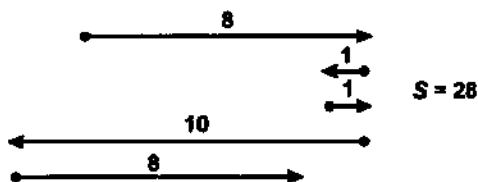


Рис. 16.1. Перемещение Гермеса в горизонтальном направлении для данных, приведенных в формулировке задачи

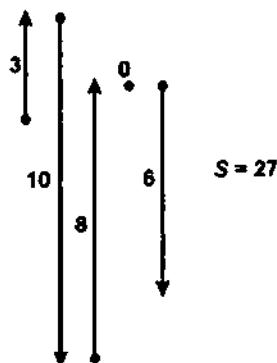


Рис. 16.2. Перемещение Гермеса в вертикальном направлении для данных, приведенных в формулировке задачи

смаатривать моменты времени, связанные с посылкой сообщений, а не в процессе их перемещений. Введем теперь «много» Гермесов, которые все стартуют из точки (0, 0) и каждый перемещается только по своей прямой — горизонтальной («горизонтальные Гермесы») или вертикальной («вертикальные Гермесы»). Другими словами, например, второй «горизонтальный Гермес» смещается на одну единицу вверх по оси Y, а затем перемещается только горизонтально, посылая вертикальные лучи.

Изменим немного исходные данные примера (для компактности таблиц), файл `hermes.in`, и вернемся к обсуждению задачи:

```

5
5 5
4 -5
5 2
-4 3
2 -2

```

Ответом задачи по-прежнему является значение 11 — минимальная суммарная длина.

В табл. 16.1 даны оценки длин для каждого Гермеса, а их у нас 22: из них 11 двигаются по горизонтальным прямым и 11 — по вертикальным прямым.

Таблица 16.1

	-5	-4	-3	-2	-1	0	1	2	3	4	5
$a[0]$	5	4	3	2	1	0	1	2	3	4	5
$a[1]$	10	9	8	7	6	5	6	7	8	9	10
$a[2]$	11	10	9	8	7	6	7	8	9	10	11
$a[3]$	12	11	10	9	8	7	8	9	10	11	12
$a[4]$	21	20	19	18	17	16	17	18	19	20	21
$a[5]$	27	26	25	24	23	22	23	24	25	26	27
$b[0]$	5	4	3	2	1	0	1	2	3	4	5
$b[1]$	10	9	8	7	6	5	6	7	8	9	10
$b[2]$	20	19	18	17	16	15	16	17	18	19	20
$b[3]$	27	26	25	24	23	22	23	24	25	26	27
$b[4]$	28	27	26	25	24	23	24	25	26	27	28
$b[5]$	33	32	31	30	29	28	29	30	31	32	33

Подсчитать их не составляет труда:

```
Const XMax=1000; {Максимальное значение координат
кафетериев}
```

```
...
For i:=-XMax To XMax Do Begin
  a[0,i]:=Abs(i);
  b[0,i]:=Abs(i);
End;
For i:=0 To n-1 Do
  For j:=-XMax To XMax Do Begin {x,y — массивы
для хранения координат кафетериев, min — функция
вычисления минимального из двух чисел}
    a[i+1,j]:=min(a[i+1,j],a[i,j]+Abs(x[i]-x[i+1]));
    b[i+1,j]:=min(b[i+1,j],b[i,j]+Abs(y[i]-y[i+1]));
    <...>;
  End;
```

Однако вопрос о том, «что это дает», остается открытым до тех пор, пока мы не определим: когда и как пересекаются Гермесы и что при этом может произойти?

Рассмотрим момент послышки сообщения первому богу, который находится в кафетерии с координатами (5, 5). Все «горизонтальные Гермесы» посылают вертикальный луч, и их координата равна (5, j), а «вертикальные Гермесы» посылают горизонтальный луч, и их координаты — (j, 5) ($-XMax < j < XMax$).

Остановимся на «вертикальных Гермесах» — все они находятся на пятой горизонтали, по которой «бегает» пятый «горизонтальный Гермес» (нумерация начинается с нуля), — он прошел расстояние, равное 10. Чтобы последнему Гермесу послать световой луч следующему богу с координатами (4, -5), ему надо вернуться на единицу назад, и его пробег составит 11. А если он «перехватит» оценку у одного из «вертикальных Гермесов» (они же на одной прямой!), у того, который до этой точки пробегает с меньшей оценкой, и будет считать эту оценку своей? Значит (в этот момент времени), мы имеем возможность несколько иначе корректировать именно оценку $a[2, 5]$.

Все эти рассуждения проиллюстрированы на рис. 16.3. Оценкой пятого «горизонтального Гермеса» будет 9, он ее «перехватывает» у четвертого «вертикального Гермеса».

Аналогичные рассуждения применимы и для пятого «вертикального Гермеса». Ему, чтобы послать следующий световой луч, требуется пробежать 20 единиц расстояния. Однако в это время все «горизонтальные Гермесы» находятся на пятой вер-

тикали, и наш Гермес может «перехватить» одну из оценок, — самую лучшую, если она есть. Что он и делает, перехватывая ее у «минус пятого» Гермеса. Она составит не 20, а только 10. Иллюстрация этого факта приведена на рис. 16.4. В последующие моменты времени корректируются значения ($a[3, -5]$, $b[3, 4]$), ($a[4, 2]$, $b[4, 5]$) и ($a[5, 3]$, $b[5, -4]$), которые выделены в табл. 16.1 жирным шрифтом и серым фоном. Новый вариант оценок приведен в табл. 16.2.

Таблица 16.2

	-5	-4	-3	-2	-1	0	1	2	3	4	5
$a[0]$	5	4	3	2	1	0	1	2	3	4	5
$a[1]$	10	9	8	7	6	5	6	7	8	9	10
$a[2]$	11	10	9	8	7	6	7	8	9	10	9
$a[3]$	10	11	10	9	8	7	8	9	10	11	10
$a[4]$	19	20	19	18	17	16	17	16	19	20	19
$a[5]$	25	26	25	24	23	22	23	22	11	26	25
$b[0]$	5	4	3	2	1	0	1	2	3	4	5
$b[1]$	10	9	8	7	6	5	6	7	8	9	10
$b[2]$	20	19	18	17	16	15	16	17	18	19	10
$b[3]$	27	26	25	24	23	22	23	24	25	8	17
$b[4]$	28	27	26	25	24	23	24	25	26	9	10
$b[5]$	33	18	31	30	29	28	29	30	31	14	15

Методическое отступление. Данный момент обсуждения — *ключевой* в понимании задачи! Желательно, чтобы для оставшихся трех значений были сделаны рисунки (по типу рис. 16.3 и 16.4), и понимание было закреплено на других исходных данных. Например:

5
3 2
1 4
-5 6
0 3
7 5

Результат «ручного» просчета логики приведен в табл. 16.3. Ответом в этом примере является значение 7.

Таблица 16.3

	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
$a[0]$	7	6	5	4	3	2	1	0	1	2	3	4	5	6	7
$a[1]$	10	9	8	7	6	5	4	3	4	5	6	7	8	9	10
$a[2]$	12	11	10	9	8	7	6	5	6	3	8	9	10	11	12
$a[3]$	18	17	16	15	14	13	12	11	12	9	14	9	16	17	18
$a[4]$	23	22	21	20	19	18	17	16	17	14	19	14	21	6	23
$a[5]$	30	29	28	27	26	25	24	23	24	21	16	21	28	13	30
$b[0]$	7	6	5	4	3	2	1	0	1	2	3	4	5	6	7
$b[1]$	9	8	7	6	5	4	3	2	3	4	5	6	7	8	9
$b[2]$	11	10	9	8	7	6	5	4	5	6	7	8	9	10	11
$b[3]$	13	12	11	10	9	8	7	6	7	8	9	10	11	12	13
$b[4]$	16	15	10	13	12	11	10	9	10	11	12	13	14	15	16
$b[5]$	18	17	16	15	14	13	12	7	12	13	14	15	16	17	18

Для реализации рассуждений о встречах Гермесов в приведенную выше логику на место угловых скобок следует записать операторы:

$$a[i+1, y[i]] := \min(a[i+1, y[i]], b[i, j] + \text{Abs}(j - x[i+1]));$$

$$b[i+1, x[i]] := \min(b[i+1, x[i]], a[i, j] + \text{Abs}(j - y[i+1])).$$

Ответ задачи — минимальная суммарная длина пути S , — находится как минимальное значение в последних строках матриц a и b .

Временная сложность алгоритма $O(n \cdot XMax)$, что позволяет надеяться на прохождение всех тестов.

Методические рекомендации

Обсуждение этой задачи лучше всего осуществлять с использованием проекционного оборудования. Слайды с рисунками и таблицами используются как для иллюстрации материала, так и для проверки самостоятельной деятельности школьника.

Последняя — обязательна! Все это, плюс «экспромты» преподавателя с использованием обычной доски, позволяет создать полноценное занятие на материале данной задачи.

Самостоятельные исследования

«Расслоить» задачу, сделать ее аддитивной или, другими словами, на динамическое программирование, позволяет условие формулировки: *«сообщения должны быть посланы богам строго в заданном порядке, поэтому Гермесу даны координаты кафетериев именно в этом порядке»*. А если убрать этот строго заданный порядок? Гермес тогда должен выбрать путь с минимальной суммарной длиной, а сообщения богам могут посылаться в произвольном порядке, но каждому богу — обязательно.

Кроме того, пусть сообщение Гермеса, посланное по вертикали или горизонтали, видят и воспринимают все боги, находящиеся в кафетериях, расположенных на этой вертикали или горизонтали. Упрощает или усложняет задачу данное условие?

«Артемида»

Условие задачи

Зевс выделил Артемиде, богине дикой природы, прямоугольную область для выращивания леса. Левая сторона области соответствует отрезку положительной части оси OY , нижняя сторона соответствует отрезку положительной части оси OX , а точка $(0, 0)$ — левому нижнему углу области. Зевс потребовал, чтобы Артемида выращивала деревья только в тех точках области, которые имеют целые координаты. Артемиде нравилось, когда лес выглядит естественно, поэтому она сажала деревья таким образом, чтобы отрезок, соединяющий *любую пару деревьев*, не был параллелен осям OX и OY .

Однажды Зевс захотел, чтобы Артемида срубила для него деревья, следуя правилам:

- Зевс требует срубить не менее t деревьев;
- чтобы получить прямоугольную футбольную площадку для будущих футбольных побед, Артемида должна срубить все деревья внутри некоторой прямоугольной области и ни одного дерева вне области;
- стороны этой прямоугольной области должны быть параллельны осям OX и OY ;
- два противоположных угла области должны находиться в местах расположения деревьев; соответственно, эти деревья также должны быть срублены.

Так как Артемида любит деревья, она хочет выполнить условия, срубив при этом как можно меньше деревьев. Вы должны написать программу, которая по информации о расположении деревьев в лесу и о минимальном количестве деревьев t , которые необходимо срубить, выбирает область вырубki деревьев для Артемиды.

Входные данные. Во входном файле с именем `artemis.in` первая строка содержит одно целое число n — количество деревьев в лесу. Вторая строка содержит одно целое число t — минимальное количество деревьев для вырубki. Последующие n строк описывают положения n деревьев. Каждая из этих строк содержит два целых числа: X и Y — x -координату и затем y -координату дерева.

Выходные данные. Выходной файл с именем `artemis.out` должен содержать одну строку с двумя целыми числами i и j , разделенными одним пробелом. Артемида должна рассматривать i -е дерево с координатами, заданными в строке $i+2$ входного файла, и j -е дерево с координатами, заданными в строке $j+2$ входного файла, как углы области для вырубki деревьев. Порядок вывода этих двух чисел не существен. Если существует несколько вариантов выбора такой пары деревьев, вы должны найти и выдать только один из них. Для всех тестов существует хотя бы одно решение.

Пример.

```
artemis.in:
3
2
1 1
2 3
5 6

artemis.out:
1 2
```

Ограничения: во всех тестах $1 < n \leq 20000$, $0 \leq X, Y \leq 64000$ и $1 < t \leq n$. В 50% тестов $1 < n < 5000$.

Обсуждение задачи

Первое, и очевидное, решение заключается в просмотре всех возможных прямоугольников и подсчете количества деревьев в каждом прямоугольнике. Его временная сложность — $O(n^3)$, и очевидно, что при заданных временных ограничениях (1 с на тест) данное решение не набирает полного балла. Компьютеров с быстродействием $8 \cdot 10^{12}$ в общедоступном пользовании еще нет.

Второй вариант решения и его различные модификации основаны на подсчете количества деревьев в прямоугольниках с углами $(0, 0)$, (x, y) . Количество деревьев t в прямоугольнике с углами (x_i, y_i) и (x_j, y_j) находится по формуле $t(x_j, y_j) - t(x_i, y_i) - t(x_j, y_i) + t(x_i, y_j) + 1$ (рис. 16.5). Используется следующий факт из формулировки задачи: «отрезок, соединяющий любую пару деревьев, не параллелен осям OX и OY ». Временная сложность алгоритмов — $O(n^2)$, что позволяет надеяться на прохождение большего количества тестов, но для получения полного балла, вероятно, требуется уменьшить константу в оценке временной сложности ($O(n^2)$ — это $c \cdot n^2$, где c — некоторая константа; речь идет именно о ней).

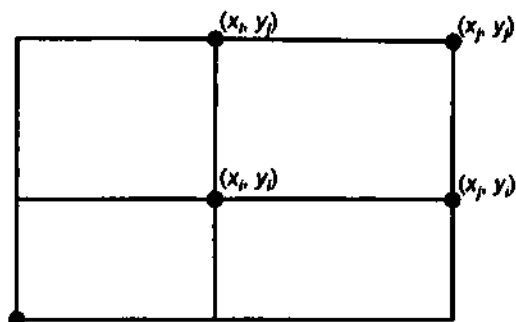


Рис. 16.5. Подсчет количества деревьев в прямоугольнике с углами (x_i, y_i) и (x_j, y_j)

Методическое отступление. Желательно в этот момент обсуждения для произвольных входных данных прорисовать расположение деревьев и найти те, которые подлежат вырубке. Как минимум, следует придумать один вариант размещения, для которого способа вырубки деревьев не существует.

Третий вариант решения. «Игра» идет на уровне квадратичной временной сложности алгоритма, поэтому рассмотрим пример, прямым образом, казалось бы, не связанный с задачей.

Пусть дан фрагмент логики:

```
For i:=2 To n Do
  For j:=1 To i-1 Do
    <действие>;
```

Сколько раз выполнится данное действие? Получаем: $1 + 2 + 3 + \dots + (n-1) = n(n-1)/2$ — временная сложность $O(n^2/2)$.

Предположим, что эквивалентный результат достигается по логике:

```
For i:=2 To n Div 2 Do
  For j:=1 To i-1 Do
    <действие>;
```

```

For i:=n Div 2 +1 To n-1 Do
  For j:=i+1 To n Do
    <действие>;

```

Подсчитаем количество выполнений для первого вложенного цикла: $1 + 2 + \dots + n \text{ Div } 2 - 1$. Сумма равна $n^2/8$. Аналогично — для второго цикла. Итого: $n^2/4$, что примерно в два раза быстрее, чем в первом случае!

Второе предварительное замечание: очевидно, что сортировка точек («деревьев» — в терминологии задачи) по одной из координат необходима. Пусть это будет сортировка по X . Алгоритмы с квадратичной временной сложностью ($O(n^2)$) явно отпадают; с логарифмической сложностью ($O(n \log_2 n)$) — под вопросом. Ограничения в задаче на объем разрешенной памяти (16 Мбайт) позволяют использовать алгоритм с линейной временной сложностью ($O(n)$), а именно — «сортировку подсчетом»*.

«Привяжем» для наглядности наше обсуждение задачи к конкретным входным данным. Они приведены на рис. 16.6; их графическое изображение — на рис. 16.7. Координаты точек («деревьев») хранятся в двух одномерных массивах (x, y), и их мы не трогаем при сортировке.

Предположим, что точки отсортированы по значению X . Номера точек хранятся в массиве qx . Для приведенного примера он имеет вид: $qx = (3, 7, 1, 8, 5, 2, 6, 4)$.

Зафиксируем некоторую точку, пусть это будет точка с номером 3. Зададим себе вопрос — какую информацию следует «связать» с каждой точкой, чтобы при их просмотре в порядке

8
3
4 7
10 11
2 6
14 12
6 5
12 1
3 8
5 2

Рис. 16.6. Пример входных данных

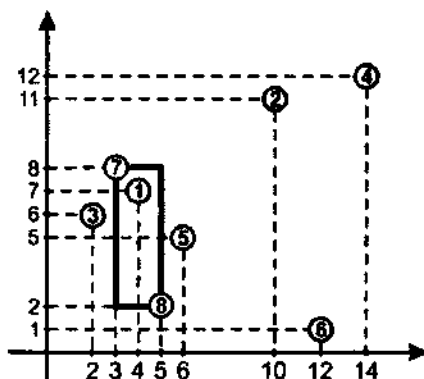


Рис. 16.7. Пример входных данных. Требуется вырубить три дерева

* Окулов С. М. Основы программирования. М.: БИНОМ. Лаборатория знаний, 2004. С. 249–251.

возрастания мы сразу могли сказать, образует ли данная точка с зафиксированной точкой (точкой отсчета) требуемый прямоугольник? Причем это искомое должно подсчитываться (формироваться) за квадратичное время с небольшим значением константы.

Методическое отступление. Это самый сложный момент в обсуждении задачи! Пауза и дискуссия просто необходимы! Как только требуемое будет «сгенерировано» в результате «мозгового штурма», дальнейшее становится лишь технической задачей.

Предположим, что для каждой точки j мы знаем, сколько точек находится слева и выше от нее ($wu[j]$) и сколько точек находится слева и ниже от нее ($wd[j]$). Для рассматриваемого примера $wu = (1, 0, 0, 0, 3, 6, 0, 3)$, а $wd = (1, 5, 0, 7, 1, 0, 1, 0)$. Мы берем точку i (точку отсчета, ее фактический номер $qx[i]$) и затем, просматривая дальше по координате x , подсчитываем, сколько точек находится выше ее (значение переменной u) и сколько точек расположено ниже ее (значение переменной d). Пусть очередной точкой является точка j (ее фактический номер — $qx[j]$) и она расположена ниже точки отсчета, тогда в прямоугольнике с углами $qx[i]$ и $qx[j]$ находится $d - wd[qx[j]] + 1$ точек (rez); если же точка выше, то в прямоугольнике $u - wu[qx[j]] + 1$ точек (rez). При совпадении значения rez с t (количеством деревьев, подлежащих рубке) задача решена.

Назовем совокупность данных $\langle u, d, rez \rangle$ *кортежем*. На рис. 16.8 показан процесс поиска прямоугольника. В кружках приводится номер точки, а рядом — текущее значение кортежа. Видим, что при точке отсчета 3 требуемый прямоугольник не найден. После просмотра этой ветви логики массивы wu и wd имеют вид: $wu = (1, 0, 0, 0, 2, 5, 0, 2)$; $wd = (0, 4, 0, 6, 1, 0, 0, 0)$. Берем в качестве точки отсчета следующее значение 7 и пытаемся построить прямоугольник с данной вершиной. Точка 8 может быть взята в качестве второй вершины прямоугольника, ибо в него попадает три точки (дерева) — обработка закончена.

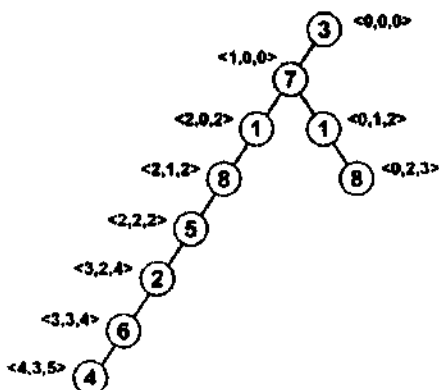


Рис. 16.8. Процесс поиска прямоугольника с заданными характеристиками

Методическое отступление. Целесообразно, на наш взгляд, в данный момент заняться на некоторое время практической работой. Школьник определяет произвольный набор точек и значение t . Формируются массивы qx , wu , wd , а затем осуществляется поиск прямоугольника так, как показано на рис. 16.8. Для облегчения работы можно демонстрировать слайд с фрагментом логики:

```

...
For i := 1 To n-t+1 Do Begin {Точка отсчета}
  p := qx[i];
  u := 0; {Количество точек, расположенных выше
точки отсчета}
  d := 0; {Количество точек, расположенных ниже
точки отсчета}
  For j := i+1 To n Do Begin
    q := qx[j];
    If y[p] > y[q] Then Begin {Сравнение координаты
у точек}
      d:=d+1;
      rez := d - wd[q] + 1;
      wu[q]:=wu[q]-1; {Не забываем уменьшать значение,
так как точка отсчета используется один раз}
    End
    Else Begin
      u:=u+1;
      rez := u - wu[q] + 1;
      wd[q]:=wd[q]-1; {Не забываем уменьшать значение,
так как точка отсчета используется один раз}
    End;
    If rez=t Then Begin
      WriteLn(p, q);
      Halt(0);
    End;
  End;
End;
...

```

В нашем обсуждении остался единственный еще не раскрытый момент — формирование значений элементов массивов wu и wd .

При вводе данных мы формируем не только массивы x и y — значения координат точек, но и массивы $sx[i]$, $cy[i]$ ($i = 1..64000$), в которых i — значение координаты, а $sx[i]$, $cy[i]$ — номер точки. Тогда при сортировке подсчетом (линейная временная зависимость) по координате X у нас появляется не только возможность формирования массива qx (номеров точек, отсортированных по X),

но и массивов px и py . Значение $px[i]$ указывает положение точки i по возрастанию координаты X , а $py[i]$ — положение точки i по возрастанию координаты Y . Для рассматриваемого в обсуждении задачи примера они равны: $px = (3, 6, 1, 8, 5, 7, 2, 4)$; $py = (5, 7, 4, 8, 3, 1, 6, 2)$. Так, точка с номером «один» является третьей в соответствии с упорядочиванием по X и пятой — по Y . Эти данные являются исходными при формировании массивов wu и wd .

Очевидно, что решение данной части задачи требует времени $O(n^2)$. Воспользуемся приемом уменьшения значения константы, данным в начале обсуждения.

Цель обсуждения достигается легче при «привязке» произносимых слов к конкретному наброску кода логики — он может быть и не окончательным (один из методических приемов учителя):

Procedure CountWeights;

Var i, j, k : **Integer**;

Begin

For $i:=2$ **To** $n \text{ Div } 2$ **Do Begin** {Идем по первой половине точек}

$k:=qx[i]$;

$wu[k]:=0$;

For $j:=1$ **To** $i-1$ **Do**

If $y[k] < y[qx[j]]$ **Then** $wu[k]:=wu[k]+1$;

 {Считаем количество точек слева и выше относительно точки k }

$wd[k]:=px[k] - 1 - wu[k]$; {Тогда количество точек, расположенных слева и ниже, равно общему количеству $(px[k] - 1)$ минус количество точек слева и выше относительно точки k }

End;

For $i:=n \text{ Div } 2 + 1$ **To** $n-1$ **Do Begin** {Идем по второй половине точек}

$k:=qx[i]$;

$wu[k] := 0$;

For $j:=i+1$ **To** n **Do**

If $y[k] < y[qx[j]]$ **Then** $wu[k]:=wu[k]+1$;

 {Подсчитываем количество точек справа и выше относительно точки k }

$wu[k]:=n - py[k] - wu[k]$; {Тогда количество точек слева и выше равно общему количеству точек минус справа и выше, а также количеству точек, находящихся ниже точки k . У нас "заработала" упорядоченность точек по оси Y . Значение $py[k]$ }

указывает на положение точки k по возрастанию координаты Y . Другими словами, ниже ее находится $py[k]$ точек)

$wd[k] := px[k] - 1 - wu[k];$

End;

$k := qx[n];$ (Вычисляем характеристики последней точки)

$wu[k] := n - py[k];$

$wd[k] := py[k] - 1;$

End;

Как мы видим, разбивка множества точек на две половины и вычисление значений wu , wd по отдельности уменьшают константу при квадратичной временной сложности в два раза.

Методические рекомендации

Эта задача, при всей очевидности ее формулировки, не относится (при заданных ограничениях на время решения) к разряду простых. Из алгоритмической базы требуется знание алгоритмов сортировки, и только. В решении интенсивно используется фундаментальный принцип информатики — «*принцип косвенной адресации*», когда содержимое ячейки является не конкретными данными, а адресом данных в другой структуре.

Для организации соревновательного момента при практической реализации задачи требуется наличие тестов и проверяющей программы.

Самостоятельные исследования

1. Измените решение для поиска всех прямоугольников с заданным количеством точек. Найдите прямоугольник с наименьшей площадью.
2. Для заданного значения t получите (сгенерируйте) координаты n точек, при которых задача не имеет решения (если это возможно).

«Многоугольник»

Условие задачи

Точками многоугольника являются как его внутренние точки, так и точки на границе. Выпуклым многоугольником называется такой многоугольник, в котором для любых двух его точек X и Y отрезок, соединяющий X и Y , полностью принадлежит многоугольнику. Все многоугольники в этой задаче выпуклые и содержат не менее 2 вершин. Все вершины любого многоугольника в этой задаче различны и имеют целочисленные координаты. Никакие три вершины не лежат на одной прямой. Слово

«многоугольник» далее всегда обозначает именно такие многоугольники.

Для двух данных многоугольников A и B их суммой Минковского называется фигура, состоящая из всех точек вида $(x_1 + x_2, y_1 + y_2)$, где (x_1, y_1) — все возможные точки многоугольника A и (x_2, y_2) — все возможные точки многоугольника B . Оказывается, что сумма Минковского двух многоугольников также является многоугольником. Рис. 16.9 иллюстрирует сумму Минковского для двух треугольников.

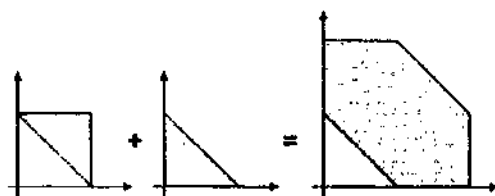


Рис. 16.9. Иллюстрация операции «сумма Минковского»

Будем рассматривать операцию, обратную сумме Минковского. Для данного многоугольника P требуется найти такие два многоугольника A и B , что:

- P является суммой Минковского A и B ;
- A имеет от 2 до 4 различных вершин, то есть это отрезок (2 вершины), треугольник (3 вершины) или четырехугольник (4 вершины);
- A должен иметь как можно больше вершин, то есть:
 - A должен быть четырехугольником, если это возможно,
 - если A не может быть четырехугольником, он должен быть треугольником, если это возможно,
 - иначе он должен быть отрезком.

Очевидно, что ни A и ни B не может быть равен P , поскольку в этом случае один из многоугольников, A или B , должен быть точкой, что не является многоугольником по определению.

Вам дан набор входных файлов, каждый из которых содержит описание многоугольника P . Для каждого входного файла вы должны найти многоугольники A и B , такие, как описано выше, и создать выходной файл, содержащий описания A и B . Гарантируется, что для каждого из данных входных файлов такие многоугольники существуют. Если существует несколько вариантов ответа, запишите любой из них. Вы не должны сдавать какие-либо программы, вы должны сдать на проверку только выходные файлы.

Входные данные. Вам дан набор из 10 входных файлов с названиями `polygon1.in`, `polygon2.in`, ..., `polygon10.in`, где число после слова `polygon` является номером теста. Каждый входной файл имеет следующий формат. Первая строка содержит одно целое число n — количество вершин многоугольника P .

Следующие n строк описывают вершины многоугольника в порядке их обхода против часовой стрелки, по одной вершине в строке. Строка $i+1$ ($i = 1, 2, \dots, n$) содержит два целых числа X_i и Y_i , разделенных пробелом: координаты i -й вершины многоугольника. Все координаты во входных файлах — неотрицательные целые числа.

Выходные данные. Вы должны сдать 10 выходных файлов, соответствующих данным входным файлам с описаниями искомым многоугольников A и B . Первая строка каждого выходного файла должна содержать текст: `#FILE polygon i`, где целое число i ($1 \leq i \leq 10$) является номером соответствующего входного файла.

Далее формат выходного файла подобен формату входного файла. Вторая строка должна содержать одно целое число n_A — количество вершин многоугольника A ($2 \leq n_A \leq 4$). Следующие n_A строк должны описывать вершины многоугольника A в порядке обхода против часовой стрелки, по одной вершине в строке. Строки с номерами $i+2$ ($i = 1, 2, \dots, n_A$) должны содержать по два целых числа X и Y — координаты i -й вершины многоугольника A . Числа в строке должны разделяться одним пробелом.

Строка n_A+3 должна содержать одно целое число n_B — количество вершин многоугольника B ($2 \leq n_B$). Следующие n_B строк должны описывать вершины многоугольника B в порядке обхода против часовой стрелки, по одной вершине в строке. Строки с номерами n_A+j+3 ($j = 1, 2, \dots, n_B$) должны содержать по два целых числа X и Y — координаты j -й вершины многоугольника B . Числа в строке должны разделяться одним пробелом.

Пример.

```
polygon0.in:
```

```
5
0 1
0 0
2 0
2 1
1 2
```

```
#FILE polygon 0
```

```
3
0 0
2 0
1 1
2
0 1
0 0
```

```
#FILE polygon 0
3
0 0
1 0
1 1
3
0 1
0 0
1 0
```

Для приведенного входного файла любой из двух приведенных выходных файлов корректен, так как в обоих случаях A является треугольником и не может быть четырехугольником. На рис. 16.10 приведены эти случаи.

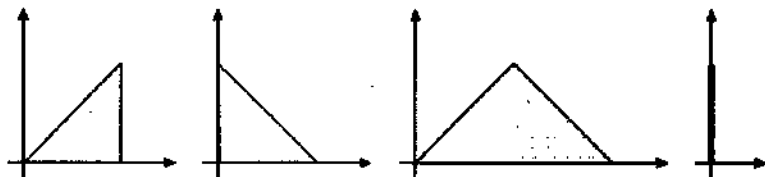


Рис. 16.10. Иллюстрация решения примера из формулировки задачи

Обсуждение задачи*

Решение задачи «опирается» на знание следующего математического факта. Периметр P многоугольника D , являющегося суммой Минковского многоугольников A и B , равен сумме периметров A и B : $P(D) = P(A) + P(B)$. Это следует из того, что сумма векторов, образующих многоугольник D при его обходе, например, против часовой стрелки, равна нулю. Это утверждение справедливо и для D , и для A , и для B (отрезок рассматривается как два противоположных вектора) и проиллюстрировано на рис. 16.11.

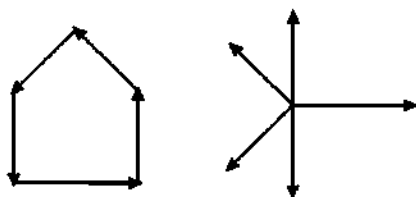


Рис. 16.11. Иллюстрация идеи решения

Если бы векторы сторон исходного многоугольника использовались как целое, их нельзя было бы «дробить» (делить на части), то в данном множестве векторов следовало бы искать

* Авторы благодарят А. Н. Семенова за помощь в разборе задачи.

подмножество из четырех векторов (A — четырехугольник), в сумме дающих ноль (аналогично — для треугольника).

И вот тут становится понятно, почему требуется сдать только выходные файлы! Необходимо «дробление», многоугольники A и B могут «складываться» из частей векторов. Естественно, что одна часть входит только в одну сумму. Но в этом случае время поиска требуемого подмножества векторов становится значительным. Просмотр входных файлов показывает, что количество вершин исходного многоугольника превосходит значение 100, а значения координат — хотя и целые числа, но превышают несколько тысяч. И жюри не интересовало, как это сделал участник, ибо, если выходные файлы получены, то школьник использовал не прямой перебор, а значит, и в информатике он не новичок.

Обратимся к решению. Основная идея заключается в том, чтобы по исходному многоугольнику (координаты вершин хранятся в массиве *Poly* типа *TPolygon=Array [1..NMax] of TCord;*, где *TCord = Record x,y:Integer; End;* массив дополняется $n+1$ элементом, равным *Poly[1]*) создать некую структуру данных, позволяющую позже организовать эффективный поиск подмножества векторов, образующих A . Для данной задачи единственным, что «приходит на ум», является *хеширование*, создание *хеш-таблиц*. Все остальные структуры требуют значительных усилий для их поддержания и не дают нужного эффекта при поиске.

Учитывая эффект «дробления», хеш-таблицу следует формировать на основе «базовых» векторов. Покажем, как они определяются (массив *Prim*):

```

For i:=1 To n Do Begin
  With Vect[i] Do Begin {Определяем координаты
    вектора, соответствующего стороне многоугольника}
    x:=Poly[i+1].x-Poly[i].x;
    y:=Poly[i+1].y-Poly[i].y;
  End;
  Count[i]:=Nod(Abs(Vect[i].x),Abs(Vect[i].y));
  {Вычисляется наибольший общий делитель. Функция
  имеет стандартный вид и здесь не приводится}
  With Prim[i] Do Begin {Вычисляем координаты
    "базового" вектора}
    x:=Vect[i].x Div Count[i];
    y:=Vect[i].y Div Count[i];
  End;
End;

```

После этого с помощью следующего фрагмента логики формируется хеш-таблица:

```

For i:=1 To n-1 Do
  For j:=i+1 To n Do
    For mi:=1 To Count[i] Do
      For mj:=1 To Count[j] Do
        AddHashTable(i, j, mi, mj);

```

Процедура AddHashTable требует пояснений. Есть еще одна (во всяком случае, видимая на данной стадии обсуждения) возможность уменьшения времени поиска векторов, определяющих многоугольник A . Следует хранить данные не по отдельным векторам, а по их суммам (рис. 16.12). Что это дает? Пусть ищется четырехугольник A и хранятся параметры векторов сумм. Оказывается, что в этом случае можно не искать четыре вектора, образующих A , а определить, есть ли в хеш-таблице вектор такой же длины, но в противоположном направлении и образованный из других векторов. Если так, то A найден; точнее, найдены два вектора, сумма которых равна нулю. А затем по этим векторам находят те, которые дают их в сумме.

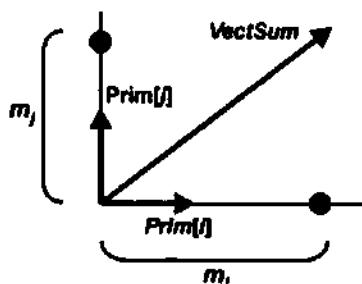


Рис. 16.12. Иллюстрация принципа организации хеш-таблицы

```

Procedure AddHashTable(i, j, mi, mj: Integer);
Var VectSum: TCord;
    HashCode: Integer;
Begin
  With VectSum Do Begin {Вычисляем вектор суммы
    двух векторов. Их номера - i и j. Первый вектор
    берется mi раз, второй - mj (рис. 16.12)}
    x:=Prim[i].x*mi+Prim[j].x*mj;
    y:=Prim[i].y*mi+Prim[j].y*mj
  End;
  {Вычисляем адрес в хеш-таблице (Hash) по значению
  координат вектора суммы}
  HashCode:=GetHashCode(VectSum.x, VectSum.y);
  {Функция GetHashCode не приводится. Ее вид может
  быть различным. Например, значения координат
  умножаются на определенные ранее случайные числа,

```

находится сумма произведений по модулю размера хеш-таблицы}

{Разрешение коллизий осуществляется стандартным для методов хеширования образом, например путем организации списка для каждой совокупности данных, дающих одно значение хеш-адреса}

<Запоминаем в хеш-таблице значения VectSum.x, VectSum.y, i, j, mi, mj — это данные информационного поля элемента хеш-таблицы>;

End;

Итак, хеш-таблица организована. Осталось выполнить поиск — последовательно, от четырехугольника до отрезка. Разумно, в случае успеха на каком-то шаге, после вывода результата в процедуре Done заканчивать обработку. Детали поиска мы «скрываем» в процедурах FindReverse4, FindReverse3 и FindReverse2, что позволяет сделать общую логику достаточно «прозрачной»:

```
For i:=1 To n-1 Do
  For j:=i+1 To n Do
    For mi:=1 To Count[i] Do
      For mj:=1 To Count[j] Do
        FindReverse4(i,j,mi,mj); {A — четырехугольник?}
      For i:=1 To n Do
        For mi:=1 To Count[i] Do FindReverse3(i,mi);
      {A — треугольник?}
    FindReverse2; {A является отрезком}
```

Процедуры FindReverse имеют типовую структуру, ибо их назначение заключается в выборке (поиске) из хеш-таблицы записи, соответствующей искомому вектору, и в формировании массива векторов, образующих многоугольник A. Разберем эту структуру на примере одной из процедур, а затем укажем отличия между ними:

```
Procedure FindReverse4(i,j,mi,mj:Integer);
  Var VectSum:TCord;
      HashCode:Integer;
  Begin
    With VectSum Do Begin {Формируем противоположный вектор}
      x:=-Prim[i].x*mi-Prim[j].x*mj;
      y:=-Prim[i].y*mi-Prim[j].y*mj
    End;
```

HashCode:=GetHashCode (VectSum.x, VectSum.y);
{Определяем адрес в хеш-таблице. Это, естественно, та же функция, что использовалась при формировании хеш-таблицы}

<Просматриваем записи хеш-таблицы, соответствующие одному адресу>; {Для реализации действия необходимо использовать конструкцию повторения (вспомним о коллизиях)}

If <вектора сумм противоположны и образованы разными "базовыми" векторами>

Then Begin

<преобразовать вектор суммы в массив исходных векторов, по которым они получены>; {Если первым элементом массива сделать счетчик количества ненулевых элементов в нем, то процедура Done может быть унифицирована}

<вызвать процедуру Done с этими данными для вывода результата и завершения обработки>;

End;

End;

Процедура FindReverse3 имеет только два входных параметра — номер «базового» вектора (i) и значение mi (сколько раз он берется). Ее отличие сводится к измененному фрагменту:

With VectSum Do Begin

x:=-Prim[i].x*mi;

y:=-Prim[i].y*mi

End;

а также несколько измененным фрагментам проверки противоположности векторов и формирования массива координат векторов для процедуры Done.

Процедура FindReverse2 не имеет параметров, но содержит фрагмент:

With VectSum Do Begin

x:=0;

y:=0;

End;

HashCode:=0;

что позволяет неким единообразным способом сформировать массив для процедуры Done.

Осталась еще не выясненной логика процедуры Done. Ее функциональное назначение сводится к исключению из множества векторов, описывающих исходный многоугольник (массив *Poly*), найденного множества векторов, определяющих многоугольник *A*, и выводу в результирующий файл координат многоугольников, построенных по вычисленным множествам векторов. Естественно, что суммирование по Минковскому выполнять не требуется.

Методические указания

Эта задача требует знания базовых алгоритмов вычислительной геометрии, а математические факты могут быть сообщены в процессе обсуждения. Задачу следует использовать при изучении темы «Поиск», а именно — методов хеширования, тем более что содержательных задач, связанных с этими интересными и фундаментальными методами, встречалось немного, и не только в олимпиадной проблематике, но и в учебной литературе.

Самостоятельные исследования

1. Качественная реализация решения и проверка его на входных файлах позволит убедиться, что время получения (для всех тестов) каждого выходного файла не превысит 30–40 секунд.
2. Незначительные изменения в формулировке задачи приводят к существенным ее смысловым изменениям. Достаточно задать, что *A* имеет от 2 до 5 различных вершин, или что многоугольники *A* и *B* должны быть наиболее близкими к друг другу по количеству вершин, и предложенная схема эффективного поиска «рассыпается».

ЗАДАЧИ ВТОРОГО ТУРА

«Фидий»

Условие задачи

Известный древнегреческий скульптор Фидий готовится к постройке очередного выдающегося монумента. Для этого ему требуются мраморные прямоугольные плитки заданных размеров $w_1 \times h_1$, $w_2 \times h_2$, ..., $w_n \times h_n$.

Недавно Фидий получил большую прямоугольную мраморную плиту. Он хочет разрезать плиту так, чтобы получить плитки заданных размеров. Мраморная плита или плитка может быть разрезана только вертикальным или горизонтальным разрезом на две прямоугольные плитки. Разрез осуществляется от края до края. Это единственный способ разрезания плит и

плиток. Вырезанные плитки нельзя соединять никаким образом. Так как мрамор имеет рисунок, плитки нельзя вращать. То есть, если Фидий вырезал плитку размером $a \times b$, она не может быть использована в качестве плитки $b \times a$, кроме случая,

когда $a = b$. Он может вырезать 0 или более плиток каждого из требуемых размеров, и при этом могут получиться плитки других размеров, которые не будут использоваться. Фидий хочет знать, как нужно разрезать исходную плиту, чтобы суммарная площадь неиспользуемых плиток была минимальна.

Например, пусть ширина исходной плиты равна 21 и высота равна 11, а заданные размеры плиток: 10×4 , 6×2 , 7×5 и 15×10 . Тогда минимальная возможная площадь неиспользуемых плиток равна 10. Рис. 16.13 иллюстрирует один из способов разрезания исходной плиты.

Требуется написать программу, которая по заданному размеру исходной плиты и размерам требуемых плиток вычисляет минимальную суммарную площадь неиспользуемых плиток.

Входные данные. В первой строке входного файла `phidias.in` содержатся два целых числа. Первое число w — ширина исходной плиты, второе h — высота исходной плиты. Вторая строка содержит одно целое число n — количество требуемых размеров плиток. Следующие n строк содержат требуемые размеры плиток. Каждая из этих строк содержит по два целых числа: w_i — ширину и h_i — высоту ($1 \leq i \leq n$).

Выходные данные. Выходной файл `phidias.out` должен содержать одно целое число — минимальную суммарную площадь неиспользуемых плиток.

Пример.

```
phidias.in:
21 11
4
10 4
6 2
7 5
15 10
```

```
phidias.out:
10
```

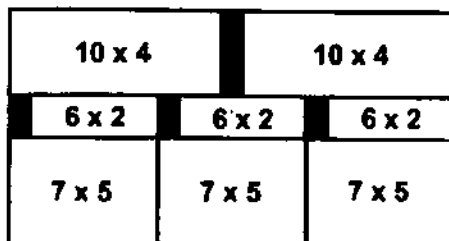


Рис. 16.13. Иллюстрация примера

Ограничения: во всех входных файлах $1 \leq w \leq 600$, $1 \leq h \leq 600$, $0 < n \leq 200$, $1 \leq w_i \leq w$ и $1 \leq h_i \leq h$. В 50% тестов: $w \leq 20$, $h \leq 20$ и $n \leq 5$.

Обсуждение задачи

Рассмотрим крайние случаи.

Из плиты нельзя вырезать ни одной плитки: ответ — $w \times h$.

Из плиты можно вырезать в точности одну плитку: ответ — 0.

Если требуемый размер плиток — 1×1 , то ответ также равен нулю.

Пусть размер плиты равен $w \times (w+1)$ и вырезается одна плитка $w \times w$. Ответ — w .

Продолжим. Размер требуемых плиток — $1 \times h$: ответ — 0, а если запретить горизонтальные размеры, то ответ — $w \times h$.

Вырезаемые плитки $w \times 1$: ответ — 0; запрещая вертикальные разрезы плиты, получаем $w \times h$.

Явно просматривается необходимость введения матрицы $d[1..w, 1..h]$, элемент $d[i, j]$ которой равен величине остатка (его мы и минимизируем) при вырезании плиток из плиты размером $i \times j$. Явно просматривается динамическая схема формирования d . Можно было бы выписать рекуррентные соотношения и пояснить их, тем более что они просматриваются; мы же вместо этого сделаем методическое отступление*.

Методическое отступление. Предлагается «поиграть», — а именно, по простым исходным данным «вручную» заполнять матрицу d . Образцы такой «игры» приведены на рис. 16.14. Слева на рис. 16.14 приводится файл `phidias.in`, справа — матрица d . После этой работы можно вернуться к написанию формальных рекуррентных соотношений.

Первоначально элементам матрицы $d[i, j]$ присваивается значение $i \times j$, а затем при вводе размеров плиток w_i, h_i , где

* М. К. Мамардашвили говорил: «Одно умное слово, другое, логически безупречные построения, но нет некоего невербального состояния внутреннего слова, при котором все проверялось, невербального корня. Переход на этот корень — очень жесткий шаг».

Наша задача не в том, чтобы написать и показать, а в том, чтобы возникло то состояние у школьника, в котором он сам напишет и покажет. Школьник сам обязан захотеть и сделать этот жесткий шаг перехода на невербальный корень, а учитель создает среду (обстановку), которая способствует возникновению желания хотеть, пойти на эту жесткость, ибо она требует напряжения умственных сил и возможностей. А это для любого человека не просто, и особенно для школьника. Это качество души — ежедневной жесткости по отношению к самой себе, воспитывается только путем ежедневных усилий и тренировок. Состояние «по моему хотению, по моему велению» возникает только у тренированной души; сказки врут — «лежа на печи» его не достигнешь!

		1	2	3	4	5	6			1	2	3	4	5	6			
5	6		2	4	6	8	10	12		5	6		2	4	6	8	10	12
1	→	3	6	9	12	15	18			1	→	3	6	9	12	15	18	
5	5		4	8	12	16	20	24		4	6		4	8	12	16	20	0
			5	10	15	20	0	⑤					5	10	15	20	25	6

		1	2	3	4	5	6			1	2	3	4	5	6			
5	6		2	4	0	2	4	0		5	6		2	0	2	0	2	0
1	→	3	6	3	6	9	6			1	→	3	2	5	4	7	6	
2	3		4	8	0	4	8	0		2	2		4	0	4	0	4	0
			5	10	3	8	13	⑥					5	2	7	4	9	6

		1	2	3	4	5	6			5	6		1	2	3	4	5	6
5	6		2	4	6	8	10	12		2			2	4	0	2	4	0
1	→	3	0	3	0	3	0			2	3	→	3	0	3	0	3	0
3	2		4	2	6	4	8	6		3	2		4	2	0	4	2	0
			5	4	9	8	13	⑫					5	4	3	2	7	0

Рис. 16.14. Образцы заполнения матрицы d

$1 \leq t \leq n$, элементам $d[w_t, h_t]$ присваивается значение «нуль». После этого осталось последовательно сформировать значения $d[i, j]$ следующим образом:

$d[i, j] := \text{Min}_{1 \leq t \leq i \text{ Div } 2} (d[i, j], d[t, j] + d[i - t, j])$ — учет горизонтальных разрезов;

$d[i, j] := \text{Min}_{1 \leq t \leq i \text{ Div } 2} (d[i, j], d[i, t] + d[i, j - t])$ — учет вертикальных разрезов плиты.

Ответ задачи — значение $d[w, h]$.

Временная сложность алгоритма — $O(w \times h \times \max(w, h))$. Вероятно, требуется очень тщательное программирование (например, исключение лишних вызовов процедур и функций), чтобы выполнить ограничения на время прохождения тестов.

Методические рекомендации

Это задача на тему «Динамическое программирование», и ее вряд ли можно отнести к разряду сложных. На целое занятие, особенно при подготовленном слайде с материалом рис. 16.14, ее недостаточно, но с дополнениями, вынесенными на самостоятельные исследования, получается вполне «полнокровное» мероприятие.

Самостоятельные исследования

1. Как «испортить» эту задачу? Требуется изменить некоторые условия в ее формулировке так, чтобы она перестала быть аддитивной. После этого решите задачу в новой редакции и определите ограничения на диапазон входных параметров, при которых решение работает разумное время.

2. *Покрытие несравнимыми прямоугольниками**. Два прямоугольника называются *несравнимыми*, если ни один из них не может быть размещен внутри другого, когда они расположены так, что соответственные стороны параллельны. Найдите покрытия различных прямоугольных областей наборами попарно несравнимых прямоугольников. Например, наименьшее возможное решение представляет собой прямоугольник размера 13×22 , покрытый так, как показано на рис. 16.15.

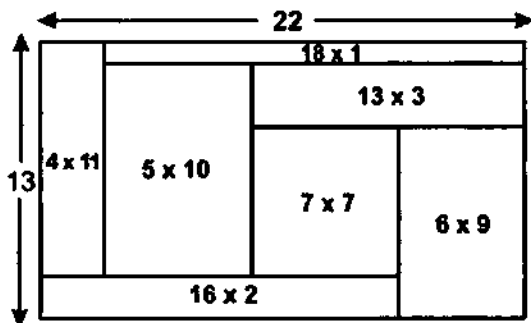


Рис. 16.15. Покрытие прямоугольника 13×22 несравнимыми прямоугольниками

Наименьший из известных квадратов, который можно покрыть несравнимыми прямоугольниками, имеет размер 27×27 . Существует ли меньший?

3. *Задача о паркете***. Комнату размером $w \times h$ единиц требуется покрыть одинаковыми плитками размером 2×1 единиц без пропусков и наложений ($1 \leq w \leq 8$, $1 \leq h \leq 20$; w, h — целые числа). Пол можно покрыть паркетом различными способами. Например, для $w = 2$, $h = 3$ все возможные способы укладки приведены на рис. 16.16.

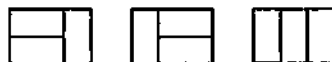


Рис. 16.16. Пример укладки паркета

* Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980. С. 175.

** Задача С. Г. Волченкова с VI Всероссийской олимпиады школьников по информатике (1994 год).

Требуется определить количество всех возможных способов укладки паркета для конкретных значений w и h . Результат задачи — таблица $d[1..8, 1..20]$. Элементом таблицы $d[i, j]$ является число, являющееся решением задачи для соответствующих значений i и j . На месте не найденных результатов должен стоять символ «*».

Комментарий. Это задача на динамическое программирование, но поиск схемы аддитивности не так прост, как в обсуждаемой ранее олимпиадной задаче*.

«Фермер»

Условие задачи

Фермер владеет несколькими полями, на границе каждого из которых растут кипарисовые деревья. У фермера также есть полосы земли, на каждой из которых растет ряд кипарисовых деревьев. На границах полей и на полосках между каждой парой подряд стоящих кипарисовых деревьев растет одно оливковое дерево. Все кипарисовые деревья фермера растут либо по границе поля, либо на полоске, и все оливковые деревья фермера находятся между двумя соседними кипарисовыми деревьями на границе поля или на полоске.

Однажды фермер сильно заболел и почувствовал, что скоро умрет. За несколько дней до смерти он позвал своего старшего сына и сказал ему: «Я завещаю тебе любые Q кипарисовых деревьев на твой выбор, а также все оливковые деревья, которые растут между любыми двумя подряд стоящими кипарисовыми деревьями, выбранными тобой». Из каждого поля и из каждой полоски сын может выбрать любую комбинацию деревьев. Так как старший сын любит оливки, он хочет выбрать такие Q кипарисовых деревьев, которые позволят ему унаследовать как можно большее количество оливковых деревьев.

Рассмотрим пример, когда сын получает для полей и полосок, изображенных на рис. 16.17, $Q = 17$ кипарисовых деревьев. Для того, чтобы максимизировать количество унаследованных оливковых деревьев, он должен выбрать все кипарисовые деревья на поле 1 и поле 2, получив, таким образом, 17 оливковых деревьев.

Требуется написать программу, которая по информации о полях, полосках и количестве выбираемых сыном кипарисовых деревьев определяет наибольшее возможное количество оливковых деревьев, которое сын может унаследовать.

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 119–121.

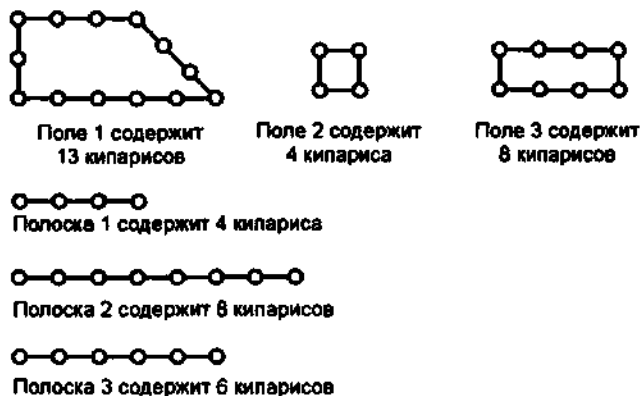


Рис. 16.17. Пример размещения кипарисовых деревьев (оливковые деревья не изображены)

Входные данные. Первая строка входного файла `farmer.in` содержит три целых числа: Q — количество кипарисовых деревьев, которое должен выбрать сын, затем M — количество полей и затем K — количество полосок. Вторая строка содержит M целых чисел $N_1, N_2, \dots, N_M$ — количества кипарисовых деревьев на полях. Третья строка содержит K целых чисел $R_1, R_2, \dots, R_K$ — количества кипарисовых деревьев в полосках.

Выходные данные. В единственной строке выходного файла `farmer.out` должно находиться одно целое число — наибольшее возможное количество оливковых деревьев, которое может унаследовать сын.

Пример.

`farmer.in:`

17 3 3

13 4 8

4 8 6

`farmer.out:`

17

Ограничения. Во всех тестах $0 \leq Q \leq 150000$, $0 \leq M \leq 2000$, $0 \leq K \leq 2000$, $3 \leq N_1 \leq 150$, $3 \leq N_2 \leq 150$, ..., $3 \leq N_M \leq 150$, $2 \leq R_1 \leq 150$, $2 \leq R_2 \leq 150$, ..., $2 \leq R_K \leq 150$. Общее количество кипарисовых деревьев на полях и в полосках больше или равно Q . В 50% тестов $Q \leq 1500$.

Обсуждение задачи

Попытаемся найти аналогии, возникающие при чтении формулировки задачи. Требуется набрать Q кипарисовых деревьев

из известных значений, определяемых количеством деревьев на границах полей и на полосках. Пусть у нас есть рюкзак, выдерживающий вес не более Q килограмм. Известны веса предметов, помещаемые в рюкзак. Требуется выбрать предметы так, чтобы их суммарный вес был максимально близок к значению Q , но не превышал его. Задача о рюкзаке относится к классике информатики, а ее целочисленный вариант — к учебным задачам по теме «динамическое программирование»^{*}.

Вернемся к задаче. Как видим, количество оливковых деревьев на границах полей совпадает с количеством кипарисовых деревьев, а для полосок оно на единицу меньше. Следовательно, выгоднее брать все кипарисовые деревья на границе конкретного поля, чем то же количество деревьев в полоске.

Пусть полосок нет, а есть только поля, и количество кипарисовых деревьев на границах полей равно k .

Если $k = Q$, то решение очевидно, ибо это уже и есть ответ.

Пусть $k > Q$. В этом случае наличие или отсутствие информации по полоскам во входных данных не имеет какого-либо значения — достаточно рассматривать только границы полей.

Пример в формулировке задачи очень простой с очевидным ответом, — усложним его. Дано пять полей, с количеством кипарисовых деревьев 3, 4, 5, 6, 7 (хранятся в массиве v), и разрешено взять 24 дерева ($Q = 24$). Значение $k = 25$. С одной стороны, ответ очевиден — мы берем все кипарисовые деревья на границах последних полей (получаем 22 оливковых дерева) и два дерева на границе первого поля (между ними одно оливковое дерево); количество оливковых деревьев в этом случае равно 23. С другой стороны — как представить этот очевидный результат в памяти компьютера? Это уже не очевидно! Но если мы поймем или ответим на вопрос «как», то сделаем и первый набросок решения.

Методическое отступление. В этот момент обсуждения целесообразно не только обсудить способы представления результата в памяти, но и выяснить, какой вариант набора конкретного количества кипарисовых деревьев «лучше» и вообще — что значит «лучше». Например, $11 = 5 + 6$ или $11 = 4 + 7$ в данном примере?

Величина Q не очень большая, поэтому можно ввести массив d , индексом i которого является количество кипарисовых деревьев (по аналогии — вес рюкзака). А что будет значением $d[i]$? Ответ: номер последнего поля, на границе которого кипарисо-

^{*} Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 101–102.

вые деревья задействованы в наборе значения i . Начальные значения — $d[0] = 0$, а все остальные элементы, например, равны минус единице. На первой итерации ($i = 0$) мы приравниваем: $d[3] = 1$, $d[4] = 2$, $d[5] = 3$, $d[6] = 4$, $d[7] = 5$. Смысл, в частности, присвоения $d[7] = 5$ состоит в том, что 7 кипарисовых деревьев мы набираем, взяв последней границу пятого поля. Находим очередной элемент массива d , не равный минус единице. В рассматриваемом примере это $i = 3$, т. е. три кипарисовых дерева. Затем мы определяем, какие суммарные значения количеств кипарисовых деревьев можно набрать при рассмотрении полей, начиная с $d[i]+1$. Видим, что изменяются значения 7, 8, 9 и 10 элементов массива d ($d[7] = 2$, $d[8] = 3$, $d[9] = 4$, $d[10] = 5$). Окончательный вариант просчета для данного примера представлен в табл. 16.4.

Таблица 16.4

i	0	1	2	3	4	5	6	7	8	9	10	11	12
$d[i]$	0	-1	-1	1	2	3	4	2	3	3	4	4	3
i	13	14	15	16	17	18	19	20	21	22	23	24	25
$d[i]$	4	4	4	5	5	4	5	5	5	5	-1	-1	-1

Мы видим, что использование границ полей (целиком — всех кипарисовых деревьев) приводит к сумме 22, но так как $k > Q$, то оставшиеся кипарисовые деревья можно взять с границы незадействованного поля (нас не интересует, какого). Кроме того, нами сделано больше, чем требовалось! Мы можем определить и поля, дающие сумму 22. Из 22 вычитаем количество кипарисовых деревьев на границе пятого поля, получаем 15. Из 15 вычитаем количество кипарисовых деревьев на границе четвертого поля, получаем 9, и т. д. Этот процесс выделен в табл. 16.4 «жирным» шрифтом. Значение 22 дают второе, третье, четвертое и пятое поля.

Методическое отступление.

1. Обсуждение вопроса о том, почему не требуется сортировка элементов массива v (количество кипарисовых деревьев на границах полей) по неубыванию, позволяет углубить понимание идеи решения задачи.
2. Показ фрагмента логики в нотации на Паскале (используется как «псевдокод» для записи алгоритма и выбора структур данных) и «ручная» его прокрутка для конкретного примера является, на наш взгляд, необходимой составляющей за-

нятия со школьниками, начинающими изучать данную проблематику.

Фрагмент логики имеет вид:

```

...
For i:=1 To Q Do d[i]:=-1;
d[0]:=0;
For i:=0 To Q-1 Do
  If d[i]<>-1 Then
    For j:=d[i]+1 To n Do
      If ((i+v[j])<=Q) And
        ((d[i+v[j]]>j)Or(d[i+v[j]]=-1)) Then
        d[i+v[j]]:=j;
i:=Q;
While d[i]=-1 Do i:=i-1;
rez:=i; {rez - ответ задачи}
If i<>Q Then rez:=rez+Q-i-1;
...

```

Осталось оценить временную сложность алгоритма для рассматриваемого случая — для каждого значения Q просматриваются все поля, а их n , т. е. имеем $O(Q \times n)$, что при заданных ограничениях позволяет рассчитывать на прохождение всех тестов.

Перейдем теперь к рассмотрению случая $k < Q$ — количество кипарисовых деревьев на границах полей меньше числа деревьев, завещанных фермером сыну. Решение очевидно, ибо требуется набрать $Q-k$ деревьев на полосках (все деревья на границах полей сын выбирает и наследует), и для этого случая мы применим вариант «жадной» логики. Пусть $w = Q-k$. Найдём полоску с количеством кипарисовых деревьев t , максимально близким к значению w , но меньшим значения w . Включаем ее в решение: количество оливковых деревьев стало равно $k+t-1$. Осталось набрать $Q-k-t$ деревьев. Продолжаем процесс поиска, — точнее, выбора следующей полоски. Время решения линейно зависит от количества полосок, ибо сортировка с временем $O(n^2)$ или $O(n \cdot \log_2 n)$ не требуется — достаточно пометить в том же массиве d признаком наличия те значения i (количество кипарисовых деревьев), которые есть в полосках, а затем просматривать массив d с конца — наибольшего значения i .

Методические рекомендации

Эту задачу следует рассматривать при изучении темы «Динамическое программирование». Она может быть развитием

классической задачи о камнях или рюкзаке. Полезно для осознания идеи решения, возникновения состояния понимания задачи школьником использовать метод «ручной» трассировки готового фрагмента логики решения.

Вторая часть решения (о полосках) может быть использована в качестве закрепляющего упражнения при изучении алгоритмов сортировки*, а именно, сортировки подсчетом.

Самостоятельные исследования

Пусть мы решаем задачу только для полей. Как следует изменить формулировку задачи, чтобы «нарушить» ее аддитивность?

«Эмподио»

Условие задачи

Известный математик и философ Пифагор считал, что реальность носит математический характер. Современные биологи изучают свойства *биопоследовательностей*. Биопоследовательность — это последовательность M целых чисел, которая:

- содержит все целые числа $0, 1, \dots, M-1$;
- начинается с 0 и заканчивается числом $M-1$;
- не имеет двух подряд идущих элементов $E, E+1$ именно в этом порядке.

Подпоследовательность идущих подряд элементов биопоследовательности называется *сегментом*.

Сегмент биопоследовательности называется *фрейм-интервалом*, если он содержит все целые числа, значения которых находятся между значениями первого и последнего элементов сегмента. При этом первый элемент должен быть наименьшим в сегменте, а последний элемент — наибольшим в сегменте и отличным от первого. Фрейм-интервал называется *эмподио*, если он не содержит внутри себя более короткий фрейм-интервал.

В качестве примера рассмотрим биопоследовательность $(0, 3, 5, 4, 6, 2, 1, 7)$. Вся биопоследовательность является фрейм-интервалом. Эта биопоследовательность, в свою очередь, содержит другой фрейм-интервал $(3, 5, 4, 6)$, и, следовательно, она не является эмподио. Фрейм-интервал $(3, 5, 4, 6)$ не содержит более короткий фрейм-интервал, и поэтому он является эмподио. Этот фрейм-интервал является единственным эмподио в данной биопоследовательности.

* Окулов С. М. Основы программирования. М.: БИНОМ. Лаборатория знаний. 2005. С. 249–250.

Требуется написать программу, которая по заданной биопоследовательности находит все эмподии, которые в ней содержатся.

Входные данные. Первая строка входного файла с именем `empodia.in` содержит одно целое число M — количество целых чисел во входной биопоследовательности. Следующие M строк содержат целые числа — элементы биопоследовательности. Каждая из этих M строк содержит одно-единственное целое число.

Выходные данные. Первая строка выходного файла с именем `empodia.out` содержит одно целое число N — количество эмподии, которые есть во входной биопоследовательности. Следующие N строк описывают все имеющиеся эмподии во входной биопоследовательности. Каждая из этих строк содержит два целых числа A и B (именно в этом порядке), разделенных пробелом, где A — порядковый номер элемента входной биопоследовательности, который является первым элементом эмподии; B — порядковый номер элемента входной биопоследовательности, который является последним элементом эмподии. Эмподии требуется выводить в порядке увеличения числа A .

Пример.

`empodia.in:`

8
0
3
5
4
6
2
1
7

`empodia.out:`

1
2 5

Ограничения. Для одного теста $1000000 \leq M \leq 1100000$. Для всех других тестов $1 \leq M \leq 60000$. В 50% тестов $M \leq 2600$.

Обсуждение задачи*

Эта задача, с одной стороны, очень простая, с другой — очень сложная. Первый вариант — решение с квадратичной временной сложностью $O(n^2)$, второй — с линейной временной

* Авторы благодарят Р. В. Шарыгина за помощь в разборе данной задачи.

сложностью $O(n)$. «Уловить» разницу во времени решения можно только на тестах со значением $M > 10^6$. Для меньших значений M алгоритм с квадратичной сложностью дает, естественно, правильный результат за приемлемое время.

Рассмотрим первый вариант решения.

Пусть последовательность чисел хранится в массиве x . Значения элементов массива увеличиваем при вводе на единицу. Так, для примера из формулировки задачи $x = (1, 4, 6, 5, 7, 3, 2, 8)$. Находим фрейм-интервал, который может начинаться с элемента i . Часть последовательности $[i, j]$ (начинается с элемента i и заканчивается элементом j) образует фрейм-интервал, если $x[j] - x[i] = j - i$, элемент $x[j]$ является максимальным на этом участке последовательности и среди чисел между i и j нет чисел, меньших, чем $x[i]$. Очевидно, что в этом случае находится самый короткий фрейм-интервал с начальным элементом $x[i]$, а время поиска всех фрейм-интервалов пропорционально $O(n^2)$ — есть $O(n)$ элементов и для каждого элемента мы просматриваем примерно $O(n)$ элементов, расположенных справа от него.

После того как все фрейм-интервалы найдены и мы запомнили их в массиве y (начальную и конечную точки), остается исключить лишние, т. е. оставить только самые короткие фрейм-интервалы.

Пусть есть два фрейм-интервала — (i_1, j_1) и (i_2, j_2) . Во-первых, в массиве y данные по (i_2, j_2) записаны после (i_1, j_1) и $i_2 > i_1$. Во-вторых, оказывается, что фрейм-интервалы не могут пересекаться, возможна только их вложенность — рис. 16.18. И если этот факт справедлив (а это действительно так), то исключить фрейм-интервалы, содержащие в себе более короткие, можно за время $O(n)$. Для этого, начав с конца массива, следует проверять условие: $(i_2 \leq j_1) \text{ And } (j_2 \leq j_1)$. При его выполнении (i_1, j_1) исключается из рассмотрения.

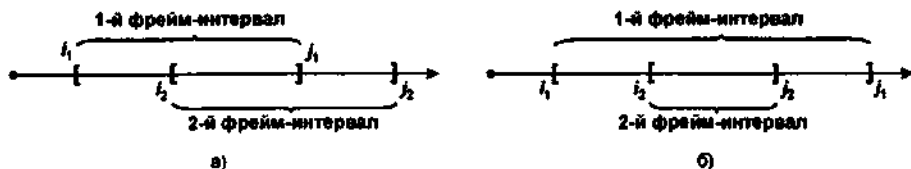


Рис. 16.18. Взаимное расположение фрейм-интервалов:
а) недопустимое; б) возможное

После этого остается только вывести оставшиеся в массиве y фрейм-интервалы: они и являются эмподио.

Утверждение. Концевые (первая и последняя) точки одного эмподио не могут находиться во внутренней части другого эмподио.

Первый случай: концевые точки одного эмподио находятся во внутренней части другого эмподио. Тогда, согласно определению, внешний интервал не является эмподио. Противоречие!

Второй случай: только одна из концевых точек лежит во внутренней части другого эмподио. Считаем, что на рис. 16.18 оба фрейм-интервала — эмподио. На рис. 16.8 первый эмподио содержит внутри себя начальную точку i_2 второго эмподио. Тогда можно заключить, что $i_1 < i_2 < j_1 < j_2$, так как элемент i_2 — есть внутренняя точка первого эмподио, а элемент j_1 — внутренняя точка второго эмподио. Возьмем любой элемент последовательности в интервале (i_2, j_1) . Очевидно, что он больше, чем элемент на месте i_2 , и меньше, чем элемент на месте j_1 . Получаем, что либо первый эмподио, либо второй таковым не является, ибо значения последовательности в интервале $[i_2, j_1]$ образуют фрейм-интервал. Противоречие доказывает утверждение.

Методическое отступление. Разбор первого варианта решения задачи может быть продолжен предложением по анализу одного из способов программной реализации логики с целью его улучшения (исключение лишних переменных и так далее).

Procedure Solve; {В массиве x хранится исходная последовательность. Границы фрейм-интервалов хранятся в массиве y . Его тип — $TSol = \text{Array}[1..NMax, 1..2]$ Of Integer, где $NMax$ — константа, определяющая максимально возможный размер массива}

Var i, j, p : Integer;
ok, stop: Boolean;

Begin

$m := 0$; {Счетчик количества фрейм-интервалов}

For $i := 1$ **To** $n - 1$ **Do** **Begin**

$p := x[i]$; {Находим, если он есть, фрейм-интервал, начинающийся с элемента $x[i]$ }

$j := i + 1$; {Начинаем со следующего элемента последовательности}

$ok := \text{False}$; { $ok = \text{True}$ говорит о том, что найден фрейм-интервал}

$stop := \text{False}$; {Фрейм-интервал не может начинаться с элемента $x[i]$, либо найден фрейм-интервал. Основанием для вывода о невозможности существования фрейм-интервала, начинающегося с элемента $x[i]$, является "встреча" с элементом $x[j]$, значение которого меньше, чем $x[i]$ }

```

While (j<=n) And (Not stop) Do Begin
  If x[j]<x[i] Then stop:=True
  Else
    If x[j]>p Then Begin
      p:=x[j]; {Находим максимальный элемент,
точнее, фиксируем его при просмотре "хвоста"
последовательности (элементов последовательности,
расположенных после элемента x[i])}
      If (j-i)=(p-x[i]) Then Begin {Найден
фрейм-интервал}
        stop:=True;
        ok:=True;
      End;
    End;
    j:=j+1;
  End;
  If ok Then Begin {Запоминаем найденный фрейм-
интервал}
    m:=m+1;
    y[m,1]:=i;
    y[m,2]:=j-1;
  End;
End;
k:=m; {Находим эмподию путем исключений фрейм-
интервалов большей длины}
For i:=m DownTo 2 Do
  If (y[i,1]<=y[i-1,2]) And (y[i,2]<=y[i-1,2])
Then
  Begin
    y[i-1,1]:=y[i,1];
    y[i-1,2]:=y[i,2];
    y[i,1]:=-1;
    k:=k-1;
  End;
End;

```

Задача решается за линейное время, если обратиться к одной идее из раздела информатики, связанного с методом *сортировки инверсиями*^{*}. Обозначим перестановку как $\pi = (\pi_1 \dots \pi_n)$. Цель сортировки инверсиями — по данной перестановке $\pi = (\pi_1 \dots \pi_n)$ найти наименьшее количество инверсий, преобразующих ее в тождественную перестановку $(1 \dots n)$. Применение инверсии

^{*} Berman, P., Hannenhall S. Fast sorting by reversals // 1996.

$\rho(i, j)$ к перестановке $\pi = (\pi_1 \dots \pi_n)$ изменяет порядок элементов $\pi_i, \pi_{i+1}, \dots, \pi_j$ на обратный $\pi_j, \pi_{j-1}, \dots, \pi_i$.

Как отмечалось, последовательность целых чисел (бипоследовательность) есть ни что иное как перестановка чисел $(0 \dots n-1)$ или $(1 \dots n)$. Для фрейм-интервала верны утверждения:

- для первой π_i и последней π_j точек фрейм-интервала выполняется условие: $\pi_j = \pi_i + j - i$;
- все значения, находящиеся внутри фрейм-интервала, различны и принадлежат отрезку $[\pi_i + 1, \pi_j - 1]$.

Преобразуем перестановку (бипоследовательность) обсуждаемой задачи в *знаковую перестановку*, в которой всем элементам присвоен знак «+», следующим образом. Элемент $+x$ заменим элементами $2x - 1, 2x$, а элемент $-x - 2x, 2x - 1$ — и дополним перестановку элементами $\pi_0 = 0$ и $\pi_{2n+1} = 2n + 1$ (рис. 16.19).

+8 +7 +1 +3 +2 +5 +4 +6
0 15 16 13 14 1 2 5 6 3 4 9 10 7 8 11 12 17

Рис. 16.19. Пример преобразования в знаковую перестановку

Построим *граф циклов** по полученной перестановке: соединим вершины π_{2i} и π_{2i+1} «черными» ребрами (непрерывная линия); вершины $\pi_k = 2i$ и $\pi_l = 2i + 1$ — «серыми» ребрами (пунктирная линия), где $0 \leq i \leq n$ (рис. 16.20). Назовем два ребра (π_i, π_j) и (π_k, π_l) *перекрывающимися*, если перекрываются, но не содержат друг друга интервалы $[i, j]$ и $[k, l]$. Например, ребра (π_1, π_4) и (π_3, π_{16}) есть перекрывающиеся ребра, а (π_1, π_4) и (π_9, π_5) — нет (рис. 16.20). Циклы C_1 и C_2 перекрывают друг

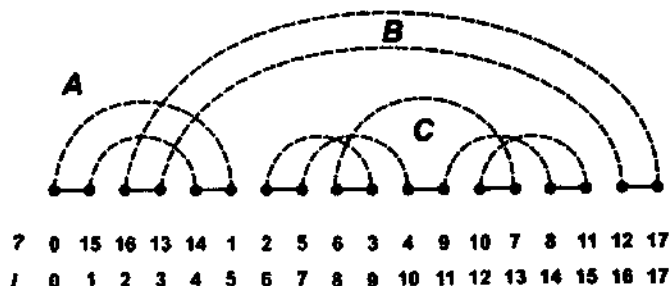


Рис. 16.20. Граф циклов ($n = 8$)

* Bader A., Bernard M., Moret E. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. Mi Yan // 2001.

друга, если они содержат перекрывающиеся ребра. Например, циклы A и C — не перекрывают друг друга, а циклы A и B перекрываются, так как содержат перекрывающиеся ребра, например ребра (π_1, π_4) и (π_3, π_{16}) .

Пространством цикла C называется интервал, левой границей которого является минимальный номер вершин цикла, а правой границей — максимальный номер вершин цикла или $[C.b = \min\{i | \pi_i \in C\}, C.e = \max\{j | \pi_j \in C\}]$. Пространством множества циклов $\{C_1, \dots, C_k\}$ называется интервал $[b = \min_{i=1}^k (C_i.b), e = \max_{j=1}^k (C_j.e)]$. На рис. 16.20 пространство цикла B есть интервал $[2, 17]$; пространство циклов A и B есть интервал $[0, 17]$.

Главное — что каждый цикл есть некоторая перестановка элементов, и применение инверсий упорядочивает элементы в пространстве цикла. Рассмотрим цикл C на рис. 16.20 и соответствующую ему часть перестановки на рис. 16.19. Видим, что внутренняя часть цикла может быть упорядочена применением инверсий, не выходящих за пределы пространства цикла (рис. 16.21).

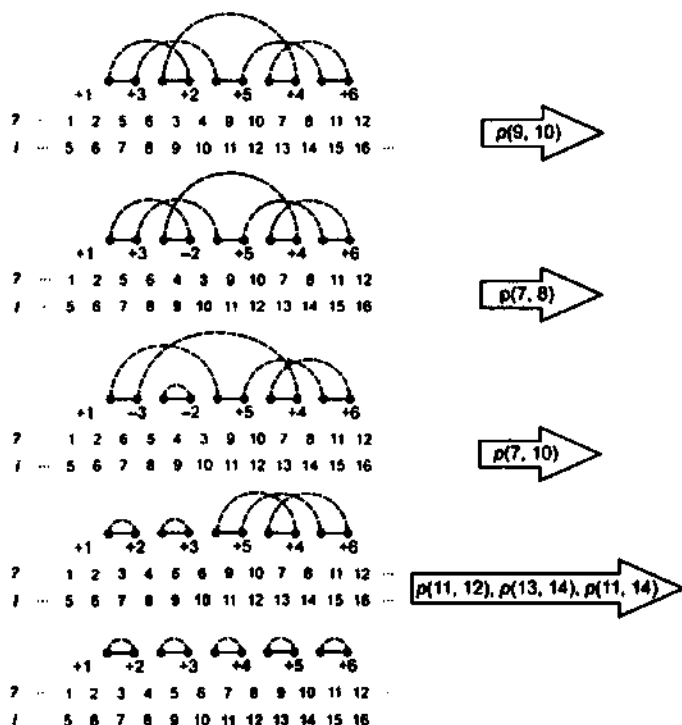


Рис. 16.21. Упорядочивание элементов в пространстве цикла C

Если рассмотреть элементы эмподии, а точнее, их упорядоченность, то мы увидим, что первый и последний элементы в последовательности уже упорядочены и стоят на своих местах, а внутренние элементы нуждаются в перестановке. То есть, внутренние элементы образуют цикл, не выходящий за пределы первого и последнего элемента эмподии и не содержащий в своем пространстве других циклов. Наличие таких циклов означало бы, что наш эмподии — всего лишь фрейм-интервал. Пространство цикла C не содержит в себе других циклов, т. е. цикл C выделяет некоторую перестановку или элементы внутри эмподии.

Определим *граф перекрытий**. Его вершинами являются циклы графа, а ребра соединяют вершины, в том случае, когда циклы перекрываются. Для рассматриваемого примера граф перекрытий приведен на рис. 16.22.

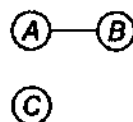


Рис. 16.22. Граф перекрытий для графа на рис. 16.20

Выделим в графе перекрытий *компоненты связности*. Фактически мы подошли к утверждению о том, что *компонента связности графа перекрытий определяет фрейм-интервал*. И если построение графа перекрытий и нахождение компонент связности будет осуществляться за линейное время, то цель достигнута.

Приведем еще один пример — рис. 16.23: (а) — преобразование биопоследовательности в беззнаковую перестановку; (б) — граф циклов; (в) — граф перекрытий. Последовательность содержит два эмподии, и их (компоненты связности) можно определить только на основе анализа «серых» ребер графа циклов. При этом, естественно, циклы, содержащие начальный и конечный элементы беззнаковой перестановки, исключаются. Каждое ребро в анализе используется два раза (начало и конец ребра), поэтому опишем его в массиве *edg* соответствующим образом: $edg[0] = (0, 15)$; $edg[1] = (1, 16)$; $edg[2] = (2, 5)$; $edg[3] = (3, 14)$; $edg[4] = (4, 17)$; $edg[5] = (2, 5)$; $edg[6] = (6, 11)$; $edg[7] = (7, 10)$; $edg[8] = (8, 13)$; $edg[9] = (9, 12)$; $edg[10] = (7, 10)$; $edg[11] = (6, 11)$; $edg[12] = (9, 12)$; $edg[13] = (8, 13)$; $edg[14] = (3, 14)$; $edg[15] = (0, 15)$; $edg[16] = (1, 16)$; $edg[17] = (4, 17)$; $edg[18] = (18, 21)$; $edg[19] = (19, 22)$; $edg[20] = (20, 23)$; $edg[21] = (18, 21)$; $edg[22] = (19, 22)$; $edg[23] = (20, 23)$; $edg[24] = (24, 25)$; $edg[25] = (24, 25)$. Обратим внимание на тот факт, что у части ребер номер первой вершины

* Bader, A. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study / David A. Bader, Bernard M. E. Moret, Mi Yan // 2001.

Таблица 16.5

Номер итерации (ребра)	Состояние стека	Действия	Новое состояние стека
0, 1, 2, 3, 4	(4, 17) (3, 14) (2, 5) (1, 16) (0, 15)	Добавляем в стек соответствующие ребра — перекрытия нет	
5, ребро (2, 5)	(4, 17) (3, 14) (2, 5) (1, 16) (0, 15)	Есть перекрытие — последовательно изменяем границы интервала на вершине стека	(2, 17) (1, 16) (0, 15)
6, 7, 8, 9	(9, 12) (8, 13) (7, 10) (6, 11) (2, 17) (1, 16) (0, 15)	Добавляем в стек соответствующие ребра — перекрытия нет	
10, ребро (7, 10)	(9, 12) (8, 13) (7, 10) (6, 11) (2, 17) (1, 16) (0, 15)	Есть перекрытие — последовательно изменяем границы интервала на вершине стека	(7, 13) (6, 11) (2, 17) (1, 16) (0, 15)
11, ребро (6, 11)	(7, 13) (6, 11) (2, 17) (1, 16) (0, 15)	Есть перекрытие — последовательно изменяем границы интервала на вершине стека	(6, 13) (2, 17) (1, 16) (0, 15)
12	(6, 13) (2, 17) (1, 16) (0, 15)		(6, 13) (2, 17) (1, 16) (0, 15)
13, ребро (8, 13)	(6, 13) (2, 17) (1, 16) (0, 15)	Номер ребра совпадает с правым концом интервала на вершине стека. Запоминаем фрейм-интервал	(2, 17) (1, 16) (0, 15)

Номер итерации (ребра)	Состояние стека	Действия	Новое состояние стека
14	(2, 17) (1, 16) (0, 15)		(2, 17) (1, 16) (0, 15)
15, ребро (0, 15)	(2, 17) (1, 16) (0, 15)	Есть перекрытие — последовательно изменяем границы интервала на вершине стека	(0, 17)
16	(0, 17)		(0, 17)
17, ребро (4, 17)	(0, 17)	Номер ребра совпадает с правым концом интервала на вершине стека. Фрейм-интервал не запоминаем, одна из границ (левая) совпадает с началом	
18, 19, 20	(20, 23) (19, 22) (18, 21)	Добавляем в стек соответствующие ребра — перекрытия нет	
21, ребро (18, 21)	(20, 23) (19, 22) (18, 21)	Есть перекрытие — последовательно изменяем границы интервала на вершине стека	(18, 23)
22	(18, 23)		(18, 23)
23, ребро (20, 23)	(18, 23)	Выводим фрейм-интервал	
24	(24, 25)		(24, 25)
25	(24, 25)	Вывода нет. Граница совпадает с концом биопоследовательности (преобразованной)	—

При наличии перекрытия «серого» ребра с интервалом в вершине стека корректировка осуществляется за время, пропорциональное некоей константе. Таким образом, временная сложность поиска фрейм-интервалов (фактически это просмотр списка «серых» ребер) составляет $O(n)$, а значит, и общая сложность — $O(n)$.

Методическое отступление. Дальнейшее понимание логики поиска эмподио достигается путем «ручной» трассировки (таблицей типа 16.5) с использованием основных фрагментов реализации:

Type

Interval = Record

left, right: Integer;

End;

TInterval = Array [0..2 * MaxN] Of Interval;

Var stack, c: TInterval;

...

Begin

<Ввод данных>;

<Формирование беззнаковой перестановки и массива

(с) "серых" ребер>;

up := -1; {Указатель стека}

count := 0; {Счетчик числа фрейм-интервалов}

For i:=0 To t Do Begin {Значение t равно $(n+1)*2-1$ - количество элементов в беззнаковой перестановке}

If (i=c[i].left) Then Begin

up:=up+1;

stack[up]:=c[i];

End;

ext:=c[i]; {ext имеет тип Interval=Record left, right: Integer; End;}

While (stack[up].left>c[i].left) Do Begin {Если есть перекрытие, то изменяем границы и состояние стека}

ext.left:= Min(ext.left, stack[up].left);

ext.rigth := Max(ext.rigth, stack[up].right);

up:=up - 1;

End;

stack[up].left:=Min(ext.left, stack[up].left);

{Изменяем границы интервала в вершине стека}

stack[up].right:=Max(ext.rigth,

stack[up].right);

If (i=stack[up].right) Then Begin {Самое правое "серое" ребро?}

If (stack[up].left<>0) And (stack[up].right<>t)

Then Begin

<запись фрейм-интервала>; {Требуется преобразование из нумерации беззнаковой перестановки в исходную нумерацию элементов биопоследовательности и вычисление длины фрейм-интервала}

count:=count+1;

End;

up:=up-1;

End;

<поиск среди фрейм-интервалов эмподию и их вывод>;

End.

Методические рекомендации

Обсуждение различных вариантов решения этой задачи можно использовать как при изучении темы «Обработка последовательностей», так и тем «Алгоритмы на графах» или «Сортировка». В последнем случае необходимо «развернуть» материал по методу «сортировка инверсиями».

Самостоятельные исследования

Полное тестирование при длинах последовательностей, превышающих значение 1000000, — это уже исследование, достойное многих специалистов по информатике.

ЗАДАЧИ ПЕРВОГО ТУРА

«Сад»

Условия задачи

Байтмен владеет красивейшим садом в Байттауне, в котором он посадил n роз. Пришло лето, и цветы выросли большими и красивыми. Байтмен понял, что он не в состоянии самостоятельно ухаживать за всеми розами, и решил нанять двух садовников в помощь. В этом случае ему нужно выбрать две прямоугольные области, чтобы каждый из садовников ухаживал за розами в одной из них. Области не должны пересекаться, и в каждой должно быть ровно k роз. Байтмен хочет установить забор, огораживающий прямоугольные области. Для экономии денег забор должен быть как можно короче. Ваша задача — помочь Байтмену выбрать две прямоугольные области.

Сад представляет собой прямоугольник длиной l метров и шириной w метров, который разделен на $l \times w$ одинаковых единичных квадратов размером 1×1 метр каждый. Зафиксируем координатную систему так, чтобы оси координат были параллельны сторонам сада. Все квадраты имеют целые координаты (x, y) , удовлетворяющие ограничениям $1 \leq x \leq l$, $1 \leq y \leq w$. В каждом единичном квадрате может содержаться любое количество роз.

Стороны прямоугольных областей, которые выбираются, должны быть параллельны сторонам сада, а их угловые единичные квадраты — иметь целые координаты. Прямоугольная область с угловыми единичными квадратами (l_1, w_1) , (l_1, w_2) , (l_2, w_1) и (l_2, w_2) (для $1 \leq l_1 \leq l_2 \leq l$ и $1 \leq w_1 \leq w_2 \leq w$):

- содержит все единичные квадраты с координатами (x, y) , которые удовлетворяют условию $l_1 \leq x \leq l_2$ и $w_1 \leq y \leq w_2$, и
- имеет периметр $2 \cdot (l_2 - l_1 + 1) + 2 \cdot (w_2 - w_1 + 1)$.

Две прямоугольные области не должны пересекаться, то есть они не должны иметь ни одного общего квадрата. Даже если они имеют общую сторону или ее часть, они ограждаются разными заборами.

Задание. Напишите программу, которая:

- читает из стандартного входа размеры сада, общее количество роз в саду, количество роз, которое должно находиться в каждой прямоугольной области, и позицию каждой розы в саду, определяемую координатами единичного квадрата, в котором она находится;
- находит угловые единичные квадраты двух таких прямоугольных областей с минимальной суммой периметров, которые удовлетворяют заданным условиям;
- выводит в стандартный выход минимальное значение суммы периметров двух непересекающихся прямоугольных областей, каждая из которых содержит точно заданное количество роз (или единственное слово «NO», если такой пары прямоугольных областей не существует).

Входные данные. Первая строка стандартного входа содержит два числа: l и w ($1 \leq l, w \leq 250$), разделенные одним пробелом, — длину и ширину сада. Вторая строка содержит два числа: n и k ($2 \leq n \leq 5000$, $1 \leq k \leq n/2$), разделенные одним пробелом и обозначающие общее количество роз в саду и количество роз, которое должно быть в каждой из прямоугольных областей. Следующие n строк содержат позиции роз, по одной розе в строке. Каждая $(i+2)$ -я строка содержит два числа l_i, w_i ($1 \leq l_i \leq l$, $1 \leq w_i \leq w$), разделенные одним пробелом — координаты квадрата, содержащего i -ю розу.

В одном квадрате может содержаться две или большее количество роз.

В 50% тестов размеры сада будут удовлетворять ограничениям: $l, w \leq 40$.

Выходные данные. В первую и единственную строку стандартного выхода ваша программа должна вывести одно число — минимальную сумму периметров двух неперекрывающихся прямоугольных областей, каждая из которых содержит ровно k роз или единственное слово «NO», если таких прямоугольников нет.

Пример (рис. 17.1).

Входные данные:

```
6 5
7 3
3 4
3 3
6 1
1 1
5 5
5 5
3 1
```

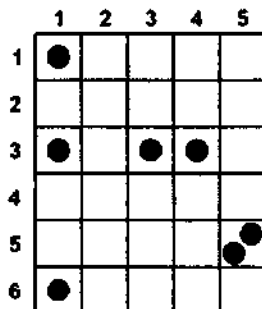


Рис. 17.1. Пример

Выходные данные:

22

Обсуждение задачи

Немного усложним пример из формулировки задачи. Пусть дополнительно в единичном квадрате (4, 2) находится три розы. Введем матрицу p . Элемент $p[i, j]$ равен количеству роз в единичном квадрате (i, j) сада. В матрице d за время, пропорциональное $O(w \times l)$, по формуле:

$$d(i, j) = \begin{cases} 0, & \text{если } i = 0 \text{ или } j = 0, \\ d(i-1, j) + d(i, j-1) - d(i-1, j-1) + p(i, j), & \text{иначе} \end{cases}$$

определим количество роз во всех прямоугольниках с противоположными углами $(1, 1)$ и (i, j) .

Для нашего примера матрица d имеет вид:

$$d = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 3 & 4 & 4 \\ 2 & 5 & 6 & 7 & 7 \\ 2 & 5 & 6 & 7 & 9 \\ 3 & 6 & 7 & 8 & 10 \end{pmatrix}$$

После этих действий количество роз в любом прямоугольнике с углами (i_1, j_1) и (i_2, j_2) вычисляется за время $O(1)$ по формуле: $d(i_2, j_2) - d(i_2, j_1 - 1) - d(i_1 - 1, j_2) + d(i_1 - 1, j_1 - 1)$.

Решение, основанное на перечислении всех пар прямоугольников, содержащих k роз, имеет временную сложность $O(w^2, l^2)$ и не удовлетворяет полностью требованию по времени выполнения задачи.

Очевидно, что любая искомая пара не пересекающихся прямоугольников разделяется вертикальной или горизонтальной прямой. Каждая прямая (а их $w+l-2$) разбивает исходный прямоугольник на две части, в которых и требуется найти по прямоугольнику с k розами, с минимальным суммарным периметром. Эта идея упрощает решение, но отвлечемся немного и рассмотрим более простую задачу.

Дан одномерный массив из целых чисел. Требуется найти все его «непрерывные участки» с суммой элементов, равных заданному числу k .

Пусть дан массив $a[1..10] = (15, 3, 4, 1, 7, 8, 4, 2, 6, 6)$, а $k = 12$. Идея решения иллюстрируется на рис. 17.2. Мы определяем некое окно (назовем его *скользящим*), «вырезающее»

фрагмент массива. Если сумма элементов окна меньше k , то увеличиваем верхнюю границу окна, а если больше k , то — нижнюю границу окна. Факт совпадения фиксируется, и происходит увеличение нижней границы.

Обобщим идею «скользящего окна» для плоскости. Зафиксируем (временно) значения (j_1, j_2) , $1 \leq j_1 \leq j_2 \leq L$. В этом случае окном является прямоугольник с углами (i_1, j_1) и (i_2, j_2) . Величины i_1 и i_2 изменяются по принципу одномерного окна; а именно, до тех пор, пока $i_2 \leq w$, выполняем действия:

- когда в прямоугольнике, соответствующем окну, находится k роз, вычисляем и запоминаем его периметр, а i_1 увеличиваем на единицу;
- если в прямоугольнике количество роз меньше k , то увеличиваем i_2 на единицу, иначе (больше k) — значение i_1 на единицу.

Пусть $j_1 = 1$, а $j_2 = 3$; для нашего примера изменение «скользящего окна» приведено на рис. 17.3. В прямоугольниках цифрой обозначено количество роз. Из рис. 17.3 ясно, что для данных значений j_1 и j_2 существует прямая, разбивающая сад на две части, в каждой из которых есть прямоугольник, содержащий k роз. Суммарный периметр прямоугольников равен 20. Осталось перебрать все пары значений j_1 и j_2 . Аналогичную работу следует выполнить и для другого направления — «скользящее окно» перемещается по другой оси координат.

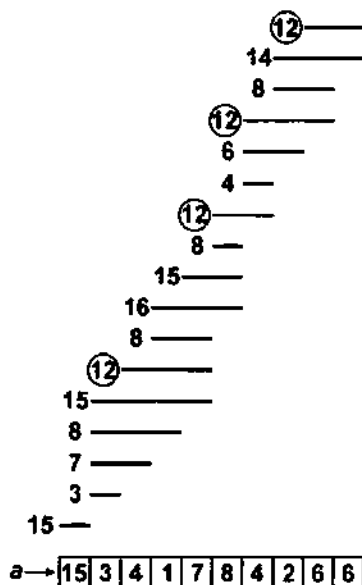


Рис. 17.2. Иллюстрация понятия «скользящее окно»

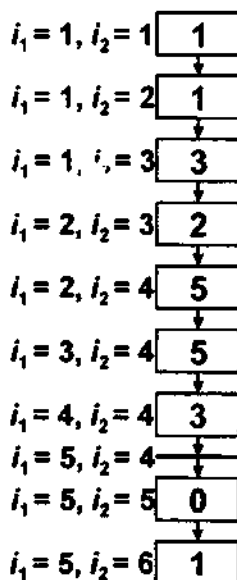


Рис. 17.3. Изменение «скользящего окна» при $j_1 = 1$ и $j_2 = 3$

Ответом в рассматриваемом примере являются два прямоугольника, выделенные на рис. 17.4; их суммарный периметр равен 14.

Методические рекомендации

1. «Натолкнуть» школьника на идею решения задачи может знание алгоритмов поиска подстроки в строке. В алгоритмах Кнута-Мориса-Пратта, Бойера-Мура, Рабина-Карпа*, Ханта-Зиманского используется различная техника сдвига подстроки для определения места ее вхождения в строку. Наиболее близок к задаче алгоритм Рабина-Карпа — он основан на той же идее и обобщается для задач на целочисленных двумерных структурах данных. Остальные алгоритмы могут использоваться (в «идейном» плане) для дальнейшего улучшения временных оценок обсуждаемого решения.
2. Введением прямой, разделяющей сад на две части по вертикали или горизонтали, мы определили *другой порядок перечисления прямоугольников*. Теперь нужно рассматривать не все пары прямоугольников, а только те, которые удовлетворяют введенному порядку. И, как обычно, порядок «вытягивается» из условий задачи. Хорошей проверкой понимания работы со «скользящим окном» является вопрос определения количества пар прямоугольников, в каждом из которых k роз, по заданному рисунку расположения роз в саду. После правильных ответов на него можно смело переходить на разработку программного кода.

	1	2	3	4	5
1	1	0	0	0	0
2	0	0	0	0	0
3	1	0	1	1	0
4	0	3	0	0	0
5	0	0	0	0	2
6	1	0	0	0	0

Рис. 17.4. Прямоугольники для примера из обсуждения задачи

Самостоятельные исследования

1. Задача естественным образом обобщается на случай поиска, например, в матрице $n \times n$ квадрата (или прямоугольника) $m \times m$ с известным или почти известным содержанием.
2. Пусть исходная матрица $n \times n$ — логического типа и описывает некое большое изображение различных букв латинского алфавита. Буква «А» может иметь различное написание и размер (прямоугольники $w \times v$), но каждое ее написание характеризуется некоей интегральной характеристикой (например, количеством элементов прямоугольника, имеющих значение true). Найдите количество букв «А» в изображении.

* Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦНМО, 1999. С. 784.

«Средняя последовательность»

Условие задачи

Рассмотрим неубывающую последовательность $s_1, \dots, s_{n+1}$ ($s_i \leq s_{i+1}$ для $1 \leq i \leq n$). Последовательность $m_1, \dots, m_n$, в которой каждый член определен как $m_i = 1/2 \cdot (s_i + s_{i+1})$ для $1 \leq i \leq n$, назовем «средней последовательностью» для последовательности $s_1, \dots, s_{n+1}$. Например, средняя последовательность для последовательности 1, 2, 2, 4 есть 1.5, 2, 3. Заметим, что элементы средней последовательности могут быть дробными числами. Тем не менее, в данной задаче используются только те средние последовательности, в которых все числа целые.

Для заданной неубывающей последовательности из n целых чисел $m_1, \dots, m_n$ необходимо вычислить количество всех неубывающих последовательностей из $n+1$ целых чисел $s_1, \dots, s_{n+1}$, для которых заданная последовательность $m_1, \dots, m_n$ является средней последовательностью.

Задание. Напишите программу, которая:

- читает из стандартного ввода неубывающую последовательность целых чисел;
- вычисляет количество всех неубывающих последовательностей целых чисел, для которых заданная последовательность является средней последовательностью;
- выводит ответ в стандартный вывод.

Входные данные. Первая строка стандартного ввода содержит одно целое число n ($2 \leq n \leq 5000000$). Оставшиеся n строк содержат значения последовательности $m_1, \dots, m_n$. Строка $i+1$ содержит одно целое число m_i ($0 \leq m_i \leq 1000000000$). Можно считать, что для 50% тестов будут справедливы неравенства $n \leq 1000$ и $0 \leq m_i \leq 20000$.

Выходные данные. Ваша программа должна вывести в стандартный вывод ровно одно целое число — количество всех неубывающих последовательностей целых чисел, для которых входная последовательность является средней последовательностью.

Пример.

Входные данные:

3
3
2
5

Выходные данные: 4

Для данного примера существует ровно четыре неубывающих последовательности целых чисел, для которых последовательность 2, 5, 9 является средней последовательностью:

- 2, 2, 8, 10,
- 1, 3, 7, 11,
- 0, 4, 6, 12,
- -1, 5, 5, 13.

Идея решения

Первое, что «бросается в глаза» при чтении задачи, — последовательности $s_1, \dots, s_{n+1}$ вычислять не требуется, — необходимо найти лишь их количество. Из этого факта и временных ограничений делаем вывод об обреченности всех вариантов решения задачи через попытки генерации последовательностей $s_1, \dots, s_{n+1}$.

Второе: из примера в формулировке задачи следует, на первый взгляд, что второй член последовательности может быть равен всем числам из интервала $[m_1, m_2]$. Если это так, то решение — сверхпростое, и оно настораживает именно этим, а также тем, что не все исходные данные задачи используются. Однако примем это утверждение в качестве рабочей гипотезы и попытаемся найти контрпример.

Пусть дана последовательность средних значений 2, 5, 7. Восстанавливаем последовательности: $s_2 = 2$, последовательность — 2, 2, 8, 6 (не удовлетворяет условию задачи — она не является неубывающей); $s_2 = 3$, последовательность — 1, 3, 7, 7; $s_2 = 4$, последовательность — 0, 4, 6, 8; $s_2 = 5$, последовательность — -1, 5, 5, 9. Итак, наше предположение неверно. Но стал очевидным тот факт, что при знании хотя бы одного элемента последовательности s мы восстанавливаем по нему и по средним значениям всю последовательность. Кроме того, количество последовательностей уменьшилось на единицу. Смотрим: $5-2+1 = 4$, $9-5+1 = 5$ для примера в условии задачи и $5-2+1 = 4$, $7-5+1 = 3$ в измененном примере.

Обозначим через $k = \min_{2 \leq i \leq n} (m_i - m_{i-1})$ и выскажем утверждение о том, что количество искоемых последовательностей равно $k+1$. Сформулируем эту мысль более формально.

Утверждение. Количество неубывающих последовательностей целых чисел $s_1, \dots, s_{n+1}$ с данной неубывающей последовательностью целых средних значений $m_1, \dots, m_n$ в точности совпадает со значением $k+1$.

Обоснование. Пусть t таково, что $m_t - m_{t-1} = k$. Рассмотрим отрезок $[m_{t-1}, m_t]$. Очевидно, он содержит в себе ровно $k+1$ целое число. Но по таким числам неубывающая последовательность $\{s_i\}$ целых чисел с заданным набором средних восстанавливается однозначно.

Реализация утверждения требует $O(n)$ времени и $O(1)$ памяти.

Методические рекомендации

Эта задача относится к разряду несложных; главное — «заострить» внимание на том, что исходя из временных ограничений, вычисление последовательностей $\{s_i\}$ — тупиковая ветвь рассмотрения. После этого несколькими примерами надо «натолкнуть» на мысль, не подсказывая ее (пусть факт открытия принадлежит самому школьнику), сформулированную в *утверждении*.

Самостоятельные исследования

1. Любая последовательность $\{s_i\}$, $i = 1..n$ образует $n(n-1)/2$ сумм, если сложить все возможные пары. По заданному набору сумм требуется, если возможно, восстановить последовательность*.
2. В теории чисел есть удивительная по простоте, но нерешенная проблема — «*гипотеза Сиракуз*»**. Дано число n . Если n четно, то поделим его на 2, а иначе — умножим на 3 и прибавим 1. Процесс будем повторять с новым значением n до тех пор, пока n не станет равным 1. *Пример:* 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Доказательство конечности алгоритма для всех значений n неизвестно. В качестве вопроса для исследования можно сформулировать следующее: для всех чисел от n до m ($n < m$) проверить «гипотезу Сиракуз».

«Горы»

Условие задачи

В Горном Парке Развлечений открылся новый аттракцион с американскими горками. Трек в аттракционе состоит из n рельсов, соединенных последовательно друг с другом, причем первый рельс начинается на высоте 0. Оператор Байтмэн может изменять конфигурацию трека по своему усмотрению, корректируя наклон некоторых последовательно соединенных рельсов. Наклон остальных рельсов при этом не изменяется.

* Скиена С. С., Ревилла М. А. Олимпиадные задачи по программированию. Руководство по подготовке к соревнованиям. М.: КУДИЦ-ОБРАЗ, 2005. С. 146.

** Ноден П., Ките К. Алгебраическая алгоритмика. М.: Мир, 1999. С. 31.

Всякий раз, когда наклон некоторых рельсов изменяется, все следующие за ними рельсы соответственно поднимаются или опускаются. При этом высота начала трека всегда остается равной 0. На рис. 17.5 показан пример изменения конфигурации трека.

Каждый заезд начинается с того, что кабина запускается из начала трека с энергией, достаточной для достижения высоты h . Иными словами, кабина будет продолжать движение по рельсам, пока ее высота не превысит h или пока она не достигнет конца трека.

Вычислите по записям обо всех заездах и изменениях конфигураций трека число рельсов, которые были полностью пройдены кабиной в каждом из заездов до ее остановки.

В аттракционе трек описывается последовательностью из n наклонов, по одному для каждого рельса. i -е число d_i равно разнице высот (в сантиметрах) между концом i -го рельса и его началом. Иными словами, если после прохождения по первым $i-1$ рельсам кабина оказывается на высоте h сантиметров, то после прохождения по i рельсам она будет на высоте $h + d_i$ сантиметров.

Изначально все рельсы горизонтальны, то есть $d_i = 0$ для всех i . Заезды и изменения конфигурации происходят в течение дня. Каждое изменение конфигурации описывается тремя числами: a , b и D . Такое изменение затрагивает рельсы с a -го по b -й (включительно). Наклон каждого из этих рельсов устанавливается равным D . Иными словами, $d_i = D$ для всех $a \leq i \leq b$.

Каждый заезд задается одним числом h — максимальной высотой, на которую может подняться кабина.

Задание. Напишите программу, которая:

- читает из стандартного ввода последовательность описаний изменений конфигурации трека и заездов;
- для каждого заезда вычисляет количество рельсов, которые пройдены кабиной;
- выводит результаты в стандартный вывод.

Входные данные. Первая строка входных данных содержит одно положительное целое число n — количество рельсов в треке, $1 \leq n \leq 1000000000$. Последующие строки содержат описания изменений конфигурации и заездов. Последняя строка входных данных содержит признак окончания. Каждая строка, начиная со второй, может содержать:

- описание изменения конфигурации трека — один символ «I» и целые числа a , b и D , разделенные одним пробелом ($1 \leq a \leq b \leq n$, $-1000000000 \leq D \leq 1000000000$);

- описание заезда — один символ «Q» и целое число h ($0 \leq h \leq 1000000000$), разделенные одним пробелом;
- один символ «E» — признак окончания входных данных.

Вы можете предполагать, что в каждый момент высота любой точки трека содержится в промежутке $[0, 1000000000]$. Входные данные содержат не более 100000 строк. В 50% тестов число n будет удовлетворять ограничению $1 \leq n \leq 20000$, и входные данные будут содержать не более 1000 строк.

Выходные данные. i -я строка выходных данных должна содержать единственное целое число — количество рельсов, которые пройдены кабиной в i -м заезде.

Пример.

Входные данные:

```
4
Q 1
I 1 4 2
Q 3
Q 1
I 2 2 -1
Q 3
E
```

Выходные данные:

```
4
1
0
3
```

На рис. 17.5 представлены изображения трека до и после каждого изменения конфигурации. По оси x отложены номера рельсов. По оси y отложена высота. Числа рядом с выделенными точками обозначают их высоту. Числа рядом с отрезками обозначают наклоны.

Обсуждение задачи*

Ограничения в формулировке задачи говорят о том, что для полного решения требуется ло-

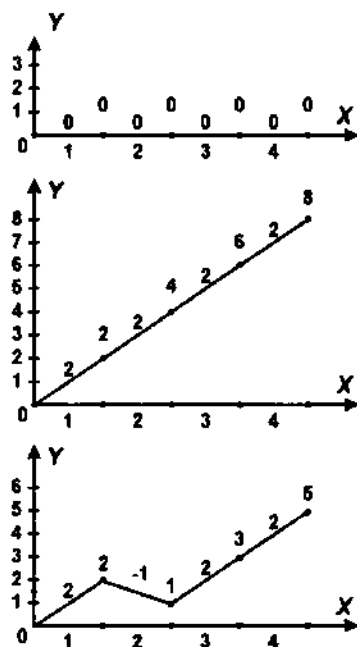


Рис. 17.5. Изображение трека до и после каждого изменения конфигурации. По оси x отложены номера рельсов. По оси y отложена высота. Числа рядом с выделенными точками обозначают их высоту. Числа рядом с отрезком означают наклоны

* Авторы благодарят О. А. Пестова за помощь в разборе задачи.

гарифмическая временная сложность алгоритма. Из этого следует, что данные о треке должны быть представлены в виде некой древовидной структуры (бинарного дерева, дерева интервалов или АВЛ-дерева).

Простые варианты решения.

Опишем трек массивом a , где значение элемента $a[i]$ соответствует высоте конца i -го рельса. Как при изменении конфигурации трека, так и при обработке запроса в этом случае требуется выполнить порядка $O(n)$ действий, а общая сложность составляет $O((I + Q)n)$, где I — количество запросов, а Q — количество заездов.

Другой вариант решения основан на представлении трека в виде последовательности интервалов, на протяжении которых наклон рельсов постоянен. В этом случае время изменения конфигурации трека пропорционально $O(I)$, а общая сложность решения составляет $O((I + Q)I)$.

Перейдем к решению, полностью отвечающему требованиям задачи. В формулировке задачи есть ключевая фраза — она выделена нами курсивом. Информация обо всех изменениях конфигурации трека известна заранее. Что можно извлечь из того, что заранее известны все точки излома трека, а значит, и длины участков с постоянным наклоном? (Мы пока не считаем высоты, наклоны и т. д.) Извлечь можно некую структуру данных (мы создаем ее при вводе), на которую «накладывается» все дальнейшее решение задачи и которая обеспечивает быструю (логарифмическую) обработку запросов типа Q и I . Очевидно, что эта структура должна иметь древовидный характер, иначе мы не получим логарифмического времени работы.

Предположим, что такая структура данных создана. Назовем ее *деревом интервалов*. Мы опять-таки не говорим — как, какими методами она создана. Примем гипотетически — вот «если бы было так», то задача решалась бы быстро, и это «если» строится только на основе входных данных. Для примера из формулировки задачи дерево интервалов показано на рис. 17.6. В вершинах дерева указаны границы интервалов, а в прямоугольниках рядом с вершинами — длины интервалов. Обратите внимание на нижний уровень дерева, на его соответствие входным данным. Все, что сверху, — это уже надстройка над ним. В корне дерева (верхней вершине) получен весь трек, общее

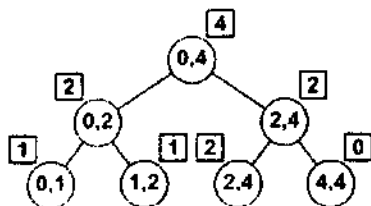


Рис. 17.6. Дерево интервалов для примера из формулировки задачи

количество рельсов. Главное заключается в том, что структура дерева не изменяется в процессе решения задачи. Сопутствующая (связанная) с деревом информация изменяется, — но не сама структура.

Методическое отступление. В этот момент лучше прервать обсуждение задачи и для простых примеров входных данных (произвольных) прорисовать соответствующие деревья интервалов, уяснив, что такое «сыновья» вершины (левый и правый), высота дерева, как высота дерева связана с вершинами нижнего уровня — «листьями». Эта операция выполняется формальным образом — строится нижний уровень, затем — следующий, и так до корня дерева. Обратите внимание на вершину «4, 4» и попытайтесь ответить на вопрос: с какой целью она введена? Требуется «натолкнуть» школьника на мысль о том, как строить дерево интервалов по входным данным, как его представить в памяти компьютера, с помощью каких структур данных. Если ученику известен метод *пирамидальной сортировки*, то «семена упадут на подготовленную почву».

Отложим пока вопрос о том, как формально на уровне структур данных строить дерево, и попытаемся понять, что оно нам дает и какую информацию необходимо с ним связать, чтобы задача имела простое и изящное решение.

Начнем с обработки первого запроса (пример из формулировки задачи) — Q 1. На входе нашей логики — адрес вершины дерева *root* и значение высоты *height*. В данном случае — корня дерева. Трек горизонтален, и мы проходим все четыре рельса.

Как за одну операцию ответить на этот запрос? Во-первых, с каждой вершиной дерева требуется связать некую логическую величину *u* — признак того, изменяется наклон рельсов на этом интервале (False) или нет (True). Если не изменяется (True), то данных, связанных с узлом, должно быть достаточно для ответа на вопрос. Становится очевидным, что нам требуется высота участка *h*. Пусть значение *h* меньше или равно *height*, тогда очевидно, что количество пройденных рельсов равно длине участка, а она известна. И в нашем случае — для запроса Q 1 — ответ равен четырем.

Однако обобщим рассуждения. При *h*, большем, чем значение *height*, мы проходим только часть этого участка с постоянным наклоном. Для вычисления части участка, кроме его длины и высоты, требуется знать перепад высот *s* на всем интервале. Определив значение перепада на единицы длины, мы можем подсчитать количество пройденных рельсов.

Случай неизменения наклона рельсов на интервале, представленном в вершине дерева (не обязательно в корне!), мы разобрали. Пусть значение логической величины равно `False` — наклон на интервале не постоянный. Если высота интервала у левого сына вершины меньше или равна значению *height*, то берем длину интервала у левого сына (количество пройденных рельсов) и идем к правому сыну с учетом пройденного перепада высот. В случае, когда высота интервала у левого сына вершины больше, чем *height*, мы идем к левому сыну, чтобы выяснить, как обстоят дела у него.

Итак, резюмируя, можно сказать, что с каждой вершиной дерева (*root*) должна быть связана следующая информация: *u[root]* — признак изменения, *h[root]* — высота интервала, *c[root]* — перепад высот на данном интервале, *t[root]* — длина интервала.

Набросок логики на нашем псевдокоде имеет вид:

```
Function Calc(root:<тип данных зависит от реализации
дерева интервалов>; height : LongInt) : LongInt;
  Var z:LongInt;
  Begin
    If (u[root]) Then Begin
      If h[root]<=height Then Calc:=t[root]
      (Берем длину интервала (количество рельсов))
      Else Begin
        z:=c[root] Div t[root];
        Calc:=height Div z;
      End;
    Exit;
  End;
  If h[левого сына]<-height Then
    Calc:=t[левого сына]+Calc(правого сына, height -
c[левого сына])
  Else Calc:=-Calc(левого сына, height);
End;
```

Таким образом, время обработки запроса типа Q при той сопутствующей информации (вершине дерева), которая нами описана, пропорционально высоте дерева, а она равна $O(\log_2 I)$.

Методическое отступление. В данный момент обсуждения целесообразно сделать паузу и вручную просчитать для вершин ранее составленных деревьев значения *u*, *h*, *c*, *t*.

Следующий вопрос обсуждения — о том, как изменять сопутствующую информацию, связанную с вершинами дерева

при обработке запросов типа I. Определим это как *реконфигурацию дерева*.

Рассмотрим первый запрос из примера — I 1 4 2. Интервал совпадает с тем, что дан в корне дерева (из нижней границы вычитаем единицу). Сопутствующая информация изменяется только у корня. Значение u остается равным True — наклон постоянный; значения h и c будут равны длине интервала, умноженной на величину наклона (*change*).

Примечание. Проверьте, что после этих изменений функция Calc даст правильный ответ на запросы Q 3 и Q 1 из формулировки задачи.

Рассмотрим второй запрос типа I — I 2 2 -1. Напомним, что с каждой вершиной дерева связаны следующие данные: u , h , c , t . Значения первых трех — $\langle u, h, c \rangle$ — показаны на рис. 17.7. Как они сформированы? Высота интервала h и перепад высот на интервале c относятся к данному конкретному интервалу, т. е. предполагается, что левый конец интервала находится на нулевой высоте. Другими словами, эту высоту и этот перепад можно назвать относительными величинами.

Перейдем к обработке запроса. Обозначим через *left* и *right* левую и правую границы запроса I . В данном случае они равны двум. Определяем адреса левого и правого сыновей вершины. Если значение u вершины было True, то перебрасываем ее в False, а у сыновей -- в True. При этом не забываем изменить значения h и c у сыновей.

Далее, сравниваем левую границу интервала *left* с длиной интервала у левого сына. Если она меньше или равна длине интервала левого сына, то рекурсивно переходим в левое поддерево, иначе — в правое поддерево. На выходе из рекурсии корректируем значения параметров h и c по принципу:

$c[\text{вершины}] = c[\text{левого сына}] + c[\text{правого сына}];$

$h[\text{вершины}] = h[\text{левого сына}];$

If $h[\text{вершины}] < h[\text{правого сына}] + c[\text{левого сына}]$ Then

$h[\text{вершины}] = h[\text{правого сына}] + c[\text{левого сына}].$

Время обработки запроса пропорционально высоте дерева, т. е. $O(\log_2 I)$. Проверьте обработку следующего запроса — Q 3

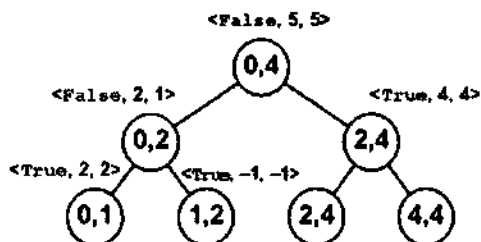


Рис. 17.7. Информация, связанная с вершинами дерева, после обработки запроса I 2 2 -1

из примера в формулировке задачи — две единицы высоты (два рельса) берется из первой вершины левого поддерева корня и одна единица получается при вычислениях по данным из первой вершины правого поддерева корня.

Методическое отступление. Казалось бы, мы уже все или почти все обсудили («долг» по организации дерева интервалов — остался), и можно переходить к программной реализации с тестированием. Однако достичь полного понимания, или, если хотите, «видения» задачи у школьника еще не удастся. Необходимо еще «поплавать на мелководье» — произвести «ручной» счет для конкретных примеров. В этом случае иногда целесообразно использовать следующий прием: дать шаблон, заготовку логики, а затем с его использованием просчитать результат для конкретного примера.

Воспользуемся этим приемом. Пусть дано:

```

8
I 1 8 2
I 3 4 -1
I 6 7 -1

```

и шаблон логики — процедура Reconfig. Требуется прорисовать дерево интервалов с сопутствующей информацией после обработки второго и третьего запросов.

Procedure Reconfig(*root*:<тип данных зависит от реализации дерева интервалов>; *left*, *right*, *change*: **LongInt**); (*left*, *right*, *change* — параметры запроса типа **I**)

Var *childL*, *childR*:<тип данных зависит от реализации дерева интервалов>;

leftLN, *z*:**LongInt**;

Begin

If (*left*=1) **And** (*right*=*t*[*root*]) **Then Begin**

u[*root*]:=True;

c[*root*]:=*t*[*root*]**change*;

h[*root*]:=*t*[*root*]**change*;

Exit; ("Обрываем" дерево — оно состоит из одного интервала)

End;

childL:=<адрес левого сына>;

childR:=<адрес правого сына>;

leftLN:=*t*[*childL*]; (Длина интервала левого сына)

If (*left*<=*leftLN*) **And** (*right*>*leftLN*) **Then Begin**
{Разбиваем интервал (*left*, *right*) на два —
корректируем левое и правое поддеревья}

```

    Reconfig(root, left, leftLN, change);
    Reconfig(root, leftLN+1, right, change);
    Exit;
End;
If (u[root]) Then Begin {Интервал имел одинаковый
наклон}
    u[root]:=False; {Интервал перестает иметь одинаковый
наклон}
    z:=c[root] Div t[root];
    u[childL]:=True;
    c[childL]:=t[childL] * z; h[childL] := c[childL];
    u[childR]:=True;
    c[childR]:=t[childR] * z; h[childR] := c[childR];
End;
If (left <= leftLN) Then
    Reconfig(childL, left, right, change) {Идем
в левое поддерево}
Else
    Reconfig(childR, left-leftLN, right-leftLN, change);
{Идем в правое поддерево}
    c[root]:=c[childL] + c[childR]; {Действия
на выходе из рекурсии}
    h[root]:=h[childL];
    If (h[childR] + c[childL] > h[root]) Then
        h[root]:=h[childR]+c[childL];
End;

```

На наш взгляд, только после того, как ученик уверенно рисует деревья интервалов (для данного примера они приведены на рис. 17.8), можно считать, что задачу он понял.

Закончим обсуждение «возвратом долга» — логикой построения дерева интервалов. При вводе данных следует сформировать массив границ (из нижней границы вычитается единица), например q ; в конце этого массива запишем границы всего интервала, отсортируем элементы массива по неубыванию и уберем повторяющиеся элементы. Допишем в конец массива верхнюю границу интервала столько раз, сколько требуется, чтобы общее количество элементов совпадало с ближайшим числом, являющимся степенью двойки. Массив q для последнего примера после перечисленных действий имеет вид: $q = (0, 2, 4, 5, 7, 8, 8, 8)$. Сформируем «пирамиду» (по принципу пирамидальной сортировки*) из значений длин отрезков. Для этого в конец

* Ожудов С. М. Основы программирования. М.: БИНОМ. Лаборатория знаний, 2004. С. 262–266.

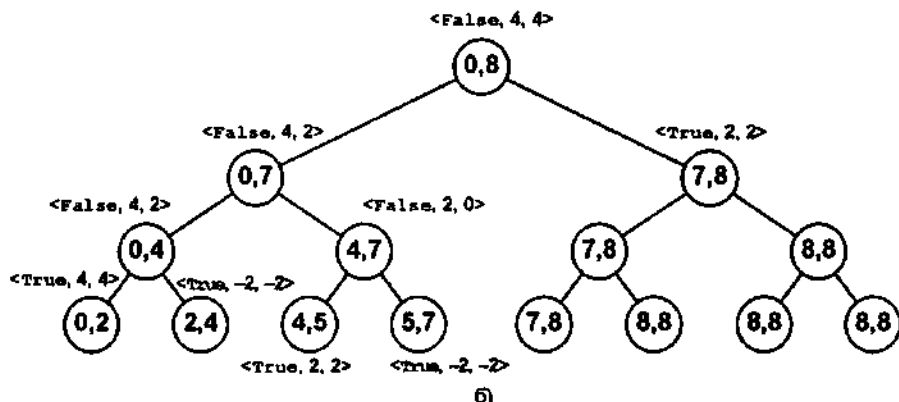
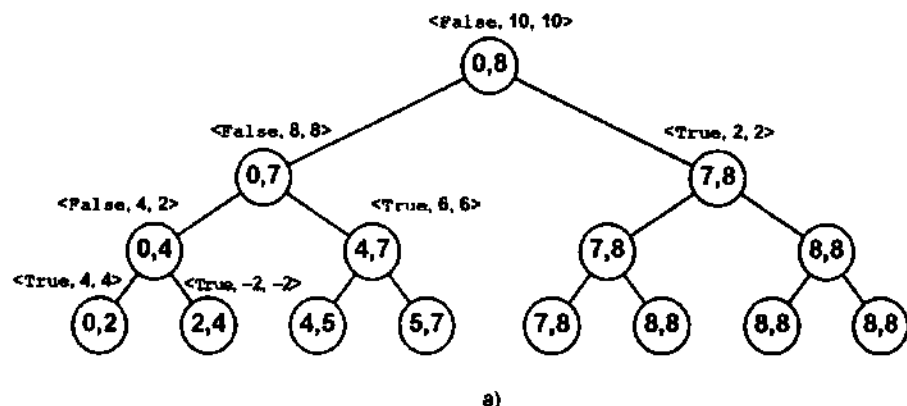


Рис. 17.8. Деревья интервалов и сопутствующая информация после обработки запросов: а) 1 3 4 -1; б) 1 6 7 -1.

Если у вершины дерева значение признака u равно True, то информация, связанная с ее потомками, не приводится

массива t (начиная с восьмого элемента, $k = 8$) запишем длины интервалов, формируемых как разность двух соседних значений из массива q — основание пирамиды есть! «Надстройка» строится с помощью простой рекурсивной логики. Мы не можем отказать себе в удовольствии привести ее:

```
Function fillL(i:LongInt):LongInt; {Первый вызов — fillL(1)}
Begin
  If (i>=k) Then Begin fillL := t[i];Exit;End;
  t[i] := fillL(2*i)+fillL(2*i + 1);
  fillL:=t[i];
End;
```

Массив t , описывающий дерево интервалов после выполнения этой логики, для последнего рассматриваемого примера имеет вид: (8, 7, 1, 4, 3, 1, 2, 2, 1, 2, 1, 0, 0, 0, 0).

Методические рекомендации

Эта задача достаточно сложная. На олимпиадах пока не часто встречались задачи, требующие для своего эффективного решения использовать абстрактный тип данных «двоичное дерево». В принципе, при условии владения (в совершенстве) школьником техникой написания рекурсивных программ, задача может использоваться и при изучении данного раздела практической информатики.

Эффективность работы с задачей определяется не только ее профессиональным обсуждением, но и наличием проверяющей программы и тестов, позволяющих оценить временные характеристики различных решений.

Самостоятельные исследования

1. Решите задачу с использованием типа данных «указатель» (ссылочного типа).
2. Снимем ограничение о том, что все запросы известны заранее. Имеет ли в этом случае задача эффективное решение при тех же ограничениях на входные данные?

ЗАДАЧИ ВТОРОГО ТУРА

«День рождения»

Условие задачи

Сегодня день рождения Байтмена. На празднование дня рождения приглашены n детей (включая самого Байтмена). Все дети пронумерованы числами от 1 до n . Родители Байтмена приготовили большой круглый стол и поставили вокруг него n стульев. Как только дети приходят на день рождения, они рассаживаются за столом. Ребенок с номером 1 занимает одно из мест. Ребенок с номером 2 занимает место слева от ребенка с номером 1. Ребенок с номером 3 занимает следующее за ним место слева и так далее. Наконец, ребенок с номером n займет оставшееся свободное место между детьми с номерами $n-1$ и 1.

Родители Байтмена хорошо знают, что некоторые из приглашенных детей ведут себя весьма шумно за столом, если сидят друг с другом. Поэтому родители собираются пересадить детей в некотором порядке. Этот порядок описывается перестановкой $p_1, p_2, \dots, p_n$ ($p_1, p_2, \dots, p_n$ — различные целые числа от 1 до n).

То есть ребенок p_1 должен сидеть между p_n и p_2 , ребенок p_i ($i = 2, 3, \dots, n-1$) должен сидеть между p_{i-1} и p_{i+1} ; ребенок p_n должен сидеть между p_{n-1} и p_1 .

К сожалению, все дети пришли и расселись так, как указано в первом абзаце. Поэтому для того, чтобы рассадить детей в нужном порядке, родителям придется пересадить кого-то из детей на некоторое количество мест влево или вправо. Для каждого ребенка родителям необходимо решить, как он будет пересаживаться: они должны выбрать направление (влево или вправо) и расстояние (на сколько мест нужно переместиться). Затем по заданному сигналу все дети должны одновременно встать и переместиться на места, определенные родителями Байтмена.

Пересаживание детей вносит беспорядок в празднование дня рождения. Величина беспорядка пропорциональна максимальному из расстояний, на которые дети будут пересажены. Пересаживание детей можно организовать многими способами. Родители решили выбрать тот из них, при котором величина беспорядка будет минимальной, то есть тот способ, при котором максимальное из расстояний пересаживания будет минимальным. Помогите им найти такой способ.

Задание. Ваша задача — написать программу, которая:

- читает из стандартного ввода количество детей и перестановку, определяющую желаемый порядок рассадки детей;
- вычисляет минимально возможную величину беспорядка;
- выводит результат в стандартный вывод.

Входные данные. В первой строке стандартного ввода содержится единственное целое число n ($1 \leq n \leq 1\,000\,000$). Во второй строке содержатся n целых чисел $p_1, p_2, \dots, p_n$, разделенных одним пробелом. Числа $p_1, p_2, \dots, p_n$ образуют перестановку множества $\{1, 2, \dots, n\}$, описывающую желаемый порядок рассадки детей. Кроме того, в 50% тестов число n не будет превышать 1 000.

Выходные данные. В первую и единственную строку стандартного вывода выведите единственное число — минимально возможное из максимальных расстояний, на которые придется пересесть детям.

Пример.

Для входных данных:

6

3 4 5 1 2 6

правильный ответ таков: 2.

На рис. 17.9 слева изображена исходная рассадка детей. На рис. 17.9 в середине показан результат пересаживания, при котором дети с номерами 1 и 2 перемещаются на одно место,

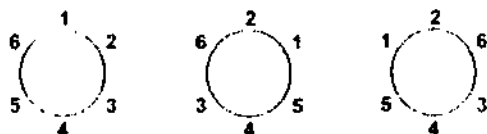


Рис. 17.9. Иллюстрация примера

то, дети с номерами 3 и 5 перемещаются на два места, и дети с номерами 4 и 6 не меняют своего положения. Требуемые условия рассадки выполнены, поскольку 3-й сидит между 6-м и 4-м, 4-й сидит между 3-м и 5-м, 5-й сидит между 4-м и 1-м, 1-й сидит между 5-м и 2-м, 2-й сидит между 1-м и 6-м и 6-й сидит между 2-м и 3-м. Также существует другой вариант конечной рассадки детей, изображенный на рис. 17.9 справа. В обоих вариантах величина беспорядка равна 2.

Обсуждение задачи

Проанализируем рис. 17.9, исходную единичную перестановку и одну из результирующих (скажем, последнюю). Запишем перестановку из входных данных по кругу так, как это показано на рис. 17.10. Результирующая перестановка получается из показанной на рис. 17.10 путем вращения по кругу на две позиции.

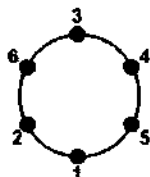


Рис. 17.10. Иллюстрация входных данных

Итак, первый вариант решения — вращаем (по часовой стрелке) и для каждого состояния находим максимум расстояний элементов от единичной перестановки. Количество вращений — $n-1$, время для нахождения максимума пропорционально $O(n)$. Общая временная сложность алгоритма — $O(n^2)$. Случай вращения против часовой стрелки рассматривается аналогично. Очевидно, что 10^{12} действий (при $n = 10^6$) за 2 с (время прохождения одного теста) не выполнить — требуется искать другое решение.

Возьмем более сложный пример. Пусть $n = 15$ и перестановка, представляющая желаемую рассадку детей, имеет вид: 10, 11, 12, 13, 3, 1, 2, 15, 4, 9, 14, 5, 7, 8, 6. В этом случае результат не так очевиден, но все же можно просчитать его «вручную», и процесс счета «наталкивает» на идеи, позволяющие получить результат (*важное методическое замечание*).

Возьмем первый элемент перестановки — 10. Чтобы 10 оказалось на первом месте, его требуется сдвинуть на 6 позиций из единичной перестановки по часовой стрелке и на 9 позиций против часовой стрелки. Элементу массива $d[1]$ присваиваем

значение 6 — минимальное значение сдвига. Элемент 15 находится на 8-м месте. По часовой стрелке, при получении из единичной перестановки, требуется 8 сдвигов, против часовой — 7 сдвигов, $d[8] = -7$. Окончательный вид массива d : (6, 6, 6, 6, 2, 5, 5, -7, 5, 1, -3, 7, 6, 6, -6). Пока ничего не изменилось, мы идем по первому пути решения задачи. Нанесем эти сдвиги на окружность так, как это показано на рис. 17.11, и выделим значения, встречающиеся в массиве d .

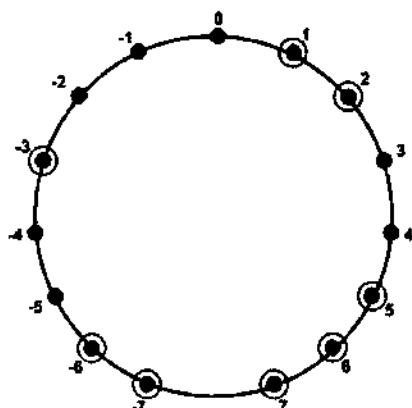


Рис. 17.11. Иллюстрация первого примера в обсуждении

Пока идея решения не просматривается. Сдвинем перестановку 10, 11, 12, 13, 3, 1, 2, 15, 4, 9, 14, 5, 7, 8, 6 на две позиции (например) по часовой стрелке. Получаем 8, 6, 10, 11, 12, 13, 3, 1, 2, 15, 4, 9, 14, 5, 7. Сформируем массив d : (-7, -4, -7, -7, -7, -7, 4, 7, 7, -5, 7, 3, -1, -6, -7). На рис. 17.12 представлено графическое изображение значений сдвигов. Искомая перестановка имеет вид 13, 3, 1, 2, 15, 4, 9, 14, 5, 7, 8, 6, 10, 11, 12, а массив d для нее — (3, -1, 2, 2, 5, 2, -2, -6, 4, 3, 3, 6, 3, 3, 3). Его графическое изображение представлено на рис. 17.13.

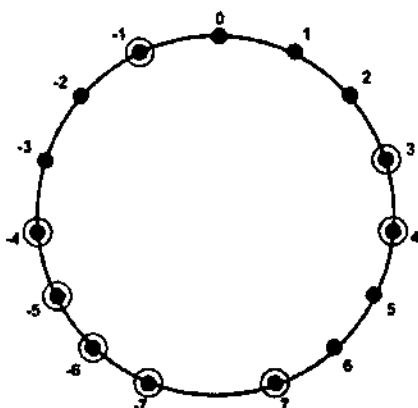


Рис. 17.12. Иллюстрация второго примера в обсуждении

Зададим себе вопрос: что общего есть в рис. 17.11, 17.12 и 17.13? Если мы на него ответим, то сможем по первому варианту массива d вычислить значение сдвига («величину беспорядка», как сказано в условии задачи). Массив d формируется за время $O(n)$, и, вы-

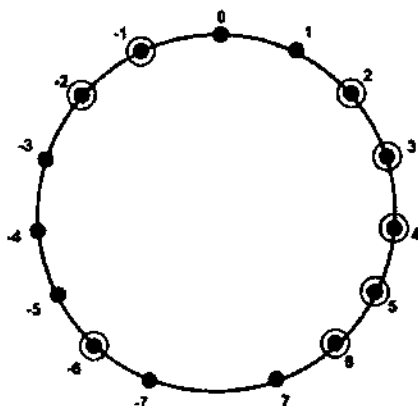


Рис. 17.13. Иллюстрация искомой перестановки в примере из обсуждения

числив по нему значение беспорядка, мы исключаем дальнейшие сдвиги — в результате временная сложность алгоритма становится равной $O(n)$. Структура (взаимное расположение) выделенных точек на окружности остается неизменной. Это и очевидно — структура перестановки не изменяется при выполнении циклических сдвигов! Следовательно, нам необходимо вычислить «длину дуги», покрывающей все выделенные точки. Половина этой длины и будет ответом в задаче. Для рассматриваемого примера она равна $(15-3)/2 = 6$, т. е. величина беспорядка равна 6.

Методические рекомендации

Эту задачу можно отнести к теме «Комбинаторика», но знаний каких-либо специальных алгоритмов, например вычисления по перестановке ее номера в лексикографическом порядке и т. д., она не требует. Решение через генерацию всех перестановок не оптимально по времени. Главное в процессе обсуждения задачи — «натолкнуть» школьника на мысль о существовании некоего инварианта относительно сдвигов перестановки и связи этого инварианта с искомым ответом задачи.

Самостоятельные исследования

1. Данная задача идеально подходит для анализа времени решения задачи с помощью алгоритмов с временными сложностями $O(n^2)$ и $O(n)$. Генератор входных данных по значению n пишется просто — случайным образом формируется перестановка порядка n . Оценка времени работы алгоритма выполняется с использованием средств используемой системы программирования. Далее следует экспериментальным путем установить, для каких значений n задача решается за 2 секунды тем и другим алгоритмом.
2. Задача об инверсивном векторе перестановки* может быть использована как для самостоятельной работы школьника по теме, так и при проведении занятия. Ее можно сформулировать следующим образом. Имеется перестановка $P = \{p_i\}$ чисел $1, 2, \dots, n$, где $1 \leq n \leq 1000000$. Для каждой перестановки P существует последовательность $T = \{t_i\}$, в которой t_i равно количеству чисел, больших i и расположенных левее положения числа i в $P = \{p_i\}$. Требуется по заданной последовательности $T = \{t_i\}$ найти $P = \{p_i\}$.

* Липский В. Комбинаторика для программистов. М.: Мир, 1988. С. 77.

«Игра с прямоугольником»

Условие задачи

Рассмотрим игру для двух игроков. Игрокам дан прямоугольник размером $x \times y$ (где x и y — положительные целые числа). Игроки ходят по очереди. Ход состоит в разделении прямоугольника на два прямоугольника одним горизонтальным или вертикальным разрезом. Полученные прямоугольники должны иметь положительные целочисленные размеры. На рис. 17.14 показаны возможные разрезы прямоугольника 4×3 .

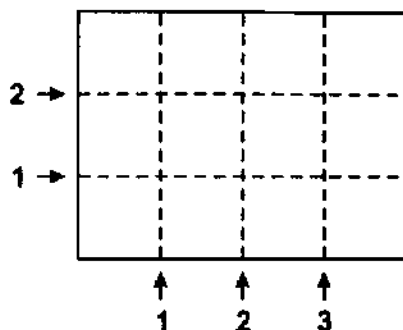


Рис. 17.14. Возможные разрезы прямоугольника размером 4×3

После каждого разреза прямоугольник с меньшей площадью убирается, а оставшийся передается другому игроку. Если прямоугольник разделен на две равные части, то убирается одна из них. Игрок, который получает прямоугольник размером 1×1 , проигрывает, так как не может сделать следующий ход.

Ваша задача — написать программу, которая бы играла в игру с прямоугольниками и выигрывала. Для того, чтобы играть, программа должна использовать специальную библиотеку. В библиотеке есть функции `dimension_x()` и `dimension_y()`, возвращающие размеры прямоугольника. Начальные размеры прямоугольника — целые числа от 1 до 100 000 000. Как минимум один из размеров больше 1. К тому же в 50% тестов размеры прямоугольника не будут превышать 25.

В библиотеке есть также процедура `cut(dir, position)`, которая должна вызываться вашей программой, чтобы сделать ход. Параметры `dir` и `position` описывают направление и позицию разреза, соответственно. Параметр `dir` может принимать одно из двух значений: *vertical* и *horizontal*. Если `dir = vertical`, то проводится вертикальный разрез, а параметр `position` указывает x -координату разреза, как показано на рисунке выше. При этом вы должны гарантировать выполнение неравенства $1 \leq position \leq dimension\_x() - 1$. Если `dir = horizontal`, то проводится горизонтальный разрез, а параметр `position` указывает y -координату разреза. При этом вы должны гарантировать выполнение неравенства $1 \leq position \leq dimension\_y() - 1$.

После запуска вашей программы она будет играть за одного из игроков. Ваша программа ходит первой, она должна разре-

зать исходный прямоугольник. Когда ваша программа вызывает процедуру `cut`, ваш ход записывается и управление передается программе соперника. После хода соперника управление возвращается вашей программе. Значения, которые возвращаются функциями `dimension_x()` и `dimension_y()`, будут отражать результат вашего хода и хода соперника. Как только ваша программа выигрывает, проигрывает или делает неправильный ход, ее исполнение будет прервано. Прерывание вашей программы — это автоматический процесс, так что ваша программа должна продолжать делать столько ходов, сколько возможно, до автоматического прерывания ее исполнения. Вы можете предполагать, что для предложенных входных данных всегда существует выигрышная стратегия для вашей программы.

Ваша программа не должна читать какие-либо файлы или писать в них, не должна использовать стандартный ввод и вывод и не должна пытаться изменять содержимое какой-либо памяти вне вашей программы. Нарушение любого из этих правил может привести к дисквалификации.

Экспериментирование. Для того, чтобы дать вам возможность поэкспериментировать с библиотекой, в ваше распоряжение предоставлены примеры библиотек: их исходные тексты находятся в файлах `preclib.pas`, `creclib.c` и `creclib.h`. Эти библиотеки вы можете взять по адресу: <http://contest/>. Они реализуют очень простую стратегию. Когда вы запустите вашу программу, она будет играть против этих простых соперников. Вы можете изменять их, чтобы протестировать вашу программу с лучшими соперниками. Следует учесть, что во время тестирования после окончания тура ваша программа будет играть против другого соперника.

При посылке вашей программы на проверку с использованием интерфейса `TEST` она будет откомпилирована с немодифицированной библиотекой соперника. При этом посылаемый вами входной файл будет передан как стандартный ввод вашей программе. Входной файл должен состоять из двух строк, каждая из которых должна содержать по одному целому числу. Первая строка должна содержать исходную ширину, вторая — исходную высоту прямоугольника. Эти размеры будут прочитаны библиотекой, предложенной в качестве примера.

Если вы модифицируете часть `implementation` библиотеки `preclib.pas`, пожалуйста, перекомпилируйте ее, используя команду `ppc386 -O2 preclib.pas`. Эта команда создаст файлы `preclib.o` и `preclib.ppu`. Эти файлы необходимы для компиляции вашей программы и должны быть помещены в каталог, где находится ваша программа. Пожалуйста, не модифицируйте часть `interface` библиотеки `preclib.pas`.

Если вы модифицируете библиотеку `creclib.c`, пожалуйста, не забудьте поместить ее вместе с `creclib.h` в каталог, где находится ваша программа, — они необходимы для компиляции. Пожалуйста, не модифицируйте файл `creclib.h`.

В ваше распоряжение также предоставляются две простые программы, которые иллюстрируют использование библиотек `cres.c` и `pres.pas`. Пожалуйста, помните, что эти программы не являются правильными решениями. Вы можете откомпилировать их, используя такие команды:

```
gcc -O2 -static cres.c creclib.c -lm
g++ -O2 -static cres.c creclib.c -lm
ppc386 -O2 -XS pres.pas
```

Библиотеки. В ваше распоряжение предоставлены библиотеки, которые обеспечивают следующую функциональность.

- Библиотека для *FreePascal* (`preclib.ppu`, `preclib.o`):

```
type direction = (vertical, horizontal);
function dimension_x(): longint;
function dimension_y(): longint;
procedure cut(dir: direction; position: longint);
```

Включите следующий оператор в ваш исходный файл `rec.pas`:

```
uses preclib;
```

Чтобы откомпилировать вашу программу, скопируйте файлы `preclib.o` и `reclib.ppu` в каталог, где находится ваш исходный файл, и выполните следующую команду:

```
ppc386 -O2 -XS rec.pas
```

Файл `pres.pas` является примером использования библиотеки `preclib`.

- Библиотека для *GNU C/C++* (`creclib.h`, `creclib.c`):

```
typedef enum __direction {vertical, horizontal}
direction;
int dimension_x();
int dimension_y();
void cut(direction dir, int position);
```

Включите следующий оператор в ваш исходный файл (`rec.c` или `rec.cpp`):

```
#include "creclib.h"
```

Чтобы откомпилировать вашу программу, скопируйте файлы `creclib.c` и `creclib.h` в каталог, где находится ваш исходный файл, и выполните следующую команду:

```
gcc -O2 -static rec.c creclib.c -lm
```

или

```
g++ -O2 -static rec.cpp creclib.c -lm
```

Файл `rec.c` является примером использования библиотеки в C.

Пример взаимодействия. В табл. 17.1 приведен пример взаимодействия вашей программы с библиотекой. Он показывает, как может проходить игра. Игра начинается с прямоугольника размером 4×3 . Существует выигрышная стратегия для этого случая.

Таблица 17.1

Ваша программа вызывает:	Что происходит:
<code>dimension_x()</code>	возвращает 4
<code>dimension_y()</code>	возвращает 3
<code>cut(vertical, 1)</code>	ваш разрез записывается, и прямоугольник размером 3×3 передается вашему сопернику, который разрезает его, и получается прямоугольник размером 3×2 ; после этого управление передается вашей программе
<code>dimension_x()</code>	возвращает 3
<code>dimension_y()</code>	возвращает 2
<code>cut(horizontal, 1)</code>	ваш разрез записывается, и прямоугольник размером 3×1 передается вашему сопернику, который разрезает его и получается прямоугольник размером 2×1 ; после этого управление передается вашей программе
<code>dimension_x()</code>	возвращает 2
<code>dimension_y()</code>	возвращает 1
<code>cut(vertical, 1)</code>	в результате вашего разреза получается прямоугольник размером 1×1 , так что вы выиграли; после этого работа вашей программы автоматически прекращается

Обсуждение задачи

Поиск выигрышной стратегии начинается с анализа возможных значений размеров прямоугольника $x \times y$. Достаточно быстро определить, что в том случае, когда прямоугольник является квадратом ($x = y$), мы проигрываем, делая первый ход, при любом выборе отсекаемого прямоугольника. Действительно, первым ходом мы переводим квадрат в прямоугольник размера $x \times t$, где $x/2 \leq t < x$. Противник (назовем его «компьютером») делает ход, переводящий прямоугольник $x \times t$ в квадрат $t \times t$, и так отвечает на каждый наш следующий ход, пока перед нами не останется квадрат 1×1 .

Естественным образом возникает вопрос о том, является ли данное условие на размер исходного прямоугольника единственным, при котором мы оказываемся в проигрыше. Другими словами, нам требуется выделить те прямоугольники, которые являются проигрышными. А нашей выигрышной стратегией будет сведение исходного прямоугольника (если он изначально не проигрышный) к проигрышному ближайшему прямоугольнику и затем, после каждого хода компьютера, поддержка этой стратегии. Будем считать для определенности, что $x \leq y$. Схема сведения случаев $x > y$ к рассматриваемым прямоугольникам очевидна.

Пусть $x = 1$. Попытаемся найти проигрышные прямоугольники. На рис. 17.15 показаны результаты поиска. Заштрихованные прямоугольники — это проигрышные для нас позиции (1×1 , 1×3 , 1×7 , 1×15 , 1×31 , ...). Так, если $4 \leq y \leq 6$, то первым ходом мы переводим его в прямоугольник 1×3 . При $8 \leq y \leq 14$ — переводим в прямоугольник 1×7 и т. д.

Казалось бы, зависимость найдена — $y = 2^k \times x - 1$, при $k > 0$. Однако не будем торопиться. Проанализируем случай $x = 2$. Результаты анализа показаны на рис. 17.16. Проигрышные прямоугольники 2×2 , 2×5 , 2×11 , 2×23 , 2×47 , ...

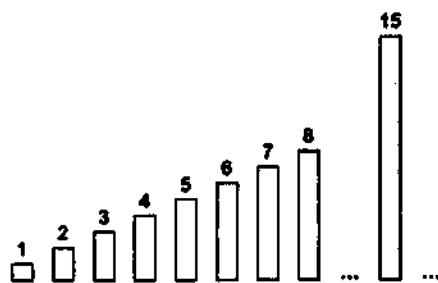


Рис. 17.15. Поиск проигрышных позиций при $x = 1$

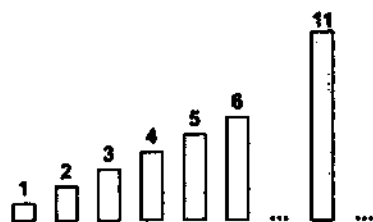


Рис. 17.16. Поиск проигрышных позиций при $x = 2$

Рассмотрим прямоугольник 2×5 . При любом нашем ходе у компьютера есть возможность перевода прямоугольника в один из прямоугольников 1×3 или 2×2 , а они для нас — проигрышные!

Мы видим, что найденная зависимость нас не удовлетворяет. Если ответ (зависимость) пока еще «не просматривается», то анализ случая $x = 3$ дает прямоугольники 3×3 , 3×7 , 3×15 , 3×31 , 3×63 и т. д. При $x = 4$ — прямоугольники 4×4 , 4×9 , 4×19 , 4×39 , 4×79 .

Вернемся к наброску зависимости и сделаем так, чтобы она работала при $k = 0$. Получаем: $y = 2^k(x+1)-1$ и видим, что данная зависимость работает и при $x > 1$.

Оставим формальное обоснование «на потом», а сейчас попытаемся осознать — что мы сделали? На множестве всех возможных прямоугольников (на множестве всех исходных данных) мы определили некие прямоугольники (некие точки), определяющие нечто типа отношения порядка. Это позволяет нам определить «близость» прямоугольников между собой, ибо для каждого прямоугольника мы обязаны уметь находить ближайший к нему проигрышный прямоугольник.

Пусть числа x и y таковы, что не существует целого k , при котором $y = 2^k(x+1)-1$. Число $\log_2 \left(\frac{y+1}{x+1} \right)$ в этом случае не является

целым. Возьмем $t = 2^{\left\lfloor \log_2 \left(\frac{y+1}{x+1} \right) \right\rfloor} (x+1) - 1$. Значение $t \leq y/2$ и $t < y$. Следовательно, в результате нашего первого хода получается прямоугольник $t \times x$, являющийся проигрышным для компьютера.

Утверждение. Прямоугольники (x, y) , при $y = 2^k(x+1)-1$ для некоторого k являются проигрышными.

Обоснование. Рассуждение носит индуктивный характер. База индукции — $k = 0$ (квадраты). При $x = 1$ позиция проигрышная по условию задачи. При $x > 1$ мы делаем ход, такой, что получается прямоугольник (t, y) , где $x/2 \leq t < x$. Компьютер делает ход, переводящий прямоугольник (t, y) в квадрат (t, t) , а он для нас — проигрышный.

Пусть $k \neq 0$, $y = 2^k(x+1)-1$ и для всех меньших значений k утверждение справедливо. У нас две возможности — отсекать прямоугольник по оси y или по оси x .

Первый случай. Мы преобразуем своим ходом исходный вариант в прямоугольник (t, x) , где $y/2 \leq t < y$. Подставляя значение y в неравенства, получаем: $2^{k-1}(x+1)-1 < t < 2^k(x+1)-1$. У компьютера есть возможность ответить ходом, в результате которого получается прямоугольник $(2^{k-1}(x+1)-1, x)$, а он, согласно индуктивной схеме, проигрышный.

Второй случай. Мы сделали ход (y, t) , где $x/2 \leq t < x$. Требуется показать, что эта позиция не является проигрышной для компьютера, а значит, он имеет возможность перевести ее вновь в позицию, проигрышную для нас. Предположим обратное: $y = 2^i(t+1)-1$ для некоторого целого i , то есть позиция проигрышная для компьютера. Так как $y = 2^i(t+1)-1 < 2^i(x+1)-1$, то $i > k$. Однако $x/2 = t$, следовательно, $y = 2^i(t+1)-1 = 2^i(x/2+1)-1 = 2^{i-1}(x+2)-1 > 2^{i-1}(x+1)-1$. Итак, $i-1 < k$, то есть $i < k+1$. Противоречие!

Методические рекомендации

«Найти» мысль о том, какие прямоугольники являются проигрышными, путем доказательства утверждений сложно. Доказательствами можно только обосновать то, что найденное решение верно. Через упрощение задачи и прорисовку вариантов ее решения определяются начальные варианты решения.

Самостоятельные исследования

Задача естественным образом обобщается для трехмерного случая. Отсекаем от первоначального параллелепипеда поочередно плоскостью параллелепипеды меньшего объема до тех пор, пока не останется единичный куб.

«Реки»

Условия задачи

Почти все Королевство Байтленд покрыто лесами и реками. Малые реки сливаются в более крупные реки, которые, в свою очередь, сливаются друг с другом; в конечном счете, все реки сливаются вместе в одну большую реку. Большая река впадает в море вблизи города Байттаун.

В Байтленде имеется n лесозаготовительных поселков, каждый из которых расположен вблизи какой-либо реки. В настоящее время в Байттауне находится большая пилорама, которая обрабатывает все деревья, срубленные в Королевстве. Деревья сплавляются вниз по рекам от поселков, где они срублены, к пилораме в Байттауне. Король Байтленда решил поставить k дополнительных пилорам в поселках, чтобы уменьшить стоимость сплава деревьев. После установки пилорам деревья не обязательно должны сплавляться в Байттаун, а могут быть обработаны на ближайшей пилораме, находящейся ниже по течению рек. Очевидно, что деревья, срубленные в окрестности поселка с пилорамой, вообще не сплавляются по рекам.

Необходимо отметить, что реки в Байтленде не разветвляются. Из этого следует, что для каждого поселка существует един-

ственный путь сплава деревьев вниз по течению рек от него в Байттаун.

Королевские счетоводы подсчитали количество деревьев, срубаемых в каждом поселке за год. Вам необходимо определить, в каких поселках следует установить пилорамы, чтобы минимизировать общую стоимость сплава деревьев за год. Стоимость сплава одного дерева составляет один цент за каждый километр пути.

Задание. Напишите программу, которая:

- читает из стандартного ввода количество поселков, количество дополнительных пилорам, которые будут установлены, количество срубленных в каждом поселке деревьев и описание рек;
- вычисляет минимальную стоимость сплава деревьев после установки дополнительных пилорам;
- выводит результат в стандартный вывод.

Входные данные. Первая строка входных данных содержит два целых числа: n — количество поселков, не считая Байттауна ($2 \leq n \leq 100$), и k — количество дополнительных пилорам, которые будут установлены ($1 \leq k \leq 50$ и $k \leq n$). Поселки нумеруются числами 1, 2, ..., n , а Байттаун имеет номер 0.

Каждая из последующих n строк содержит три целых числа, разделенных одним пробелом. Строка $i+1$ содержит:

- w_i — количество деревьев, срубаемых в поселке i за год ($0 \leq w_i \leq 10\,000$);
- v_i — ближайший поселок (либо Байттаун) вниз по реке от поселка i ($0 \leq v_i \leq n$);
- d_i — расстояние (в километрах) по реке от поселка i до поселка v_i ($1 \leq d_i \leq 10\,000$).

Гарантируется, что суммарная стоимость сплава всех деревьев к пилораме в Байттауне не превосходит 2 000 000 000 центов в год. В 50% тестов число n не превосходит 20.

Выходные данные. Первая и единственная строка выходных данных должна содержать одно целое число: минимальную стоимость сплава (в центах).

Пример.

Для входных данных:

```
4 2
1 0 1
1 1 10
10 2 5
1 2 3
```

правильным результатом будет 4.

Рис. 17.17 иллюстрирует входные данные примера. Номера поселков указаны внутри кругов. Числа под кругами обозначают количество деревьев, срубаемых вблизи данного поселка. Числа над стрелками указывают длины рек.

Пилорамы должны быть установлены в поселках 2 и 3.

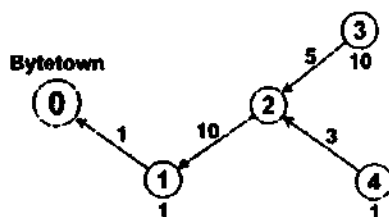


Рис. 17.17. Иллюстрация примера в условии

Обсуждение задачи

Упростим пример и научимся решать задачу для него. Пусть входные данные имеют вид:

```

4 1
2 0 1
6 1 4
3 2 5
5 3 7

```

Практически здесь у нас одна река и на ней расположены поселки (рис. 17.18). Пусть нам пока не требуется размещать пилораму. Подсчитаем стоимость сплава и зафиксируем результат в матрице Q^0 (нумерация строк и столбцов начинается с нуля). Индекс «0» означает отсутствие пилорам.



Рис. 17.18. Графическое изображение данных для простого примера

$$Q^0 = \begin{pmatrix} 147 & \infty & \infty & \infty & \infty \\ 147 & \infty & \infty & \infty & \infty \\ 145 & 131 & \infty & \infty & \infty \\ 115 & 107 & 75 & \infty & \infty \\ 85 & 80 & 60 & 35 & \infty \end{pmatrix}.$$

Элемент $Q^0[i, j]$ при этом определяет стоимость сплава леса из всех поселков, начиная с номера i и ниже по течению, в поселок с номером j . Подсчет начинается с четвертого поселка, т. е. $Q^0[i, j] = Q^0[i+1, j] + \text{стоимость сплава деревьев из поселка } i \text{ в поселок } j$ при $i < 4$. Так, $Q^0[3, 2] = Q^0[4, 2] + \text{стоимость сплава деревьев из поселка 3 в поселок 2} = 60 + 15 = 75$.

Попробуем разместить одну пилораму — матрица Q^1 имеет вид:

$$Q^1 = \begin{pmatrix} 62 & \infty & \infty & \infty & \infty \\ 62 & \infty & \infty & \infty & \infty \\ 60 & 51 & \infty & \infty & \infty \\ 30 & 27 & 15 & \infty & \infty \\ 0 & 0 & 0 & 0 & \infty \end{pmatrix}.$$

Как мы ее получили? Очевидно, что элементы $Q^1[4, 0]$, $Q^1[4, 1]$, $Q^1[4, 2]$, $Q^1[4, 3]$ равны нулю — пилорама размещена в четвертом поселке, и сплавлять из него в поселки ниже по течению нечего.

Рассмотрим элемент $Q^1[3, 2]$. Пилораму можно разместить или в четвертом поселке, или в третьем. В первом случае стоимость сплава в поселок 2 равна 15, во втором случае 35, — мы выбираем минимальное значение.

Элемент $Q^1[3, 1]$. Если мы размещаем пилораму в четвертом поселке, то затраты на сплав до первого поселка равны 27. А при размещении пилорамы в третьем поселке из четвертого — требуется сплав в третий, затраты — 35.

Очевидно, элемент $Q^1[i, j] = \text{Min}(Q^0[i+1, j]; Q^1[i+1, j] + \text{стоимость сплава из } i \text{ в } j)$.

Таким образом, что такое $Q^k[i, j]$? Это минимальные затраты при размещении k пилорам в поселках, начиная с i и находящихся выше по течению, при условии, что ближайшая пилорама ниже по течению находится в поселке j . Ответом для данного примера является значение 62, и пилораму следует разместить в четвертом поселке. А для случая $k = 2$? Ответ — 17, и пилорамы следует разместить во втором и четвертом поселках.

Методическое отступление. В этом месте обсуждения лучше сделать паузу и предложить просчитать ответ «вручную» для нескольких наборов входных данных.

Пример 1:

```

4 1 (2)
2 0 1
6 1 4
3 2 5
5 3 1
```

$$Q^1 = \begin{pmatrix} 37 & \infty & \infty & \infty & \infty \\ 37 & \infty & \infty & \infty & \infty \\ 35 & 29 & \infty & \infty & \infty \\ 5 & 5 & 5 & \infty & \infty \\ 0 & 0 & 0 & 0 & \infty \end{pmatrix}.$$

Ответ — 37, третий поселок.

$$Q^2 = \begin{pmatrix} 7 & \infty & \infty & \infty \\ 7 & \infty & \infty & \infty \\ 5 & 5 & \infty & \infty \\ 0 & 0 & 0 & \infty \end{pmatrix}.$$

Ответ — 7, второй и третий поселки.

Пример 2:

4 1
2 0 1
6 1 4
3 2 2
5 3 1

$$Q^1 = \begin{pmatrix} 23 & \infty & \infty & \infty & \infty \\ 23 & \infty & \infty & \infty & \infty \\ 21 & 21 & \infty & \infty & \infty \\ 5 & 5 & 5 & \infty & \infty \\ 0 & 0 & 0 & 0 & \infty \end{pmatrix}.$$

Ответ — 23, пилораму следует разместить во втором поселке.

А теперь, наоборот, усложним пример для продуктивности дальнейшего обсуждения задачи. Пусть входные данные имеют вид:

12 1
1 0 1
1 1 10
10 2 5
1 2 3
2 1 5
5 9 3
4 9 4
3 9 5
1 1 8
2 3 7
6 9 2
7 9 1

Графическое изображение данных для этого примера приведено на рис. 17.19. Требуется найти местонахождение одной пилорамы в 12 поселках, оптимизирующее общие затраты на сплав леса. То, что мы нарисовали, носит название «*дерев*», его корнем является вершина с номером 0 (Байттаун). Кружками обозначены вершины дерева, а направленными отрезками — его ребра.

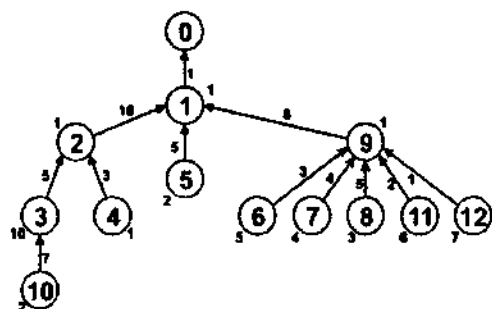


Рис. 17.19. Графическое изображение данных для второго примера

Как представить в памяти компьютера связи дерева? Если школьник знает ссылочный тип данных (работу с указателями) и изучал тему «Абстрактные типы данных» (в частности, списки), то в этом случае проблема решается за счет их использования, где каждой вершине дерева соответствует список вершин, являющихся ее потомками. Возможно описание и с помощью матрицы (C). Строка матрицы i соответствует вершине с номером i , значение элемента $C[i, 0]$ определяет количество потомков, а затем в этой строке перечисляются номера вершин, соответствующих потомкам вершины i .

Дерево на рис. 17.19 имеет некий «люфт», а именно, у вершин дерева, кроме конечных — «листьев», может существовать любое число потомков в пределах заданного значения n . Введение более «жесткого» отношения порядка на множестве исходных данных, как правило, упрощает решение. При этом, естественно, новое отношение порядка не должно изменять содержательного наполнения задачи.

В данном конкретном случае за счет введения дополнительных вершин с нулевыми параметрами ($w_i = 0$, $d_i = 0$ для всех новых значений i) исходная структура преобразуется в *двоичное дерево* — каждая вершина (не «лист»), имеет ровно два потомка. Вид дерева после преобразования показан на рис. 17.20.

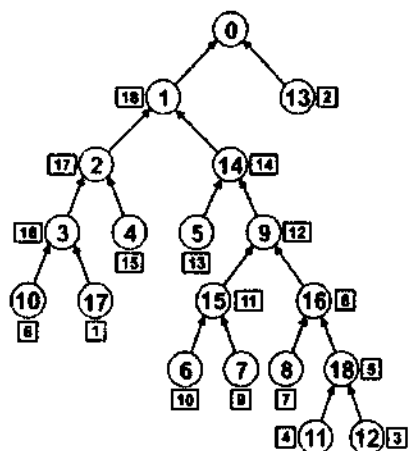


Рис. 17.20. Вид дерева, представленного на рис. 17.19, после его преобразования

Как мы это делаем? Просматриваем строки матрицы C и если они не удовлетворяют условию — два потомка у соответствующей вершины, — то добавляем новую строку (новую вершину) и часть данных из исходной строки переписываем в новую строку. Сравнение деревьев на рис. 17.19 и 17.20 дает представление об этой логике. Массив v для преобразованного дерева имеет вид: (0, 1, 2, 2, 14, 15, 15, 16, 14, 3, 18, 18, 0, 1, 9, 9, 3, 16).

Методическое отступление. В этом месте обсуждения задачи уместен вопрос: как по значению новых номеров вершин дерева определить структуру данных, используемую в преобразовании?

Далее, в первых простых примерах номер поселка, расположенного ниже по течению, был всегда меньше номеров поселков, находящихся выше по течению. Это условие облегчало решение задачи. Рассматривая поселок с номером i , мы знали, что для всех j ($j > i$) оценки уже вычислены. В данном же случае требуется выполнить *топологическую сортировку** вершин дерева. При этом изменять существующие структуры данных не следует, а можно просто зафиксировать в новом массиве (например, s) очередность просмотра вершин дерева при их обработке так, чтобы при обработке любой вершины с номером i оценки для ее потомков уже были сделаны. Для рассматриваемого примера $s = (17, 13, 12, 11, 18, 10, 8, 16, 7, 6, 15, 9, 5, 14, 4, 3, 2, 1)$. Эта очередность показана на рис. 17.20 в небольших квадратах у каждой вершины дерева.

Оценим ситуацию для вершины с номером 18 (рис. 17.20), — она пятая по очередности в нашем рассмотрении. К этому моменту времени, в частности (мы приводим оценки только до Байттауна), получены оценки: $Q^0[11, 0] = 66$, $Q^1[11, 0] = 0$, $Q^0[12, 0] = 70$, $Q^1[12, 0] = 0$. Оценка $Q^0[18, 0] = 136$ очевидна. При отсутствии пилорам в поддереве с корнем 18 и нахождении ближайшей пилорамы в Байттауне общая стоимость сплава равна стоимости сплава из левого и правого поддеревьев, плюс стоимость сплава из корня. При получении оценки $Q^1[18, 0]$ следует рассмотреть оценки размещения по одной пилораме в левом и правом поддеревах и выбрать минимальное значение. Оно равно 66 — пилораму следует разместить в двенадцатом поселке.

* Окулов С. М. Программирование в алгоритмах. М.: БИНОМ. Лаборатория знаний, 2004. С. 164–166. (Топологическая сортировка использована в алгоритме поиска кратчайших путей в бесконтурном графе.)