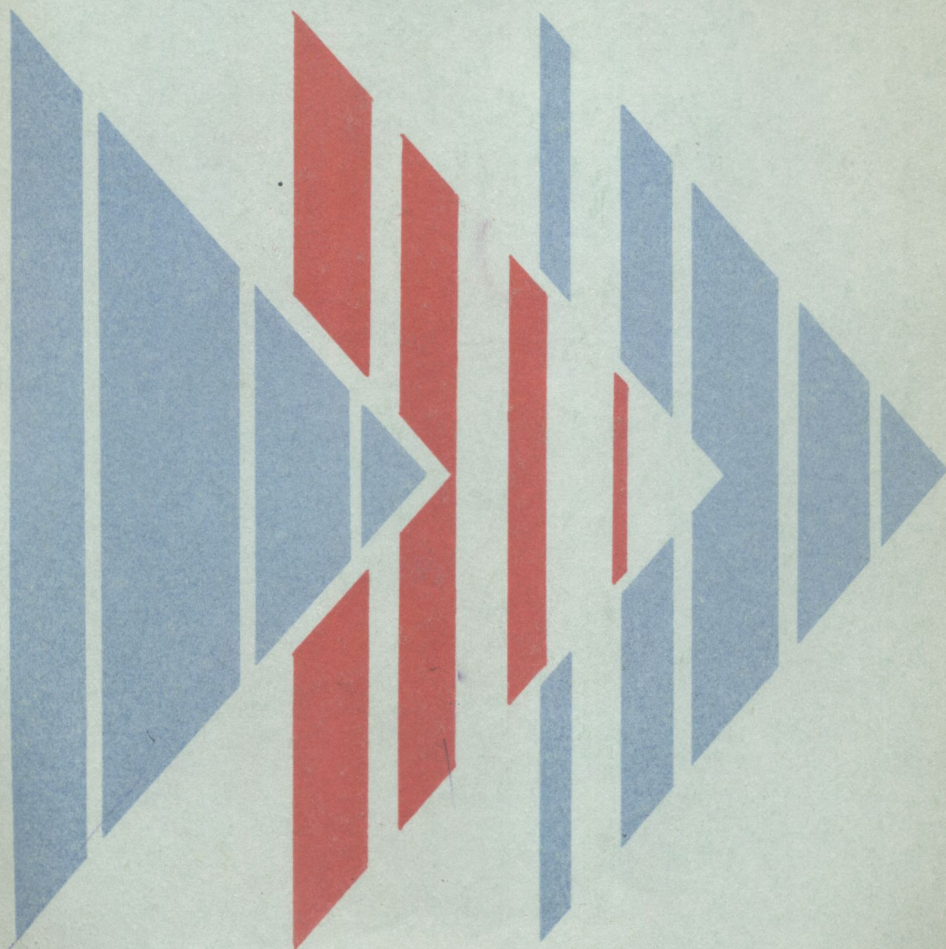


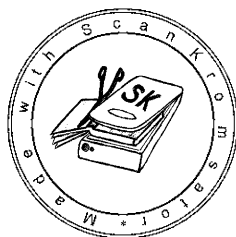
Д. Бентли

# ЖЕМЧУЖИНЫ ТВОРЧЕСТВА ПРОГРАММИСТОВ

Д. Бентли ЖЕМЧУЖИНЫ ТВОРЧЕСТВА ПРОГРАММИСТОВ



**Д. Бентли**  
**ЖЕМЧУЖИНЫ**  
**ТВОРЧЕСТВА**  
**ПРОГРАММИСТОВ**



Scan AAW

# Programming Pearls

**JON BENTLEY**

AT&T Bell Laboratories  
Murray Hill, New Jersey



**ADDISON-WESLEY PUBLISHING COMPANY**

**Д. Бентли**

**Жемчужины  
творчества  
программистов**

Перевод с английского М. Г. Логунова  
Под редакцией И. Г. Шестакова



Москва  
«Радио и связь»  
1990

ББК 32.973  
Б 46  
УДК 681.3.06

**Редакция переводной литературы**

**Бентли Д.**

**Б 46 Жемчужины творчества программистов: Пер. с англ. – М.: Радио и связь, 1990. – 224 с.: ил.**

**ISBN 5-256-00704-1.**

В книге американского автора на различных примерах из практики программирования показано, как хорошее понимание особенностей поставленной задачи позволяет найти оптимальное по быстродействию, объему требуемой памяти, легкости модификации решения. Наряду с конкретными примерами даны общие рекомендации по составлению оптимальных алгоритмов и программ. Рассмотрение построено по следующему принципу: постановка задачи, пример традиционного решения и объяснение его недостатков, углубленный анализ задачи и найденное в результате этого лучшее решение, изложение ряда принципов грамотного программирования.

Для программистов.

**Б 2404010000-065**  
**046 (01)-90**

**138-90**

**ББК 32.973**

Производственное издание

**БЕНТЛИ ДЖОН**

**Жемчужины творчества программистов**

Заведующий редакцией Ю. Г. Ивашов

Редактор М. Г. Коробочкина

Обложка художника Б. И. Николашина

Художественный редактор А. С. Широков

Технический редактор Т. Г. Родина

Корректор Л. А. Буданцева

**ИБ № 1845**

Подписано в печать 12.02.90 Формат 60 x 88/16 Бумага офсетная № 2 Гарнитура "Пресс-роман"  
Печать офсетная Усл. печ. л. 13,72 Усл.кр.-отт. 14,33 Уч.изд.л. 13,75 Тираж 20 000 экз. Изд. №  
22732 Зак. № 6998 Цена 90 к.

Издательство "Радио и связь". 101000, Москва, Почтамт, а/я 693

Ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО "Первая Образцовая типография" Государственного комитета СССР по печати. 113054, Москва, Валуевская, 28.

**ISBN 5-256-00704-1 (рус.)**

**ISBN 0-201-10331-1 (англ.)**

© 1986 by Bell Telephone Laboratories,  
Incorporated

© Перевод на русский язык, примечания. Логунов М.Г., 1990

## ОГЛАВЛЕНИЕ

Предисловие . . . . .	8
ЧАСТЬ I. Введение . . . . .	11
Глава 1. Как расколоть орешек . . . . .	11
1.1. Дружеский разговор . . . . .	12
1.2. Точная формулировка задачи . . . . .	13
1.3. Разработка программы . . . . .	13
1.4. Набросок решения . . . . .	15
1.5. Основные принципы . . . . .	16
1.6. Задачи . . . . .	17
1.7. Литература для дополнительного чтения . . . . .	20
Глава 2. Ага! Алгоритмы . . . . .	21
2.1. Три задачи . . . . .	21
2.2. Вездесущий двоичный поиск . . . . .	22
2.3. Сила примитивов . . . . .	24
2.4. Соберем все вместе (сортировка) . . . . .	26
2.5. Основные принципы . . . . .	27
2.6. Задачи . . . . .	29
2.7. Литература для вспомогательного чтения . . . . .	31
2.8. Реализация программы для анаграмм (дополнение) . . . . .	31
Глава 3. Программы, работающие со структурами данных . . . . .	34
3.1. Программа обработки результатов обследования . . . . .	34
3.2. Формирование писем . . . . .	37
3.3. Примеры . . . . .	41
3.4. Большая по объему программа . . . . .	43
3.5. Основные принципы . . . . .	45
3.6. Задачи . . . . .	46
3.7. Литература для дополнительного чтения . . . . .	49
Глава 4. Написание программ, не содержащих ошибок . . . . .	49
4.1. Двоичный поиск "бросает вызов" . . . . .	49
4.2. Написание программы . . . . .	51
4.3. Разбор программы . . . . .	54
4.4. Реализация программы . . . . .	57
4.5. Основные принципы . . . . .	59
4.6. Зачем нужна верификация программ . . . . .	60

4.7. Задачи .....	62
4.8. Литература для дополнительного чтения .....	65
4.9. Верификация программ при их "промышленном производстве" (дополнение) .....	65
<b>ЧАСТЬ II. Эффективность .....</b>	<b>68</b>
<b>Глава 5. Производительность в перспективе. ....</b>	<b>69</b>
5.1. Разбор примера .....	69
5.2. Этапы разработки. ....	72
5.3. Основные принципы .....	74
5.4. Задачи .....	75
5.5. Литература для дополнительного чтения .....	76
<b>Глава 6. Предварительные оценки .....</b>	<b>77</b>
6.1. Основные навыки .....	78
6.2. Быстрые вычисления при разработке компьютерных систем .....	80
6.3. "Запас прочности" .....	81
6.4. Разбор примера .....	83
6.5. Основные принципы .....	85
6.6. Задачи .....	85
6.7. Литература для дополнительного чтения .....	86
6.8. Быстрые вычисления в повседневной жизни (дополнение) .....	87
<b>Глава 7. Методы разработки алгоритмов .....</b>	<b>88</b>
7.1. Задача и простой алгоритм ее решения .....	89
7.2. Два квадратичных алгоритма .....	90
7.3. Алгоритм "разделяй и властвуй" .....	92
7.4. Сканирующий алгоритм .....	94
7.5. Проверка на практике .....	95
7.6. Основные принципы .....	97
7.7. Задачи. ....	99
7.8. Литература для дополнительного чтения .....	100
7.9. Влияние алгоритмов на качество программ (дополнение) .....	101
<b>Глава 8. Оптимизация программы. ....</b>	<b>103</b>
8.1. Типичная история .....	103
8.2. Первая помощь .....	104
8.3. "Основное лечение" — двоичный поиск. ....	108
8.4. Основные принципы .....	113
8.5. Задачи. ....	115
8.6. Литература для дополнительного чтения .....	117
8.7. Оптимизация программ на Коболе, используемых федеральным правительством (дополнение) .....	117
<b>Глава 9. Сокращение объема памяти. ....</b>	<b>118</b>
9.1. Простота — это ключ к успеху .....	119
9.2. Память для хранения данных .....	120
9.3. Сокращение количества команд .....	125
9.4. Основные принципы .....	127
9.5. Задачи. ....	129
9.6. Литература для дополнительного чтения .....	130
9.7. Два примера большого сокращения памяти (дополнение) .....	130

ЧАСТЬ III. Программный продукт . . . . .	133
Глава 10. <b>Сортировка</b> . . . . .	133
10.1. Сортировка методом вставок. Алгоритм порядка $O(N^2)$ . . . . .	134
10.2. Быстрая сортировка — алгоритм порядка $O(N \log N)$ . . . . .	136
10.3. Основные принципы. . . . .	142
10.4. Задачи. . . . .	143
10.5. Литература для дополнительного чтения . . . . .	146
Глава 11. <b>Поиск</b> . . . . .	146
11.1. Задача. . . . .	146
11.2. Одно из решений . . . . .	148
11.3. Пространство решений . . . . .	149
11.4. Основные принципы. . . . .	154
11.5. Задачи. . . . .	155
11.6. Литература для дополнительного чтения . . . . .	157
Глава 12. <b>Пирамиды</b> . . . . .	157
12.1. Структуры данных . . . . .	158
12.2. Две важные подпрограммы . . . . .	160
12.3. Очереди с приоритетами . . . . .	164
12.4. Алгоритм сортировки . . . . .	166
12.5. Основные принципы. . . . .	169
12.6. Задачи. . . . .	170
12.7. Литература для дополнительного чтения . . . . .	173
Глава 13. <b>Программа проверки правописания</b> . . . . .	173
13.1. Простая программа . . . . .	173
13.2. Пространство возможных решений . . . . .	175
13.3. Программа с тонкостями. . . . .	179
13.4. Основные принципы. . . . .	183
13.5. Задачи. . . . .	184
13.6. Литература для дополнительного чтения . . . . .	185
13.7. Почему проверка правописания — трудная задача (дополнение) . . . . .	186
ЭПИЛОГ . . . . .	187
Приложение. <b>Каталог алгоритмов</b> . . . . .	189
Сортировка . . . . .	190
Поиск . . . . .	192
Другие алгоритмы для работы с множествами . . . . .	193
Алгоритмы для векторов и матриц . . . . .	193
Случайные объекты. . . . .	194
Другие алгоритмы . . . . .	194
Подсказки для некоторых задач. . . . .	194
Решения некоторых задач . . . . .	198



## ПРЕДИСЛОВИЕ

Программирование компьютеров многосторонне, что отразил Ф. П. Брукс в своей книге *Mythical Man Month*<sup>1</sup>. В его очерках подчеркивается также решающая роль управления разработкой больших программных проектов. Детально качественному программированию можно научиться по книге Кернигана, Плоджера *Elements of Programming Style*<sup>2</sup>. Рассматриваемые в этих книгах темы дают ключ к разработке хороших программных средств и критерии оценки профессионального программиста. Но, к сожалению, если программный продукт готов в срок и работает без сюрпризов, то искусное использование здравых инженерных принципов волнует далеко не всех.

Эта книга рассказывает о наиболее притягательной стороне профессии программиста: жемчужинах программирования, которые возникают в мире творчества и интуиции на основе хорошей инженерной подготовки. Подобно истинным жемчужинам, вырастающим в раковинах из песчинок, жемчужины программирования выросли из реальных задач в умах программистов. Эти программы учат важным методам программирования и фундаментальным принципам разработки программ.

Предлагаемые очерки выбраны из раздела "Жемчужины программирования" (*Programming Pearls*), который я веду в журнале *Communications of the Association for Computing Machinery (CACM)*; историю этой публикации можно найти во введениях к ч. I, II и III. После появления в журнале CACM материал, изложенный в этой книге, был существенно переработан: добавлены новые разделы, в старые внесена масса улучшений, главы стали более взаимосвязанными (более подробно см. эпилог). Единственное, что нужно для понимания этого материала, — опыт программирования на языках высокого уровня. Читатели, не знакомые

---

<sup>1</sup> Ф. П. Брукс, мл. Как проектируются и создаются программные комплексы: Пер. с англ. — М.: Наука, 1979. — *Прим. перев.*

<sup>2</sup> Б. Керниган, П. Плоджер. Элементы стиля программирования: Пер. с англ. — М.: Радио и связь, 1984. — *Прим. перев.*

с такими сложными методами, как рекурсия, могут, ничего не потеряв, перейти к следующему разделу.

Хотя каждую главу можно читать отдельно, они логически связаны между собой. Главы 1–4 составляют ч. I этой книги и содержат обзор основ программирования: постановку задачи, алгоритмы, структуры данных, верификацию программ. В ч. II входят гл. 5–9. В них рассмотрена только одна, иногда очень важная тема – эффективность, повышение которой всегда является замечательным трамплином для прыжка к решению интересных задач программирования. В главах ч. III изложенные ранее методы используются в реальных задачах.

Хочу дать один совет: при чтении этой книги не спешите. Читайте главы тщательно, по одной. Пытайтесь решить задачи – некоторые из них кажутся простыми до тех пор, пока не поломаете над ними голову в течение одного или двух часов. Работайте над задачей тщательно и до самого конца: большая часть того, что вы вынесете из этой книги, будет реализована при записи ваших решений. Прежде чем заглянуть в подсказки и решения, помещенные в конце книги, обсудите свои идеи с друзьями и коллегами. Литература для дополнительного чтения приведена к концу каждой главы, но это не строгий список литературы, который принято давать в научных трудах, а лишь перечень некоторых хороших книг, составляющих важную часть моей личной библиотеки.

Эта книга написана для программистов. Я надеюсь, что задачи, подсказки, решения и рекомендованная литература сделают ее полезной как для профессионалов, так и для любителей. Я использовал ее наброски на занятиях со студентами по курсам "Разработка прикладных алгоритмов" и "Проектирование программных средств"; у издателя имеется дополнительный учебный материал. Каталог алгоритмов в приложении – удобный справочник для "практикующих" программистов, объясняющий, как эту книгу можно включить в учебные курсы по алгоритмам и структурам данных.

При написании и выпуске книги мне оказали поддержку многие, за что я им очень благодарен. Идея возникновения раздела в журнале Communications of the ACM первоначально возникла у Питера Деннинга и Стюарда Линна. Питер усердно трудился в ACM над тем, чтобы сделать этот раздел реальным, и привлек меня к этой теме. Сотрудники "штаб-квартиры" ACM, особенно Роз Стайер и Нэнси Эйдриенс, очень помогли тому, чтобы материал этих разделов был опубликован в первоначальном виде. Я особенно признателен ACM за поддержку при публикации материала этих разделов в их теперешнем виде, а также

многим читателям, которые своими комментариями к ним сделали возможным и необходимым их издание в развернутом виде.

Э. Ахо, П. Деннинг, М. Гари, Д. Джонсон, Б. Керниган, Д. Линдерман, Д. Макилрой и Д. Стонат, несмотря на сильную занятость, прочитали все главы с великим вниманием. За особенно ценные комментарии я благодарю также Г. Бэрда, М. Бентли, Б. Кливленда, Д. Гризау, Э. Гросса, Л. Джнелински, С. Джонсона, Б. Меллвила, Б. Мартина, А. Пензиаса, К. Ван Уайк, В. Висоцки и П. Зейв. А. Ахо, А. Хьюм, Б. Керниган, Р. Сети, Л. Скингер и Б. Строустреп оказали неоценимую помощь в подготовке книги, а курсанты Уэст-Пойнта провели "полевые испытания" предпоследнего варианта рукописи этой книги. Спасибо всем.

*Мюррей Хилл, Нью-Джерси.*

*Джон Бентли*

## Часть I

### ВВЕДЕНИЕ

Эта часть состоит из четырех глав, в которых рассматриваются основы программирования. Глава 1 посвящена истории возникновения некоторой задачи. Тщательная постановка этой задачи и простая методика программирования привели к элегантному решению. Этот случай иллюстрирует основную тему книги: упорные размышления над конкретной задачей могут привести к результатам, полезным на практике.

В гл. 2 исследуются три задачи, при этом упор делается на то, как глубокое понимание алгоритма позволяет получить простую программу. В гл. 3 рассматривается значение структур данных в разработке программы.

Глава 4 знакомит нас с проблемой верификации программ и с ролью, которую верификация может играть после того, как программа написана. Методы верификации широко используются в гл. 8, 10 и 12.

Составляющие ч. I главы были первыми из опубликованных в разделе "Жемчужины программирования" (Programming Pearls) журнала Communications of the ACM. Глава 1 появилась в августовском, гл. 2 – в сентябрьском, гл. 3 – в октябрьском, а гл. 4 – в декабрьском номере за 1983 г.

#### Глава 1. КАК РАСКОЛОТЬ ОРЕШЕК

Вопрос программиста был прост: "Как мне выполнить сортировку на диске?" Перед тем, как рассказать о своей первой ошибке, я хочу предоставить вам возможность поступить правильнее, чем я в свое время. Представьте себе, что вам задали простой вопрос: "Как выполнить сортировку на диске?" Что бы вы ответили?

Моя ошибка была в том, что я ответил на вопрос, вкратце объяснив, как надо выполнять сортировку на диске. Так как мое предложение изучить классическую книгу Кнута "Сортировка и поиск" не было встречено с энтузиазмом – собеседника больше волновало решение конкретной задачи, а не продолжение своего образования, – я рассказал ему о программе сортировки на диске, описанной в гл. 4 книги Кернигана и Плоджера "Программные средства". Эта программа состоит из 12 процедур и содержит около 200 строк в кодах языка Ратфор. На ее преобразование в несколько сотен строк на языке Фортран и тестирование потребовалось бы около недели.

Сомнения моего собеседника в том, что я разрешил его проблему, навели меня на правильный путь. В дальнейшем наша беседа протекала так (мои вопросы выделены курсивом):

*Зачем вам понадобилось писать программу сортировки?*

*Почему не воспользовались системными средствами сортировки?*

Мне необходимо выполнить сортировку в рамках большой системы, а в операционной системе нет возможности для перехода от программы пользователя к системным программам.

*Что конкретно вам надо сортировать? Сколько записей в файле, каков формат каждой записи?*

Файл содержит до 27 000 записей, каждая запись – это 16-разрядное целое число.

*Подождите минутку. Если файл настолько мал, зачем вообще связываться с диском? Почему бы просто не сортировать в ОЗУ?*

Хотя машина имеет ОЗУ емкостью 0.5. Мбайта, процедура сортировки – только часть большой программы. Я предполагаю, что в этот момент будет свободно всего около 1000 16-разрядных слов.

*Можете ли вы сообщить еще что-нибудь об этих записях?*

Каждая из них – это целое число в диапазоне от 1 до 27 000 и ни одно из них не встречается более одного раза.

Постановка задачи выяснилась из контекста. Эта система использовалась для избирательных округов (автоматической перекройки этих округов), а сортировать надо было индексы избирательных участков, образующих избирательный округ. Каждый участок в штате имеет свой уникальный номер в диапазоне от 1 до 27 000 (число участков в самом

большом штате), и нельзя включать один и тот же участок дважды в один округ. На выходе требовалось иметь список избирательных участков округа по порядку их номеров. Из контекста выяснились также требования к временным характеристикам: так как пользователь хочет вмешиваться в процесс расчета примерно один раз в час, чтобы запустить программу сортировки, и не может ничего делать, пока она не закончится, сортировка должна занимать не более нескольких минут, а лучше нескольких секунд.

## 1.2. ТОЧНАЯ ФОРМУЛИРОВКА ЗАДАЧИ

Вышеприведенные требования должны быть добавлены к вопросу программиста: "Как мне выполнить сортировку на диске?" Перед тем, как подойти к решению задачи, давайте запишем все известные теперь сведения в менее привязанных к конкретной задаче и более удобных формулировках:

**Вход:** Файл, содержащий до 27 000 целых чисел в диапазоне от 1 до 27 000. Появление любого числа дважды является фатальной ошибкой. Никакие другие данные с этими числами не связаны.

**Выход:** Список упорядоченных в порядке возрастания целых чисел, имевшихся на входе.

**Ограничения:** В ОЗУ имеется не более 1000 16-разрядных слов. Доступны буферы диска в ОЗУ и имеется обширная память на диске. Максимальное время работы может составлять несколько минут. Не требуется сокращать время работы менее чем до 10 с.

Задумайтесь на минутку, что бы вы еще посоветовали программисту?

## 1.3. РАЗРАБОТКА ПРОГРАММЫ

Очевидное решение – взять за основу универсальную программу Кернигана и Плоджера для сортировки на диске и сократить ее, воспользовавшись тем, что мы сортируем целые числа. Это позволяет ускорить ее работу и уменьшить на несколько десятков строк текст этой программы, составляющий 200 строк.

Второе решение еще в большей степени использует особенности задачи сортировки. Основной цикл этого решения состоит из 27 проходов по входному файлу. При первом проходе в память считываются все числа от 1 до 1000, далее эти числа (их не более 1000) сортируются и записываются в выходной файл. При втором проходе сортируются числа

от 1001 до 2000 и т. д. до 27-го прохода, сортирующего числа от 26 001 до 27 000. Для сортировки в ОЗУ достаточно эффективна программа быстрой сортировки QUICKSORT, разработанная Керниганом и Плотджером. Программа состоит примерно из 40 строк на языке Ратфор (мы рассмотрим несколько программ сортировки в гл. 10 и 12). Поэтому вся программа могла бы состоять из 80 операторов языка Фортран. Она обладает также следующим хорошим свойством: нам не требуется больше беспокоиться об использовании промежуточных файлов на диске. К несчастью, за это достоинство нам приходится расплачиваться чтением 27 раз всего входного файла.

Программа сортировки слиянием читает данные из входного файла один раз, сортирует их с помощью рабочих файлов, которые читаются и записываются многократно, и потом один раз записывает выходные данные:

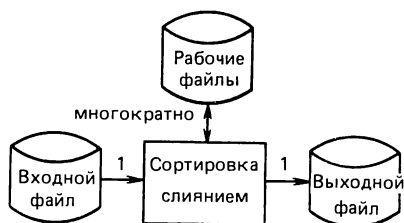


Рис. 1.1

В алгоритме с 27 проходами входной файл читается много раз, выходной файл записывается только один раз, промежуточные файлы не используются.

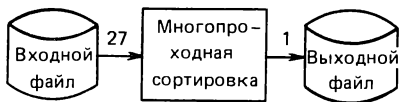


Рис. 1.2

С моей точки зрения, предпочтительнее следующая схема, которая сочетает достоинства двух предыдущих: входной файл читается только один раз и не используются промежуточные файлы:

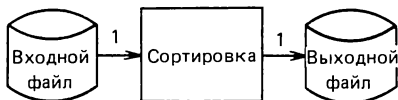


Рис. 1.3

Так можно сделать, отобразив все целые числа из входного файла примерно в 1000 слов, имеющихся в нашем распоряжении в ОЗУ. Таким образом, задача сводится к тому, сможем ли мы представить 27 000 различных целых чисел с помощью примерно 16 000 доступных нам битов? Задумайтесь о таком представлении.

#### 1.4. НАБРОСОК РЕШЕНИЯ

При таком подходе кажется разумным использовать *побитовое* представление набора данных. Мы представим файл строкой из 27 000 битов, в которой бит  $I$  равен 1 в том и только в том случае, когда в файле есть целое число  $I$ . (Мой собеседник нашел 11 000 свободных битов; в задаче 1 исследуется случай, когда 16 000 битов – верхний предел.) Такое представление использует три особенности этой задачи, обычно не встречающиеся в задачах сортировки: небольшой диапазон допустимых входных чисел, отсутствие дубликатов, а также то, что с каждой записью не связано никаких других данных, кроме единственного целого числа.

Задавшись побитовой структурой данных для представления множества целых чисел в файле, программу можно написать в виде трех очевидных частей. В первой части подготавливается массив битов – все биты устанавливаются в 0. Вторая часть заполняет этот массив посредством чтения каждого числа из файла и установки соответствующего бита в 1. В третьей части порождается отсортированный выходной файл. Для этого проверяется каждый бит и выводится соответствующее число, если бит равен 1. Пусть  $N$  – общее число битов (в нашем случае  $N = 27\,000$ ), тогда программа на псевдоязыке может быть записана так:

```
/* Часть I: начальное обнуление массива */  
  for I := 1 to N do  
    Bit[I] := 0  
/* Часть II: ввести имеющиеся элементы в массив */  
  для каждого целого числа I из входного файла  
    Bit[I] := 1  
/* Часть III: выдать упорядоченный файл */  
  for I := 1 to N do  
    if Bit[I] = 1 then  
      записать I в выходной файл
```



Этого наброска оказалось достаточно, чтобы программист решил свою задачу. Некоторые из особенностей реализации, с которыми он столкнулся, описаны в задачах 1, 2 и 6 в конце данной главы.

### 1.5. ОСНОВНЫЕ ПРИНЦИПЫ

Мой собеседник рассказал мне о своей задаче по телефону. Нам потребовалось около 15 мин, чтобы разобраться в особенностях задачи и найти решение с побитовым представлением. Написание программы из нескольких десятков строк на Фортране заняло у него пару часов. Это было неплохо по сравнению с сотнями строк и недель программирования, чего мы боялись в начале телефонного разговора. Программа работала молниеносно: в то время как программа сортировки слиянием на диске могла бы работать несколько минут, эта программа затрачивала ненамного больше времени, чем требуется для чтения входных данных и записи выходных, — менее 10 с.

В этих фактах содержится первый урок, извлекаемый из рассмотрения данного случая: тщательный анализ небольшой задачи может иногда принести потрясающую практическую пользу. В данном примере тщательное изучение в течение нескольких минут привело к сокращению на порядок числа операторов в программе, времени программирования и времени прогона. Генерал Ч. Йегер (первый человек, полетевший со сверхзвуковой скоростью) похвалил двигательную установку самолета следующими словами: "Проста, мало деталей, легко обслуживать, очень прочная". Эти эпитеты относятся и к нашей программе. Однако может оказаться, что специфические структуры данной программы будет тяжело модифицировать при изменении объемов некоторых данных. Являясь рекламой искусного программирования, этот пример иллюстрирует следующие общие принципы.

*Правильная постановка задачи.* Наш спор почти целиком был посвящен формулировке задачи. Я рад, что программист не успокоился на первой программе, которую я ему описал. Задачи 9 и 10 имеют элегантные решения, если вы правильно их сформулируете. Хорошенько подумайте над ними, прежде чем смотреть подсказки и решения.

*Побитовая структура данных.* Эта структура данных представляет собой компактное отображение конечного набора данных для случая, когда каждый элемент встречается не более одного раза и с ним не связано никаких других данных. Даже если эти условия не удовлетворяются (имеются повторяющиеся элементы или дополнительные данные), ключ, полученный с помощью этого отображения, может быть использован в качестве ссылки на адрес в таблице, содержащей более сложные входные данные.

*Многопроходные алгоритмы.* Эти алгоритмы совершают несколько проходов по входным данным, продвигаясь каждый раз немного вперед. Ранее мы рассмотрели 27-проходный алгоритм. В задаче 1 требуется разработка 2-проходного алгоритма.

*Компромисс "память-время" и случай невыполнения.* Программистский фольклор и теория о компромиссе "память-время" сообщают следующее: дольше работающей программе может потребоваться меньший объем памяти. Например, в 2-проходном алгоритме, приведенном в решении задачи 1, удваивается время работы, чтобы вдвое сократить занимаемую память. Однако, как показывает мой опыт, гораздо чаще при уменьшении объема памяти, требующейся программе, уменьшается и время ее работы<sup>1</sup>. Эффективная по объему занимаемой памяти побитовая структура существенно сокращает время сортировки. В данной задаче уменьшение объема памяти привело к сокращению времени по двум причинам: меньший объем данных, которые надо обработать, приводит к сокращению времени обработки, а хранение данных в ОЗУ, а не на диске, устраняет накладные расходы на доступ к диску. Конечно, улучшение обоих параметров стало возможным только потому, что исходный вариант программы был далеко не оптимальным.

*Простота замысла.* Антуан де Сент-Экзюпери, французский писатель и авиатор, говорил: "Конструктор знает, что он достиг совершенства не тогда, когда нечего больше добавить, а тогда, когда нечего больше убрать". Многие программисты должны оценить свою работу по этому критерию. Простые программы обычно более надежны, "здоровомыслящи", эффективны, чем их сложные родственники, их намного проще создавать и сопровождать.

*Этапы разработки программы.* Рассмотренный пример иллюстрирует процесс разработки, подробно описанный в разд. 11.4.

## 1.6. ЗАДАЧИ

Подсказки и решения для некоторых задач можно найти в конце книги.

1. Мой собеседник – программист – сказал, что в его распоряжении есть около 1000 свободных слов в памяти, но в рассмотренном фрагменте

---

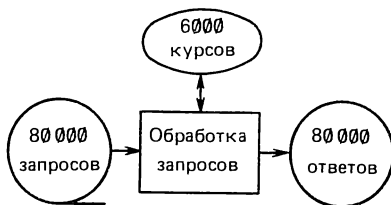
<sup>1</sup> Компромиссы часто встречаются во всех инженерных дисциплинах; конструкторы автомобилей, например, обычно увеличивают удельный расход топлива, добавляя тяжелые детали для улучшения эксплуатационных характеристик. Однако предпочтительней взаимное улучшение характеристик. Анализируя конструкцию экономичных автомобилей, которые я водил, могу заметить, что "уменьшение массы за счет основных устройств автомобиля приводит к дальнейшему сокращению массы различных деталей шасси – и даже к тому, что некоторые из них, такие, как усилитель руля, становятся не нужны".

программы используется для кодирования  $27\,000/16 = 1688$  слов. Без особых усилий он смог высвободить дополнительную память. А что можно рекомендовать, если 1000 слов – жесткий предел? Сколько времени будет работать ваш алгоритм?

2. Одной из преград между наброском алгоритма и завершенной программой на Фортране является реализация побитового представления данных. Хотя это тривиально для языков, в которых поддерживаются строки битов как один из основных типов данных, программисты, использующие Фортран, вынуждены реализовывать эти структуры, используя другие операции. Предположив, что диалект Фортрана обеспечивает битовые логические операции над словами (такие как сдвиг, И, ИЛИ), как бы посоветовали реализовать операции над строками битов? Как бы вы реализовали побитовое представление, если бы этих операций не было? Как бы вы реализовали этот алгоритм на Коболе? на Паскале?
3. Важной целью разработки была минимизация времени работы, и итоговая программа получилась достаточно эффективной. Реализуйте программу на своей вычислительной системе и измерьте время ее работы. Сравните его с временем сортировки этого же файла системной программой.
4. Если вы возьметесь за задачу 3 серьезно, то столкнетесь с проблемой генерации  $K$  целых неповторяющихся чисел от 1 до 27 000. Самый простой способ – использовать первые  $K$  положительных целых чисел. Для этого вырожденного набора данных время работы алгоритма с побитовым представлением данных изменяется несущественно, но системная сортировка для него может быть выполнена гораздо быстрее, чем в случае типичных данных. Как бы вы сгенерировали файл из  $K$  целых неповторяющихся чисел от 1 до  $N$ , расположенных в случайном порядке? Постарайтесь, чтобы программа была короткой и эффективной.
5. Что бы вы порекомендовали программисту, если бы каждое число могло появиться не по одному разу, а не более 10 раз. Как бы ваше решение изменялось в зависимости от доступного объема памяти?
6. [Р. Вейл] Набросок программы имеет ряд недостатков. Во-первых, предполагается, что ни одно число не появляется на входе дважды. Что случится, если одно из них появится более одного раза? Как можно модифицировать программу, чтобы в этом случае была вызвана подпрограмма обработки ошибок? Что произойдет, если число на входе меньше 1 или больше  $N$ ? Что программа сделала бы в этих случаях? Опишите небольшие наборы данных для проверки программы, включая проверку правильности ее работы в указанных и других случаях, которые могут привести к неправильной работе программы.

7. Для системы управления колледжем программисту требуются структуры данных для учета числа свободных мест на различных курсах. Каждый из 6000 учебных курсов имеет уникальный четырехзначный идентификационный номер (от 0000 до 9999) и трехзначный счетчик числа мест (от 000 до 999). После создания структуры данных на основе файла, содержащего номера курсов и числа мест, программа должна была обработать ленту, содержащую примерно 80 000 заявок на курсы.

Рис. 1.4



Каждый запрос с правильным номером курса либо отвергается (если соответствующий счетчик числа мест равен 0), либо удовлетворяется (в этом случае счетчик числа мест уменьшается на 1). Запросы с неверными номерами курсов помечаются и игнорируются. После выделения места под объектные коды, буфера и т. п. в системе остается для пользователя около 30 Кбайт ОЗУ. В первом варианте своей программы на Коболе программист представил описание каждого курса в виде записи на диске из 7 байтов (4 байта отведено для номера курса и 3 – для счетчика числа мест). Операции обращения к диску сделали бы эту структуру слишком накладной. Как вы считаете, лучший ли это способ организации информации о курсах?

8. Одна из проблем, связанная с увеличением объема программы с целью сокращения времени ее работы, состоит в том, что сама инициализация памяти может занять много времени. Покажите, как обойти эту проблему, разработав методику обнуления всего одномерного массива при первом обращении к нему. Вы можете использовать дополнительную память, пропорциональную размеру массива. Так как этот метод уменьшает время инициализации за счет использования дополнительной памяти, его следует применять только тогда, когда свободное пространство дешево, время дорого, а массив разреженный. (Эта задача из упражнения 2.12, помещенного в книге Ахо, Хопкрофта и Ульмана "Разработка и анализ алгоритмов для ЭВМ" (Aho, Hopcroft, Ullman. Design and Analysis of Computer Algorithms), выпущенной издательством Addison-Wesley в 1974 г.

9. Отделы заказов в некоторых больших магазинах позволяют клиентам заказывать товары по телефону, указывая номера в каталоге. В базе данных магазина в качестве первичного ключа для поиска используется номер телефона клиента (клиенты знают свои номера телефонов, и ключи почти уникальны). Как бы вы организовали базу данных, чтобы обеспечить эффективный ввод и поиск заказов?
10. В 1981 г. инженеры фирмы Lockheed должны были передавать ежедневно около дюжины чертежей из системы автоматизированного проектирования (САПР) со своего завода в Саннивейле, шт. Калифорния, на испытательную базу в Санта Круз. Хотя оборудование было всего в 25 милях, связь с помощью курьера на автомобиле занимала около часа (из-за заторов и горных дорог) и обходилась 100 дол. ежедневно. Предложите альтернативные способы передачи данных и оцените их стоимость.

### 1.7. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Создавая свои "Программные средства", Керниган и Плуджер (Kernighan, Plauger. *Softwares Tools*) решили много мелких, но сложных и важных проблем программирования. Эта книга была первоначально опубликована в 1976 г. издательством Addison-Wesley. Более позднее издание со многими важными изменениями появилось в 1981 г. под названием "Программные средства на Паскале" (*Software Tools in Pascal*). Их подход к разработке программного обеспечения как к созданию инструментов может изменить ваши взгляды на программирование. Материал этой книги показывает, к чему следует стремиться при программировании – к программе простой структуры, которую легко использовать и сопровождать. Однако иногда авторы этой книги приводят тонкие решения трудных задач. Ссылки в предметном указателе на "алгоритмы" и "структуры данных" указывают на многие из таких жемчужин. К несчастью, изящные идеи часто описаны так просто, что читатель может подумать, что они действительно очевидны. Когда вы будете читать эту замечательную книгу, обязательно потратьте время, чтобы оценить эти жемчужины – они основываются на мощных методах.

В примере, описанном в данной главе, основная задача программиста была не столько техническая, сколько психологическая: он не мог продвинуться вперед, так как пытался решить неправильно сформулированную задачу. Мы в конце концов решили ее, прорвав его концептуальную блокаду и сведя эту задачу к более простой. Такие препятствия рассматриваются в книге "Концептуальный прорыв" Дж. Л. Адамса (James L. Adams. *Conceptual Blockbusting*) (второе издание выпущено издательством Norton в 1979 г.). Эта книга является мощным стимулом к развитию творческого мышления. Хотя она написана не в расчете на

программистов, многие из ее уроков очень подходят для задач программирования. Адамс определил концептуальную блокаду как "умозрительные стены, которые препятствуют человеку, решающему задачу, правильно понять задачу или представить себе решение". Над задачами 9 и 10 вам предстоит поломать голову.

## Глава 2. Ага! АЛГОРИТМЫ

Активно работающему программисту изучение алгоритмов дает очень много. Курс по этому предмету вооружает студентов алгоритмами решения важных задач и методами разработки алгоритмов для наступления на новые задачи. В следующих главах мы увидим, как более совершенные алгоритмические средства оказывают существенное влияние на системы программного обеспечения, сокращая время разработки и увеличивая скорость выполнения программ.

Кроме того, алгоритмы играют существенную роль в решении общих задач программирования. В своей книге "Ага! Озарение" (Martin Gardner. Aha! Insight), откуда я бесстыдно украл заглавие, М. Гарднер упоминает о той роли, которую я имею в виду: "Задача, которая кажется сложной, может иметь простое, неожиданное решение". В отличие от более совершенных методов, *ага! Озарение* происходит не только после всестороннего изучения; оно доступно любому программисту, который готов серьезно подумать до, во время и после написания программы.

### 2.1. ТРИ ЗАДАЧИ

Но довольно общих рассуждений. Материал этой главы основывается на трех маленьких задачах; попытайтесь их решить перед тем, как читать дальше.

А. Имеется магнитная лента, содержащая не более миллиона 20-битовых целых чисел в случайном порядке. Требуется найти 20-битовое число, которого нет на ленте (должен иметься по меньшей мере один пропуск – кстати, почему?). Как бы вы решали эту задачу, имея а) ОЗУ достаточного объема; б) несколько лентопротяжек, но только пару дюжин слов в ОЗУ?

Б. Осуществите циклический сдвиг одномерного массива из  $N$  элементов влево на  $I$  позиций. Например, при  $N=8$  и  $I=3$  вектор ABCDEFGH после сдвига превратится в вектор DEFGHABC. Чтобы выполнить эту работу за  $N$  шагов, простая программа использует промежуточный буфер из  $N$  элементов. Можете ли вы выполнить циклический сдвиг за время, пропорциональное  $N$ , используя только несколько дополнительных слов памяти?

В. Дан словарь английских слов. Найдите все наборы анаграмм. Например, pots, stop и tops являются анаграммами по отношению друг к другу, так как каждое слово может быть образовано из другого перестановкой букв.

## 2.2. ВЕЗДЕСУЩИЙ ДВОИЧНЫЙ ПОИСК

Я задумываю целое число от 1 до 100, а вы его угадываете. 50? Слишком мало. 75? Слишком много. И так игра продолжается до тех пор, пока вы не угадаете мое число. Если мое число было задумано в диапазоне от 1 до  $N$ , тогда вы угадаете его за  $\log_2 N$  попыток. Если  $N = 1000$ , будет сделано 10 попыток, а если  $N = 1\,000\,000$ , вам потребуется не более 20 попыток.

Этот пример иллюстрирует метод, который позволяет решить множество задач программирования, он называется *двоичный поиск*. В начальный момент мы знаем, что объект находится в заданной области, а операция выбора и проверки значения "сообщает" нам, где находится объект – в заданной позиции, ниже или выше ее. При двоичном поиске местоположение объекта обнаруживается с помощью повторяющегося выбора элемента из середины текущей области. Если выбранный элемент не тот, который мы ищем, делим текущую область пополам и продолжаем. Мы остановимся, если найдем то, что искали, или область станет пуста.

Наиболее простым примером применения двоичного поиска в программировании является поиск элемента в упорядоченном массиве. При поиске числа 50 алгоритм делает следующие попытки:

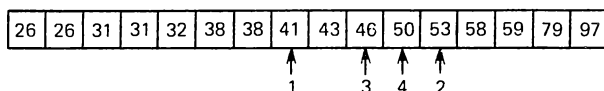


Рис. 2.1

Бытует мнение, что программу двоичного поиска тяжело написать без ошибок. Подробно мы изучим ее в гл. 4.

При последовательном поиске выполняется в среднем около  $N/2$  сравнений, если таблица содержит  $N$  элементов, в то время как при двоичном поиске никогда не выполняется более  $\log_2 N$  сравнений. Это может привести к существенному различию в эффективности системы. Вот типичная история, взятая из июльского номера журнала "Communications of the ACM" за 1984 г., где приведен анализ "Системы резервирования TWA".

У нас была программа, осуществлявшая примерно 100 раз/с линейный поиск для очень большого объема данных. В связи с ростом сети среднее время работы процессора, затрачиваемое на обработку

сообщения, возросло на 0.3 мс, что в данном случае является очень большим скачком. Мы установили, что причина в том, что поиск осуществляется линейно, заметили прикладную программу, чтобы использовать двоичный поиск, и проблема была решена.

Но рассказ о двоичном поиске не заканчивается быстрым поиском в упорядоченном массиве. Р. Уил из компании Michael Backer Jr., Inc. применил этот метод для очистки колоды, содержащей примерно 10 000 перфокарт, от имеющейся в ней единственной плохой карты. К несчастью, о наличии плохой карты неизвестно заранее. О ней можно узнать, лишь прогнав подмножество карт через программу и получив бессмысленный ответ, а это занимает несколько минут. Предшественники Уила пытались обнаружить ошибку, пропуская за один раз по несколько карт через программу и продвигалась к решению черепашьим шагом. Догадайтесь, каким образом Уил нашел подозреваемую карту только за десять прогонов программы?

После этой разминки мы можем энергично взяться за задачу А. Дана магнитная лента, содержащая в случайном порядке не более 1 млн. 20-битовых целых чисел, и мы должны найти одно 20-битовое число, отсутствующее на ленте. (Здесь должен быть хотя бы один пропуск, так как таких чисел всего  $2^{20}$  или 1 048 576.) Имея достаточный объем ОЗУ, мы могли бы использовать побитовое представление этих чисел (гл. 1), выделив 131 072 восьмибитовых слов. Однако в задаче также спрашивается о том, как найти пропущенное число, имея только несколько десятков слов в ОЗУ и несколько дополнительных лентопротяжек. Для определения того, в какой половине содержится пропущенный элемент, при использовании метода двоичного поиска, нужно найти область, представление элементов в области и метод зондирования. Как это сделать?

Областью будем считать последовательность чисел, в которой должен содержаться по крайней мере один пропущенный элемент, и пусть все эти числа записаны на магнитной ленте. Идея состоит в том, чтобы прозондировать область, подсчитав число элементов выше и ниже ее средней точки – либо в верхней, либо в нижней части области содержится не более половины элементов всей области. Так как в области нет одного элемента, то он отсутствует именно в меньшей половине. В этой задаче используется большая часть компонентов алгоритма двоичного поиска; попытайтесь сами скомпоновать их, прежде чем посмотрите решение и увидите, как это сделал Э. Рейнгоулд.

Эти примеры использования двоичного поиска дают только поверхностное представление о его применении в программировании. В программе нахождения корня используется двоичный поиск для решения



уравнения одной переменной последовательным делением интервала пополам. В решении задачи 9 из гл. 10 выбирается случайный элемент, осуществляется разбиение, а потом выполняется рекурсия для всех элементов по одну сторону от этого элемента, что называется рандомизированным двоичным поиском. Двоичный поиск применяется также для древовидных структур данных, в алгоритмах сортировки карт (в которых используется соответствующая десятичная сортировка) и отладке программ (если программа "тихо угасла", куда вставить операторы печати, чтобы локализовать неверный оператор?). В каждом из этих примеров представление о программе как о совокупности небольших надстроек на фундаменте основного алгоритма двоичного поиска может натолкнуть программиста на это всемогущее *aga!*

### 2.3. СИЛА ПРИМИТИВОВ

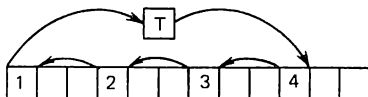
Двоичный поиск – это решение, подходящее для многих задач; теперь рассмотрим задачу, имеющую несколько решений. Задача Б состоит в циклическом сдвиге вектора  $X$  из  $N$  элементов влево на  $I$  позиций за время, пропорциональное  $N$ , и с использованием только нескольких дополнительных слов памяти. В различных приложениях эта задача имеет разный вид: в таких языках программирования, как APL, циклический сдвиг обеспечивается как примитивная операция над векторами. В работе "Программные средства на Паскале" (с. 194, 195) в своей реализации редактора текста Керниган и Плоджер используют программу циклического сдвига. Ограничения на время и память важны в обоих указанных приложениях.

Можно попытаться решить эту задачу, копируя первые  $I$  элементов из вектора  $X$  во временный вектор, сдвигая оставшиеся  $N - I$  элементов влево на  $I$  позиций и копируя потом первые  $I$  элементов из временного вектора обратно на последние позиции в вектор  $X$ . Однако  $I$  дополнительных слов ОЗУ, используемые этой схемой, делают ее слишком дорогой с точки зрения памяти. При другом подходе мы могли бы написать подпрограмму для циклического сдвига вектора  $X$  влево на одну позицию (за время, пропорциональное  $N$ ) и вызвать ее  $N$  раз, но это слишком расточительно по времени.

Для решения данной задачи при ограничениях на ресурсы, по всей вероятности, требуется более сложная программа. Один из успешных подходов осуществляется с помощью изящного трюка: поместить элемент  $X[1]$  во временную ячейку  $T$ , а потом перемещать элементы  $X[I + 1]$  в  $X[1]$ ,  $X[2I + 1]$  в  $X[I + 1]$  и т. д. (беря все индексы вектора  $X$  по модулю  $N$ ) до тех пор, пока не вернемся к элементу  $X[1]$ , вместо которого в этот момент мы подставим элемент из  $T$  и остановим процесс.

Если  $I = 3$  и  $N = 12$ , то на этой фазе элементы перемещаются в следующем порядке:

Рис. 2.2



Если при этом были перемещены не все элементы, то начнем с элемента  $X[2]$  и будем продолжать указанные действия, пока не переместим все элементы. Будьте внимательны! В задаче 3 вам брошен вызов: превратить эту идею в программу.

Следующий алгоритм является следствием другого подхода к задаче: циклически сдвинуть вектор  $X$  — это значит в действительности поменять местами две части вектора  $AB$  так, чтобы получился вектор  $BA$ , где  $A$  соответствует первым  $I$  элементам вектора  $X$ . Предположим, что  $A$  короче, чем  $B$ . Разделим  $B$  на две части  $B_L$  и  $B_R$  так, чтобы часть  $B_R$  имела такую же длину, что и  $A$ . Поменяем местами  $A$  и  $B_R$ , чтобы трансформировать  $AB_L B_R$  в  $B_R B_L A$ . Последовательность элементов  $A$  на своем конечном месте, поэтому мы можем сосредоточиться на перестановке двух частей  $B$ . Так как эта новая задача аналогична исходной, решать ее можно рекурсивно. Такой алгоритм может привести к изящной программе (в решении задачи 3 описано итеративное решение из восьми строк, предложенное Гризом и Миллзом), но требует искусного программирования и некоторых раздумий, чтобы убедиться в его эффективности.

Эта задача выглядит трудной до тех пор, пока вас не посетит ага! Озарение: давайте посмотрим на нее как на трансформацию массива  $AB$  в массив  $BA$ , но предположим, что у нас есть подпрограмма, которая реверсирует (меняет порядок элементов на обратный) заданную часть массива. Начав с массива  $AB$ , мы реверсируем его часть  $A$ , чтобы получить  $A^R B$ ; реверсируем часть  $B$ , чтобы получить  $A^R B^R$ ; а потом реверсируем их совокупность и получаем  $(A^R B^R)^R$ , что в точности соответствует массиву  $BA$ . Это приводит к следующей программе для циклического сдвига (в комментариях показаны результаты при циклическом сдвиге последовательности  $ABCDEFGH$  на 3 элемента):

```
Reverse(1,I)      /* CBADEFGH */
Reverse(I+1,N)    /* CBAHG FED */
Reverse(1,N)      /* DEFGHABC */
```

Д. Макилроем предложена иллюстрация циклического сдвига 10-элементного массива на пять позиций с помощью переворотов ладоней. Начните, расположив ладони перед собой, левую над правой.

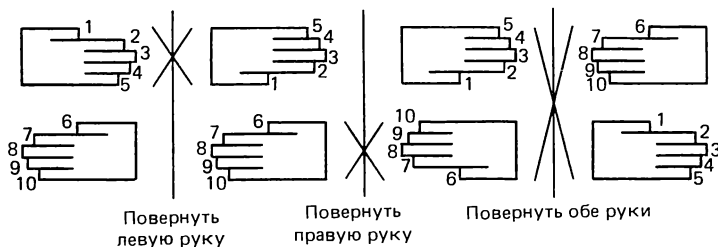


Рис. 2.3

Программа реверсирования эффективна по времени, по памяти и настолько коротка и проста, что в ней довольно тяжело допустить ошибки. Это в точности та программа, которую использовали в своей книге Керниган и Плоджер для редактора текста, которая стала работать правильно при первом же прогоне, в то время как их предыдущая программа для решения подобной задачи, в основе которой лежали списки со ссылками, содержала несколько ошибок. Эта программа использовалась в нескольких редакторах текста, в том числе в редакторе ED для операционной системы UNIX, с помощью которого я ввел с клавиатуры эту главу. К. Томпсон написал этот редактор и программу реверсирования в 1971 г. и утверждает, что она стала легендой.

#### 2.4. СОБЕРЕМ ВСЕ ВМЕСТЕ (СОРТИРОВКА)

Обратимся теперь к задаче В. Дан словарь английских слов (по одному слову, набранному строчными буквами, на каждой строке входных данных), а мы должны найти все группы анаграмм. Для изучения этой задачи есть ряд серьезных причин. Первая – техническая: ее решение является удачной комбинацией выработки правильного взгляда на задачу и выбора подходящих средств. Вторая причина существеннее: вам же не хочется оказаться единственным человеком на вечеринке, который не знает, что deposit, dopiest, posited и topside – анаграммы? А если этих причин недостаточно, то в задаче 6 будет описан подобный пример как следствие из прикладной задачи.

Есть несколько удивительно неэффективных и сложных способов решения данной задачи. Любой метод, в котором рассматриваются все перестановки букв в слове, обречен на неудачу. Для слова microphotographic (анаграмма слова photomicrographic) существует 17! перестановок, а  $17! \approx 3 \times 10^{14}$ . Даже допустив адскую скорость – 1 мкс на перестав-

новку, на вычисление потребуется  $3 \times 10^8$  с. Используя приближенное соотношение "1 с равно 1 нановеку", которое верно с точностью до 0.5%: (в столетии округленно  $3.155 \times 10^9$  с), мы получим, что  $3 \times 10^8$  с – это почти десятилетие. И любой метод, в котором сравниваются все пары слов, обречен работать на моей машине по крайней мере в течение ночи, так как в словаре, который я использую, 70 000 слов и даже простое сравнение анаграмм занимает на ней пару дюжин мкс так, что общее время приближенно равно  $70\,000 \text{ слов} \times 70\,000 \text{ сравнений/слово} \times 25 \text{ мкс/сравнений} = 4900 \times 25 \times 10^6 \text{ мкс} = 25 \times 4900 \text{ с} \approx 1.4 \text{ дня}$ . Можете ли вы найти способ обойти обе вышеприведенные ловушки?

*Ага! Озарение:* пометить каждое слово в словаре так, чтобы слова, принадлежащие к одной и той же группе анаграмм, имели бы одинаковую сигнатуру, а потом собрать вместе слова с одинаковыми сигнатурами. Это сводит исходную задачу об анаграммах к двум подзадачам: выбор сигнатур и подбор слов с одинаковыми сигнатурами. Подумайте об этих задачах, прежде чем продолжить чтение.

Для решения первой задачи мы будем использовать сигнатуру, основанную на сортировке<sup>1</sup>: упорядочим буквы в слове по алфавиту. Сигнатурой слова *deposit* является *deiopst*, которая является также сигнатурой слова *dopiest* и любого другого слова из этой группы анаграмм. Таким образом решена и вторая задача: отсортировать слова по порядку их сигнатур. Лучшее описание этого алгоритма из тех, что я слышал, – размахивание руками Т. Каргилла: "Отсортируйте это так" (горизонтальное движение рукой), "а затем так" (вертикальный взмах). Реализация этого алгоритма описана в разд. 2.8.

## 2.5. ОСНОВНЫЕ ПРИНЦИПЫ

*Сортировка.* Наиболее очевидное приложение сортировки -- получить упорядоченную выходную информацию либо как часть системной спецификации, либо в качестве данных для другой программы (возможно, для той, которая использует двоичный поиск). Но в примере с анаграммой упорядочение само по себе не представляло интереса; мы упорядочивали буквы в слове, чтобы собрать вместе одинаковые элементы (в данном случае – сигнатуры). Однако такие сигнатуры являются другим примером применения сортировки: упорядочение букв в слове приводит слова к канонической форме в группе анаграмм.

---

<sup>1</sup> Этот алгоритм для анаграмм был независимо открыт многими людьми еще в середине 60-х годов. Дополнить свои знания об анаграммах и аналогичных задачах со словами можно, прочитав октябрьский номер журнала "Scientific American" за 1984 г. (колонка "Computer Recreation").

Введя дополнительные ключи для каждой записи и сортируя по этим ключам, можно использовать программу сортировки как средство для перегруппировки данных; это особенно эффективно, когда имеешь дело с большими объемами данных на магнитных лентах – см. упражнения с 8–24 из гл. 5 в книге Кнута "Сортировка и поиск". Мы будем несколько раз возвращаться к теме сортировки в ч. III.

*Двоичный поиск.* Алгоритм поиска элемента в упорядоченной таблице в высшей степени эффективен и может быть использован и с ОЗУ, и с диском; его единственный недостаток в том, что вся таблица должна быть отсортирована заранее. Стратегия, лежащая в основе этого алгоритма, используется и во многих других прикладных задачах.

*Сигнатуры.* Когда с помощью отношения эквивалентности определены классы, полезно определить сигнатуру так, чтобы все объекты, входящие в один класс, имели одну и ту же сигнатуру, а любой другой объект – нет. Упорядочение букв в слове дает одну из сигнатур для группы анаграмм, другие сигнатуры можно получить, отсортировав буквы и представив дубликаты числом повторений (так, сигнатура слова mississippi может быть i4m1p2s4 или i4mp2s4, если удалить 1) или введя вектор из 26 целых чисел для подсчета того, сколько раз появляется каждая буква.

Другие применения сигнатур включают методику ФБР для запоминания отпечатков пальцев и эвристический метод Soundex для идентификации имен, которые звучат одинаково, но пишутся по-разному.

Имя	Сигнатура по методу Soundex
Smith	s530
Smythe	s530
Schultz	s243
Shultz	s432

Кнут описывает метод Soundex в гл. 6 своей книги "Сортировка и поиск".

*Постановка задачи.* В предыдущей главе мы показали: определение того, что пользователь в действительности хочет сделать, является существенным элементом программирования. Тема этой главы – следующий шаг в постановке задачи: какие примитивы мы будем использовать для того, чтобы решить задачу? В каждом случае озарение определяет новую элементарную операцию, что делает задачу тривиальной.

*Перспективы для программиста, решающего задачи.* Хорошие программисты немножко ленивы: они сидят и ждут озарения, а не бросаются

вперед со своей первой идеей. Это, конечно, должно согласовываться со своевременным началом написания программы. Думается, что подлинное искусство – определить этот момент. Умение правильно во всем разобраться приходит только с опытом решения задач и размышления над известными решениями.

## 2.6. ЗАДАЧИ

1. Рассмотрим задачу нахождения всех анаграмм заданного слова. Как бы вы решали эту задачу, если а) заданы слово и словарь; б) пришлось затрачивать время и использовать память для обработки словаря перед ответом на каждый вопрос?
2. Дана магнитная лента, содержащая 1 050 000 20-битовых чисел. Как вы найдете то из них, которое записано на ленте по крайней мере дважды?
3. Мы рассмотрели два алгоритма циклического сдвига вектора, для которых требуется умелое программирование. Запрограммируйте их. Какое участие принимает наибольший общий делитель чисел  $I$  и  $N$  в каждой из этих программ?
4. Некоторые читатели обратили внимание на то, что хотя на работу каждого из трех алгоритмов циклического сдвига затрачивается время, пропорциональное  $N$ , алгоритм с трюком, вероятно, вдвое быстрее, чем алгоритм реверсирования: в нем запоминается и читается каждый элемент массива только один раз, в то время как в алгоритме реверсирования это делается дважды. Я реализовал обе подпрограммы без трюков и обнаружил, что для малых значений  $N$  на выполнение этих программ затрачивается одинаковое процессорное время; при  $N = 380\,000$  – по 14 с. Однако при  $N = 390\,000$  программа реверсирования выполняется 16 с, тогда как программу с трюком я прервал через 1 час. Объясните, почему результаты реальных измерений противоречат простой теории. (Полезные дополнительные сведения: машина имеет ОЗУ емкостью 2 Мбайт, каждый элемент массива занимает 4 байта, а при работе в течение одного часа  $I = 256$ .)
5. Программа циклического сдвига заменяет вектор АВ на вектор ВА; как бы вы преобразовали вектор АВС в вектор СВА? (Это моделирует задачу свопинга (взаимной замены) блоков памяти неодинаковой длины.)
6. Фирма Bell Labs имеет программу "Справочная служба, эксплуатируемая пользователем", которая позволяет предпринимателям находить номера в телефонном справочнике фирмы, используя стандартный телефон с кнопочным набором.

1	2 ABC	3 DEF
4 GHI	5 JKL	6 MNO
7 PRS	8 TUV	9 WXY
*	0 OPER	#

Рис. 2.4

Чтобы найти номер разработчика системы М. Леска (Mike Lesk) надо набрать номер службы, ввести "LESK\*M\*" (т. е. "5375\*6\*"), после чего система сообщит его номер. Одной из проблем, возникающих при использовании такой системы, является то, что различные имена могут кодироваться одними и теми же кнопками; когда это происходит в системе Леска, она запрашивает у пользователя дополнительную информацию. Итак, дан большой файл имен (например, обычный городской телефонный справочник), как бы вы обнаружили эти "ложные совпадения"? (Когда Леск провел этот эксперимент с таким словарем, он обнаружил, что их число едва составляет 0.2 %.) Как бы вы написали программу, которой на входе дается код имени с клавиатуры телефона, а она выдает либо имя пользователя, либо соответствующее сообщение?

7. В начале 60-х годов В. Висоцки работал с программистом, который должен был транспонировать матрицу размером  $4000 \times 4000$  элементов, хранящуюся на магнитной ленте (все записи состояли из нескольких дюжин байтов и имели одинаковый формат). Первый вариант программы, предложенный его коллегой, потребовал бы для работы 50 ч; как Висоцки смог сократить время работы до  $1/2$  ч?
8. [Дж. Ульман] Дано множество  $N$  действительных чисел, действительное число  $T$  и целое число  $K$ . Как быстро определить, существует ли для этого множества подмножество из  $K$  элементов, сумма которых не больше  $T$ ?
9. Последовательный поиск и двоичный поиск соответствуют компромиссу между временем поиска и временем предварительной подготовки. Сколько циклов двоичного поиска необходимо выполнить в подготовленной заранее таблице из  $N$  элементов, чтобы окупить время подготовки, затраченной на сортировку этой таблицы?
10. В первый день работы нового сотрудника Т. Эдисон попросил его сотрудника вычислить объем пустой колбы электролампы. Проведя нес-

сколько часов в вычислениях с кронциркулем в руках, он вернулся с ответом – 150 см<sup>3</sup>. После нескольких секунд вычислений Эдисон ответил "ближе к 155". Как он узнал? Дайте другие примеры *ага!* *Озарения* в области аналоговых вычислений.

## 2.7. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Семитомник Кнута "Искусство программирования для ЭВМ" является фундаментальным научным трудом в области алгоритмов. Пока появились три из семи томов: "Основные алгоритмы" (М.: Мир, 1976), "Получисленные алгоритмы" (М.: Мир, 1976), "Сортировка и поиск" (М.: Мир, 1978). Эти энциклопедические труды содержат фактически все, что известно по их тематике на момент издания. Большинство алгоритмов приведены в виде программ на ассемблере, что раскрывает многие проблемы, возникающие при реализации их в виде программ.

Книга Седжуика "Алгоритмы" (Sedgewick. Algorithms, Addison-Wesley, 1983) является прекрасным учебником для изучающих этот предмет. В ней описаны многие более современные или еще не описанные Кнудом алгоритмические аспекты. Интуитивный подход, принятый в этой книге, хорош на практике для программистов, занятых программированием алгоритмов.

## 2.8. РЕАЛИЗАЦИЯ ПРОГРАММЫ ДЛЯ АНАГРАММ (ДОПОЛНЕНИЕ)<sup>1</sup>

Я написал свою программу для анаграмм в операционной системе UNIX, которая особенно удобна для решения данной задачи. После того, как вы ознакомитесь с моей программой, подумайте о том, как бы вы написали аналогичную программу в операционной системе, которую предпочитаете. Программа разбита на три подпрограммы конвейера, где выходные данные каждой из них поступают на вход следующей. Первая программа формирует сигнатуры слов, вторая сортирует файл сигнатур, а третья сжимает слова, образующие группу анаграмм, в одну строку. Вот этот процесс для словаря из шести слов:

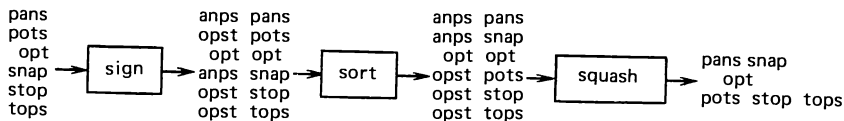


Рис. 2.5

<sup>1</sup> В журнале Communications of the ACM дополнения отделены от текста колонки. Они часто расположены в виде заметок на полях. Хотя дополнения не являются неотъемлемой частью колонки, в них содержится дальнейшее развитие материала. В этой книге дополнения приводятся в последнем разделе главы и помечаются словом (дополнение).



На выходе имеется три группы анаграмм.

В нижеприведенной программе формирования сигнатур SIGN предполагается, что ни одно слово не состоит более чем из 100 букв, и что во входном файле содержатся только строчные буквы и признаки начала новой строки. (Поэтому я предварительно обработал системный словарь с помощью состоящей из одной строки программы трансляции символов, чтобы перевести символы верхнего регистра в нижний регистр.)

```
#define WORDMAX 101

main()
(  char thisword[WORDMAX], sig[WORDMAX];
    while (scanf("%s",thisword) != EOF) {
        strcpy(sig, thisword);
        qsort(sig, strlen(sig), 1, compchar);
        printf("%s %s\n", sig, thisword);
    }
)
```

В цикле while последовательность символов читается в массив thisword до тех пор, пока не будет встречен признак конца файла. Программа strcpy копирует введенное слово в слово sig, в котором потом осуществляется сортировка символов, для чего вызывается системная программа сортировки qsort (параметрами являются имя массива, который требуется отсортировать, его длина, длина элемента в байтах и имя программы для сравнения двух элементов). Наконец, оператор printf распечатывает сигнатуру, за которой следует само слово и признак новой строки \n.

Системная программа sort собирает вместе все слова с одинаковыми сигнатурами; программа squash распечатывает их потом в одну строку. Для этого требуется только три строки на языке AWK, разработанном для выполнения операций с файлами:

```
$1 != prev    ( prev = $1: if (NR > 1) printf "\n" )
              ( printf "%s ", $2 )
END           ( printf "\n" )
```

Основной объем работы выполняется вторым оператором; для каждой входной строки он выводит второе поле (\$ 2), а за ним – пробел. В первой строке ”вылавливаются” изменения: если первое поле (\$ 1) стало отлично от значения `prev` (предыдущего значения этого поля), тогда значение `prev` переопределяется и, если эта запись не первая в файле (`NR` – номер текущей записи), выводится символ перевода строки. К третьей строке обращаются в конце файла для вывода последнего символа перевода строки.

Проверив эти простые программы на маленьких входных файлах, я создал список анаграмм, введя с клавиатуры

```
sign <dictionary : sort : squash >gramlist
```

Эта команда обеспечивает передачу файла `dictionary` на вход программы `sign`, пересылает выходные данные из программы `sign` в программу `sort`, а выходные данные программы `sort` в программу `squash` и записывает выходные данные программы `squash` в файл `gramlist`. Для всей этой работы потребовалось пять выполняемых строк на языке Си, три строки на языке AWK и одна командная строка. Такая краткость достигнута потому, что система UNIX содержит мощный набор языков и обеспечивает удобный механизм для взаимной связи программ, написанных на разных языках. Эта программа работала 27 мин (6 мин – `sign`, 6 мин – `sort` и 15 мин – `squash`). Я мог бы сократить общее время работы вдвое, написав программу `squash` на языке Си (интерпретирующий язык AWK, как обычно, на порядок медленней, чем язык Си), но, так как это одноразовая программа, не стоило тратить на нее таких усилий.

Я запускал программу со стандартным системным словарем, содержащим 71 887 слов. Однако в него не включены многие окончания `-s` и `-ed`. Приведенные ниже группы анаграмм были среди наиболее интересных.

```
subessential suitableness  
canter centra nectar recant trance  
caret cater crate react recta trace  
destain instead sainted stained  
adroitly dilatory idolatry  
eathling haltering lathering  
least setal slate stale steal stela teals  
reins resin rinse risen serin siren
```

Большинство программистов видели программы, работающие со структурами данных, а многие хорошие программисты представляют себе, как написана хотя бы одна из них.

Это огромные, беспорядочные, безобразные программы, которые на самом деле должны быть короткими, простыми и ясными. Я видел однажды программу на Коболе, в которой были такие строки:

```
IF THISINPUT IS EQUAL TO 001 ADD 1 TO COUNT001
```

```
IF THISINPUT IS EQUAL TO 002 ADD 1 TO COUNT002
```

```
....
```

```
IF THISINPUT IS EQUAL TO 500 ADD 1 TO COUNT500
```

Хотя на самом деле эта программа решает более сложную задачу, не будет ошибкой считать, что в ней вычисляется, сколько раз в файле встречается каждое целое число от 1 до 500. В ней более 1600 строк: 500 строк, чтобы определить переменные от COUNT001 до COUNT500; приведенные выше 500 строк – для подсчета, еще 500 строк, для того чтобы напечатать, сколько раз было найдено каждое число и 100 прочих операторов. Задумайтесь на минуту, как бы вы решили ту же самую задачу с помощью гораздо более короткой программы, используя другую структуру данных – массив из 500 элементов вместо 500 отдельных переменных. (Программисты, которым платят за каждую строку программы, возможно, откажутся выполнять это упражнение, а администраторы, которые платят за каждую строку, вероятно, упадут в обморок.)

Следовательно, заголовок этой главы можно понимать так: правильное представление данных способствует написанию структурированных программ. В этой главе описывается несколько программ среднего размера, которые стали короче (и лучше) в результате структуризации их внутренних данных. Обычно программы сокращались с нескольких тысяч до нескольких сотен строк. Эти принципы применимы также к большим программным системам: мы увидим, как надлежащая разработка структуры данных помогла сократить трудоемкость создания одной системы с 250 до 80 чел-ч.

#### 3.1. ПРОГРАММА ОБРАБОТКИ РЕЗУЛЬТАТОВ ОБСЛЕДОВАНИЯ

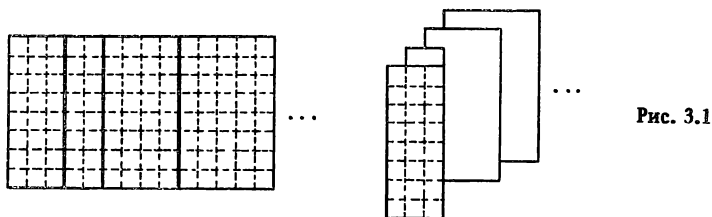
Следующая программа, которую мы рассмотрим, проводит итоговую обработку примерно 20 000 анкет, заполненных учащимися колледжей. Небольшой фрагмент ее выходных данных выглядит так:

	Всего	Гражд. США	Пост. виза	Врем. виза	Муж.	Жен.
Афро-амер.	1289	1239	17	2	684	593
Мекс-амер.	675	577	80	11	448	219
Амер. индейцы	198	182	5	3	132	64
Испан.	411	223	152	20	224	179
Азаитск.	519	312	152	41	247	270
Кавказск.	16272	15663	355	33	9367	6836
Прочие	225	123	78	19	129	92
Итого	19589	18319	839	129	11231	8253

Для каждой этнической группы число мужчин плюс число женщин несколько меньше, чем значение, указанное в строке "Всего", так как некоторые люди не ответили на какие-то вопросы. В действительности выходная информация была более сложной. Я привел все семь строк плюс строку "Итого", но только шесть столбцов, соответствующих суммарному количеству и двум другим категориям – гражданству и полу. В реальной задаче было двадцать пять столбцов, которые соответствовали восьми категориям, а также три страницы с выходной информацией: по одной для двух отдельных колледжей и одна для их суммы. Кроме того, печаталось несколько других тесно связанных таблиц, содержащих число учащихся, которые уклонились от ответа на каждый из вопросов. Любая анкета представляла собой перфокарту, в первом столбце которой указывалась этническая группа, закодированная целым числом от 1 до 8 (для семи категорий и отказа), во втором столбце – гражданство и т.д. до девятого столбца.

По блок-схеме, полученной от системного аналитика, программист стал составлять программу на Коболе. После двухмесячных трудов и написания 1000 строк он прикинул, что сделана половина работы. Я понял его затруднения, взглянув на пятистраничную блок-схему: программа строилась на 350 отдельных переменных – 25 столбцов, умноженных на 7 строк, умноженные на 2 страницы. Кроме описания переменных, в программе была тема логических условий, с помощью которых определялось, какую из переменных увеличить после чтения каждой входной записи. Задумайтесь на минуту о том, как бы вы написали эту программу.

Определяющее решение состоит в том, что числа должны храниться в массиве. Следующее решение более трудное: должен ли массив соответствовать структуре выходной информации (т. е. быть трехмерным – колледж, этническая группа и 25 столбцов) или структуре входной информации (т.е. быть четырехмерным – колледж, этническая группа, категория и значение в пределах категории)? Не учитывая размерность массива, необходимого для данного колледжа, эти подходы можно проиллюстрировать так:



Оба подхода работоспособны; в моей программе трехмерная структура привела к несколько большей работе при чтении данных и меньшей – при их выводе. В программе было 150 строк на Фортране: 80 – для формирования таблиц, 30 – для реализации описанных мной выходных данных и 40 – для создания других таблиц.

Программа, выполнявшая расчет, и программа обработки результатов обследования были слишком длинными; обе содержали многочисленные переменные, которые были заменены единственным массивом. Уменьшение длины текста программы на порядок привело к написанию правильных программ, которые были быстро разработаны и которые легко тестировать и сопровождать. И хотя в обеих прикладных задачах это не имеет большого значения, обе программы были более эффективны по времени работы и занимаемой памяти, чем их первоначальные варианты.

Почему программисты пишут длинные программы, когда можно написать короткие? Одной из причин является отсутствие у них разумной лени, которая упоминалась в разд. 2.5. Они бросаются реализовывать первую же идею, пришедшую им в голову. Но в обоих описанных случаях была более серьезная проблема: эти программисты подходили к решению задач с позиций языка Кобол, а многие программирующие на Коболе представляют себе массивы как фиксированные таблицы, которые инициализируются при начале работы программы и никогда не меняются.

Есть много других причин, по которым программисты делают такие ошибки. Когда я готовился писать эту главу, то нашел похожий пример в своей собственной программе обработки результатов обследования. В основном цикле ввода было 40 строк, он состоял из восьми блоков по

пять операторов, и первые два из этих блоков могли бы быть записаны так:

```
if InputColumn(2) = Refused then
    add 1 to Declined(EthnicGroup, 1)
else
    ThisIndex := 1 + InputColumn(2)
    add 1 to Count(Campus, EthnicGroup, ThisIndex)
if InputColumn(3) = Refused then
    add 1 to Declined(EthnicGroup, 2)
else
    ThisIndex := 4 + InputColumn(3)
    add 1 to Count(Campus, EthnicGroup, ThisIndex)
```

Упомянутые сорок строк я должен был заменить шестью, задав массив Offset с начальными значениями 1, 4, 6, . . .

```
for I := 1 to 8 do
    if InputColumn(I+1) = Refused then
        add 1 to Declined(EthnicGroup, I)
    else
        ThisIndex := Offset(I) + InputColumn(I+1)
        add 1 to Count(Campus, EthnicGroup, ThisIndex)
```

Уменьшив длину текста программы на порядок, я был настолько доволен собой, что пропустил другую возможность сокращения, которая бросалась в глаза.

### 3.2. ФОРМИРОВАНИЕ ПИСЕМ

Следующая программа для каждой записи файла, в котором содержатся имена и адреса, формирует письмо в соответствии с заданным шаблоном. Например, пусть дана запись с восьмью полями

Паблик:Джон К.:Мистер:600:Мэпл Стрит:ЙоурТоун:Айова:12345

(где символ ; разделяет поля в записи). Тогда программа должна сформировать письмо, которое начинается так:

Мистер Джон К. Паблик

600 Мэпл Стрит

ЙоурТоун, Айова 12345

Дорогой Мистер Паблик:

я уверен, что семья Паблик сильно озабочена тем,  
чтобы стать первой семьей на Мэпл Стрит, имеющей  
управляемую микропроцессором веревку для развешивания  
и сушки белья.

Описанная задача типична для многих случаев, отличных от составления писем: небольшой объем вычислений и уйма выходной информации.

Нетерпеливым программистам, вероятно, захочется написать программу, которая начинается так:

Цикл до конца входного файла

```
read LastName, FirstName, Title, StreetNum,  
    StreetName, Town, State, Zip
```

Установить начало страницы

```
print Title, " ", FirstName, " ", LastName
```

```
print StreetNum, " ", StreetName
```

```
print Town, " ", State, " ", Zip
```

```
print
```

```
print "Дорогой", Title, " ", LastName
```

```
print
```

```
print "я уверен, что семья ", LastName, " сильно озабочена тем,"
```

```
print "чтобы стать первой семьей на ", StreetName, ", имеющей"
```

```
print "управляемую микропроцессором веревку для развешивания"
```

```
print "и сушки белья."
```

Такая программа скучна при использовании языка, в котором можно задавать строки переменной длины в операторах ввода-вывода, и отвратительна для языка, не имеющего таких средств. В любом случае сопровождать эту программу ужасно: добавление нескольких слов в начало абзаца может повлечь за собой утомительное переформирование всего абзаца вручную.

Более элегантный подход предполагает написание "генератора писем", которому на входе задается шаблон письма, подобный следующему:

\$3 \$2 \$1  
\$4 \$5  
\$6, \$7 \$8

Дорогой \$3 \$1:

я уверен, что семья \$1 сильно озабочена тем,  
чтобы стать первой семьей на \$5, имеющей  
управляемую микропроцессором веревку для развешивания  
и сушки белья.

Обозначение \$i соответствует i-му полю записи входного файла, поэтому \$1 – фамилия и т. д. Этот шаблон интерпретируется приведенной ниже программой на псевдоязыке, в которой предполагается, что символ \$ записан во входном шаблоне как \$\$.

```
read Schema from schema file
/* Читать шаблон из файла Schema */
loop until end of input file
/* Цикл до конца входного файла */
  for I: = 1 to NumFields do
    read Field[I] from input file
    /* Читать поле из входного файла */
  skip to new page
/* Перейти на новую страницу */
```



```

loop from start to end of Schema
/* Цикл по шаблону */
    C := next character in Schema
    /* Следующий символ шаблона */
    if C ≠ "$" then
        /* Если он не "$", то */
        printchar C
        /* Напечатать его */
    else
        C := next character in Schema
        /* Следующий символ шаблона */
        case C of
            "$":    printchar "$"
            "1"-"9": printstring Field[C]
            others:  Error("Ошибка в шаблоне")

```

Шаблон представлен в программе как длинный массив символов, в котором строки текста заканчиваются символом "новая строка". Как и в предыдущем примере интерпретатор легко реализовать, если в языке программирования поддерживаются строки переменной длины, и сложнее, если это не так. Но даже в этом случае строки переменной длины обрабатываются теперь в одном месте, а не при каждом их появлении в шаблоне.

При использовании языка, в котором поддерживаются строки переменной длины при выводе, может потребоваться больше усилий, чем при составлении очевидной программы. Однако часто эта дополнительная работа может окупиться сполна: если письмо переделывается, то редактором текста можно изменить шаблон, что, конечно, облегчит подготовку письма по другому шаблону.

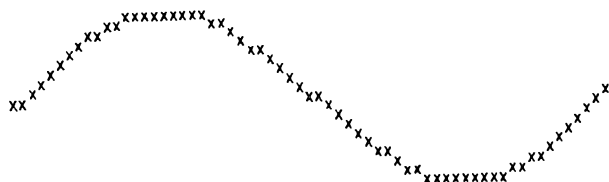
Концепция шаблона могла бы сильно упростить программу из 5300 строк на Коболе, сопровождением которой мне пришлось однажды заниматься. Входными данными этой программы было описание финансового положения семьи; ее выходом был буклет, резюмирующий это положение и рекомендуемый линию поведения на будущее. Вот некоторые цифры: на входе 120 полей на 10 картах; на выходе 400 строк на 18 страницах; 300 строк программы для очистки входных переменных; 800

строк для вычислений и 4200 строк для выдачи выходной информации. В одном месте программы 113 строк использовалось для того, чтобы вывести по центру листа текст, длина которого варьировалась от 1 до 10 символов. Я прикинул, что 4200 строк, использовавшихся для вывода, можно заменить интерпретатором не более пары сотен строк на Коболе и написание программы с самого начала в такой форме привело бы к сокращению числа строк программы на одну треть от ее первоначального размера, что облегчило бы сопровождение программы.

### 3.3. ПРИМЕРЫ

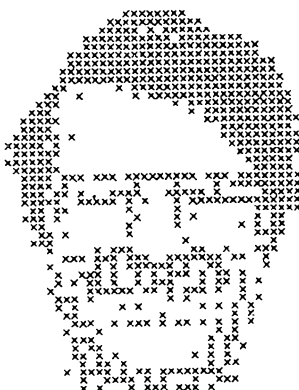
*Вывод графиков на устройство печати.* Включение в программу нескольких операторов печати – почти всегда плохой способ решения ”освященной веками” задачи рисования забавных картинок в виде матрицы символов размерностью  $66 \times 132$  (или  $24 \times 80$  и т. д.). Некоторые виды структур могут быть представлены коротким математическим выражением. Следующая синусоида во многих системах может быть описана несколькими строками программы:

Рис. 3.2



Изображения с менее регулярной структурой могут быть описаны массивом характерных точек. Произвольные изображения лучше всего представлять массивом; приведем массив  $48 \times 48$  из пробелов и буквы X:

Рис. 3.3



В задаче 3 для запоминания изображений используется совокупность описанных представлений.

*Сообщения об ошибках.* В неряшливо разработанных системах можно встретить сотни сообщений об ошибках, рассеянных по тексту программы, перемешанных с другими операторами печати; в аккуратных системах доступ к ним осуществляется через единственную подпрограмму. Рассмотрим теперь сложности ответа на следующие запросы при "аккуратной" и "неряшливой" организациях: составить список всех возможных сообщений об ошибках; изменить каждое сообщение об ошибке, где требуется вмешательство оператора, так, чтобы раздавался сигнал на пульте оператора; и перевести все сообщения об ошибках на французский или немецкий язык.

*Подпрограммы для работы с датами.* Дан год и день с начала года, требуется иметь на выходе из подпрограммы месяц и день месяца; например, 61-й день 1988 года – это первый день третьего месяца. В своей книге "Элементы стиля программирования" Керниган и Плоджер (Kernighan, Plauger. Elements of Programming Style) (второе издание, с 52-54)<sup>1</sup> для решения этой задачи приводят фрагмент из 55 строк, взятый прямо из текста чьей-то программы. Потом они дают для этой задачи решение из 5 строк, в котором используется массив из 26 целых чисел. В подпрограммах, работающих с датами, возникает ряд вопросов по их представлению, например, определение того, является ли заданный год високосным, как представлять число дней в месяце (см. задачу 4).

*Анализ слов.* В анализе английских слов имеется много задач, связанных с применением компьютеров. Например, в гл. 13 мы рассмотрим программу проверки правописания, в которой для сокращения объема словаря используется анализ суффиксов. Например, хранится только слово "laugh", а не все его различные варианты с окончаниями (-ing, -s, -ed и т. д.). Лингвисты разработали большое количество правил для таких задач. Когда Д. Макилрой писал программу для произнесения отдельных слов с помощью синтезатора речи, то знал, что кодирование – плохое средство для задания таких правил; поэтому он использовал 20 страниц команд и восьмистраничную таблицу. Если кто-нибудь захочет модифицировать эту программу, ничего не добавляя в таблицы, то придется написать 50 дополнительных страниц команд, чтобы увеличить объем

---

<sup>1</sup> Стр. 48-58 книги "Элементы стиля программирования" содержат прекрасную коллекцию маленьких программ, в которых изменение структуры данных приводит к изящному тексту программы. Эта идея в книге Кернигана и Плоджера "Программные средства" уводит далеко от искушения писать беспорядочные тексты: задача программиста – создать изящное и мощное средство, которое пользователь будет затем применять в своих задачах.

работы на 20 %. Макилрой утверждает, что теперь мог бы написать программу с расширенными возможностями, используя менее 20 страниц команд, но добавив дополнительные таблицы. Чтобы испытать свои способности применительно к подобному набору правил, решите задачу 5.

*Задачи, связанные с терминалами.* Разработка правильной структуры данных помогает уменьшить горы команд, которые обычно используются для реализации работы с терминалами. Многие терминалы поддерживают такие операции, как наложение символов на существующее изображение на экране или сдвиг строк на экране без повторной передачи символов. Эти особенности сильно увеличивают скорость и упрощают использование современных редакторов текста. Однако некоторые подходы к использованию таких свойств лишь незначительно лучше, чем их игнорирование: один из подходов – написать отдельную программу для каждого типа терминала; другой – вставить длинный оператор условного перехода везде, где программа осуществляет ввод или вывод. В системе UNIX эта задача решается с помощью базы данных “возможностей терминалов”, которая обеспечивает программисту унифицированный доступ к этим возможностям. В этой базе данных есть абстрактное описание той или иной возможности (такой, как сдвиг строк на экране), а ее реализация на конкретном терминале (последовательность символов, которая задает перемещение строк) скрыта от пользователя; программы, написанные для виртуального терминала, могут работать с любым терминалом, занесенным в эту базу данных.

### 3.4. БОЛЬШАЯ ПО ОБЪЕМУ ПРОГРАММА

Мы рассмотрели ряд структур данных, которые обеспечивают элегантные решения трудных задач. Из этих примеров можно сделать следующий вывод: вклад информатики в математическое обеспечение часто осуществляется посредством простых идей. Но время от времени теоретический прорыв потрясает мир практиков, и эта глава не будет полной без описания такого прорыва в области структур данных. Я расскажу сокращенный вариант истории, которая полностью изложена А. Дж. Гоулдштейном в его блестящей статье “База данных с ориентированным гиперграфом: модель местных линий связи телефонной станции” (A. Jay Goldstein. A directed gipergraph database: a model of the local loop telephone plant) в журнале “The Bell System Technical Journal” 61, 9, ноябрь 1982 (с. 2529-2554).

Телефонные компании относят часть своего оборудования, расположенного вне центрального офиса, к “объектам местных линий связи”. Этот термин применим к кабелям, распределительным оконечным устройствам, кроссово-соединительным устройствам, проводам и т. д. Информация об объектах местных линий связи запоминается в базе

данных с целью удовлетворения запросов потребителей и выполнения различных операций по профилактическому обслуживанию.

Исследовательская группа Bell Labs примерно из 200 человек потратила пять лет на реализацию такой базы данных для телефонных компаний. Около 50 человек были задействованы в течение пяти лет в части этого проекта, касающейся объектов местных линий связи. В завершенной системе эта часть состоит из 155 000 строк на Коболе. Такая система эксплуатировалась телефонной компанией, но имела много недостатков: работала медленно, сопровождать ее было чрезвычайно тяжело, расширять по мере развертывания новой техники почти невозможно. Ретроспективно многие из этих недостатков могли бы быть выведены из структуры данных, на основе которой была построена система, т. е. база данных CODASYL, состоящая из схем по 400 записей. При выводе их на устройство печати получался документ толщиной четыре дюйма.

По ряду причин вся система была переделана в виде трех систем, одна из которых имела дело с оборудованием местных линий связи. Эта система рассмотрена в статье Гоулдштейна. Первой стадией ее проектирования была разработка обоснованной математической модели оборудования местных линий связи. После этого бригада разработчиков создала специальную систему управления базой данных, которая отображала эту модель. Эта модель и эта система представляют собой особый вид базы данных для взаимосвязанных объектов: ориентированный гиперграф, вершины которого являются объектами оборудования местных линий связи (соединители, кабели, персонал и т. д.), а ориентированные ребра обозначают логические взаимосвязи между объектами. Такая структура обеспечивает большую гибкость при добавлении к системе новых устройств и новых соединений устройств друг с другом, а также помогает решать неприятные задачи обработки незавершенных операций, которые запланированы, но еще не выполнены. Новая база данных описывалась сорока схемами, и толщина их распечатки была менее 1/2 дюйма. При использовании этой структуры данных в качестве ядра системы вся система была разработана и поставлена заказчику за три года. В работе участвовало примерно 30 сотрудников.

Новая система несомненно лучше своей предшественницы. Она была разработана примерно за половину календарного времени и вдвое меньшим числом сотрудников, чем старая система. Обслуживать новую систему – одно удовольствие: с объекта, где она эксплуатируется, поступает на порядок меньше сообщений об ошибках в ее работе, а изменения, внесенные раньше за несколько месяцев, теперь могут быть выполнены за несколько дней. Старую систему обслуживали 50 человек, для новой потребовалось 5. Новая система более эффективна по времени работы и ее проще расширять, чтобы включать новую телефон-

ную технику. Эти различия появились при эксплуатации: в то время как старая система так выходила из строя несколько раз в неделю, что требовалась ее перегрузка, в новой системе такие неполадки не возникали два года после ее первоначального тестирования. Такая новая система используется несколькими телефонными компаниями.

Многие факторы способствовали успеху новой системы. Ее разработка начиналась с четкой постановкой задачи и изучения того, что было удачным в предыдущей реализации. Она была написана на современном языке программирования и разработана с использованием методологии сокрытия информации, несущественной для пользователя. Среди многих ее сильных сторон одна упоминалась особенно часто – это точная концептуальная модель и база данных, на которых она построена. В то время как разработчики предыдущей системы приступили к работе, мысля в терминах базы данных CODASYL, и пытались втиснуть задачу в эту среду, вторая группа начала работу, имея твердое представление о данных, и использовала его для структуризации программы. Дополнительная польза этого подхода заключается в том, что разработчики – люди, реализующие систему, и пользователи могут общаться друг с другом на одном языке, связанном с представлением о прикладной области.

### 3.5. ОСНОВНЫЕ ПРИНЦИПЫ

Мораль каждой из этих двух историй одинакова: *не пишите длинные программы, когда можно написать короткие*. Большинство этих структур служат примером того, что Пойа назвал парадоксом изобретателя в своей книге "Как решать задачу", т. е. "бывает, что более общую задачу решить проще". В программировании это означает, что решить задачу непосредственно для 73 случаев может оказаться сложнее, чем написать общую программу для N случаев, а затем применить ее для случая, когда  $N = 73$ .

В этой главе рассматривался только один аспект вклада структур данных в программное обеспечение: сведение длинных программ к коротким. Разработка структур данных может иметь много других положительных воздействий, включая сокращение времени и памяти, увеличение транспортабельности и удобства эксплуатации. В книге Ф. Брукса "Как проектируются и создаются программные комплексы" сформулировано замечание относительно сокращения памяти, которое является хорошим советом для программистов, желающих улучшить и другие характеристики:

Программист, зашедший в тупик из-за отсутствия ресурсов памяти, как правило, выпутается наилучшим образом, вернувшись к началу

своей программы, чтобы поразмышлять над представлением своих данных, ибо представление данных – сущность программирования.

Приведу несколько принципов, которыми следует руководствоваться при возвращении к началу.

*Переделать повторяющиеся коды, представив их в виде массивов.* Длинный фрагмент повторяющихся кодов лучше всего представить в виде простейшей структуры данных – массива.

*Хорошо знать развитые структуры данных.* Развитые структуры данных не всегда адекватны задаче, но когда они действительно нужны, заменить их нельзя.

*Пусть структуру программы определяют данные.* Цель этой главы – показать, что данные могут определить структуру программы, если заменить сложные последовательности команд соответствующими структурами данных. Имеется и много других интерпретаций этого совета. Д. Парнас продемонстрировал, что данные, которые обрабатывает программа, являются предпосылкой для формирования хорошей модульной структуры программы на основе глубокого понимания задачи (см. книгу "О критериях, используемых при декомпозиции системы на модули" (David Parnas. On the criteria to be used in decomposing systems into modules) в декабрьском номере журнала "Communications of the ACM за 1972 г.). Мы вернемся к этой теме при изучении "абстрактных типов данных" в гл. 11, 12. М. Джексон убедительно доказал, что для многих общих задач обработки данных, относящихся к коммерции, доскональное понимание входной и выходной информации может почти автоматически привести к получению текста программы (см. работу "Принципы разработки программ" (Michael Jackson. Principles of Program Design), опубликованную издательством Academic Press в 1975 г.). Итак, хотя и с некоторыми изменениями, основной мыслью остается то, что перед написанием команд хорошие программисты досконально разбираются в структуре входных, выходных и промежуточных данных, на которых строятся их программы.

### 3.6. ЗАДАЧИ

1. Приведенные ниже 25 операторов `if` реализуют в тексте программы возможный способ вычисления федерального налога с доходов в США за 1978 г. Последовательность процентных отчислений 0.14, 0.15, 0.16, 0.17, . . . содержит для больших значений приращения, большие 0.01. Нужны ли дополнительные комментарии?

```

/* Income - доход; Tax - налог */
if Income <= 2200 then
    Tax := 0
else if Income <= 2700 then
    Tax := .14 * (Income - 2200)
else if Income <= 3200 then
    Tax := 70 + .15 * (Income - 2700)
else if Income <= 3700 then
    Tax := 145 + .16 * (Income - 3200)
else if Income <= 4200 then
    Tax := 225 + .17 * (Income - 3700)
    ...
else
    Tax := 53090 + .70 * (Income - 102200)

```

2. Линейное рекуррентное соотношение  $k$ -го порядка определяет ряд следующим образом

$$A_n = C_1 A_{n-1} + C_2 A_{n-2} + \dots + C_k A_{n-k} + C_{k+1},$$

где  $C_1, \dots, C_{k+1}$  — действительные числа. Напишите программу, которая по заданным  $k, A_1, \dots, A_k, C_1, \dots, C_{k+1}$  и  $N$  вычисляет и выдает на выходе  $A_1, \dots, A_N$ . Насколько эта программа сложнее программы для частного случая с рекуррентным соотношением пятого порядка и не использующей массивов?

3. Напишите "образцовую" процедуру, которая для заданной на входе прописной буквы создает массив символов, графически описывающий эту букву.
4. Напишите процедуры для следующих задач обработки дат: даны две даты, вычислить количество дней между ними; дана дата, получить на выходе день недели; даны месяц и год, создать календарь для этого месяца в виде массива символов. В первых версиях своих программ можете предположить, что год задается в пределах 1900 — 1999. Следующие версии должны быть как можно более общими.



5. Эта задача является небольшой частью задачи переноса английских слов. Следующий список правил описывает некоторые правильные способы переноса слов, оканчивающихся буквой *c*:

et-ic al-is-tic s-tic p-tic -lyt-ic ot-ic an-tic n-tic c-tic at-ic h-nic n-ic m-ic l-lic  
b-lic -cl-ic l-ic h-ic f-ic d-ic -bic a-ic -mac i-ac

Эти правила должны применяться в приведенном выше порядке. Таким образом, перенос слова *ethnic* осуществляется по правилу "h-nic", слово *clinic* не подходит для этого правила, но подходит для правила "n-ic". Как надо представить эти правила в подпрограмме, в которой для задаваемого на входе слова возвращается отделенный черточкой суффикс?

6. Напишите генератор писем, достаточно универсальный для того, чтобы интерпретировать рассмотренные нами ранее шаблоны; сделайте свою программу простой, насколько это возможно. Разработайте простые шаблоны и выходные файлы для проверки корректности своей программы.
7. Типичные словари позволяют находить значения слов, а в задаче 1 из гл. 2 описан словарь, позволяющий находить анаграммы слов. Разработайте словари для определения правильного написания слов и для поиска рифмы к слову. Обсудите словари для поиска последовательности целых чисел (такой, как 1, 1, 2, 3, 5, 8, . . .), химического состава, метрической структуры песни.
8. В этой главе в качестве индексов массивов использовались целые числа. В некоторых языках, таких как Снобол и АWK, есть "ассоциативные массивы" с индексами, представляемыми последовательностями символов, поэтому допускаются присваивания, подобные следующему: `count ["cat"] = 7`. Как бы вы использовали и реализовывали такие массивы?
9. [С. К. Джонсон]. Недорогое устройство позволяет с помощью семи сегментов отображать цифры

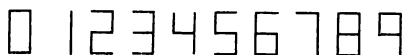


Рис. 3.4

Эти сегменты обычно нумеруются так:

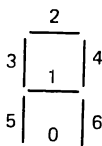


Рис. 3.5

Напишите программу отображения 16-разрядного положительного числа в виде пяти семисегментных цифр. Выходом является массив из пяти байтов; бит I байта J установлен в 1 тогда и только тогда, когда I-й сегмент участвует в отображении J-й цифры.

### 3.7. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Структуры данных тесно связаны с алгоритмами, которые их обрабатывают; учебные пособия по алгоритмам, упоминавшиеся в разд. 2.7, содержат обширную информацию о структурах данных. Помимо этого рекомендую книгу "Методы структуризации данных", написанную Стэндишем (Standish. Data Structure Techniques) и опубликованную издательством Addison-Wesley в 1980 г. , которая является хорошо написанным подробным справочником.

## Глава 4. НАПИСАНИЕ ПРОГРАММ, НЕ СОДЕРЖАЩИХ ОШИБОК

В конце 60-х годов говорилось о подающих надежды программах, которые проверяют корректность других программ. К сожалению, в конце 80-х годов, за несколькими исключениями, мы все еще находимся на стадии разговоров о системах автоматической верификации. Однако, несмотря на несбывшиеся надежды, исследование программ верификации дало нам нечто более ценное, чем представление о черном ящике, который заглатывает программу и зажигает лампочки "хорошая" или "плохая", – понимание основ того, как программировать компьютеры.

Цель этой главы – показать, как понимание основ может помочь программисту писать программы без ошибок. Но перед тем, как перейти непосредственно к этой теме, остановимся на ее значимости. Умение составлять текст программы – это только небольшая часть решения задачи написания правильной программы. Основную часть этой задачи составляют темы трех предыдущих глав: постановка задачи, разработка алгоритма и выбор структуры данных. Если вы хорошо выполните эти задания, то написать правильные команды очень просто.

### 4.1. ДВОИЧНЫЙ ПОИСК "БРОСАЕТ ВЫЗОВ"

Даже для наилучших алгоритмов от программиста иногда требуется умение писать искусные программы. В этой главе рассматривается одна из задач (двоичный поиск), которая требует особенно тщательного написания программы. Повторно рассмотрев эту задачу и составив набросок плана ее решения, воспользуемся при написании программы принципами верификации.

Впервые мы встретились с этой задачей в разд. 2.2; требовалось определить, содержит ли упорядоченный массив  $X[1 \dots N]$  элемент  $T$ . Если формулировать точно, мы знаем, что  $N \geq 0$  и  $X[1] \leq X[2] \leq \dots \leq X[N]$ ; при  $N = 0$  массив пуст. Тип переменной  $T$  совпадает с типом элементов массива; программа на псевдоязыке должна работать одинаково хорошо для целых и действительных чисел, а также для последовательностей символов. Результат должен запоминаться в целой переменной  $P$  (сокращение от слова position – позиция). При  $P = 0$   $T$  не находится в массиве  $X[1 \dots N]$ , иначе  $1 \leq P \leq N$  и  $T = X[P]$ .

Двоичный поиск решает задачу определения текущих границ области, в которой должен находиться элемент  $T$ , если он вообще есть в массиве<sup>1</sup>. Сначала эта область полностью совпадает с массивом. После сравнения элемента из ее середины с элементом  $T$  и отсечения половины области ее объем уменьшается. Этот процесс продолжается, пока элемент  $T$  не будет обнаружен в массиве или пока не станет известно, что область, в которой он должен лежать, пуста. При поиске элемента в таблице из  $N$  элементов выполняется примерно  $\log_2 N$  сравнений.

Большинство программистов думает, что, имея в руках вышеприведенное описание алгоритма, написать программу просто; они не правы. Чтобы вы поверили этому – отложите прямо сейчас эту книгу и попытайтесь написать программу сами.

Я давал эту задачу на курсах для сотрудников фирм Bell Labs и IBM. Профессиональные программисты потратили два часа для превращения приведенного выше описания в программу на языке, который они выбрали сами. Тексты на языках высокого уровня были прекрасны. По окончании установленного времени почти все программисты сообщили, что у них готовы правильные программы для решения этой задачи. После этого мы потратили 30 мин на проверку их программ (программисты делали это на тестовых примерах). В нескольких группах, состоящих более чем из 100 программистов, результаты менялись мало: 90 % программистов нашли ошибки в своих программах (и я не уверен в корректности тех программ, в которых ошибки не были найдены.)

Я был изумлен: времени было достаточно, но только 10 % профессиональных программистов были способны написать такую короткую программу правильно. Но они не были единственными, для кого эта задача оказалась сложной: в истории из разд. 6.2.1. книги Кнута "Сорти-

---

<sup>1</sup> Б. Маккиман из института повышения квалификации фирмы Wang отметил, что такая формулировка задачи устраняет необходимость решения нескольких задач, обычно возникающих при программировании двоичного поиска. Формальное решение для похожего описания приведено в разделе Programming Pearls июльского номера журнала Communications of the ACM за 1984 г., с. 631.

ровка и поиск” автор обратил внимание на то, что, хотя алгоритм двоичного поиска впервые был опубликован в 1946 г., первая публикация программы двоичного поиска, не содержащая ошибок, не появилась до 1962 г.

#### 4.2. НАПИСАНИЕ ПРОГРАММЫ

Основная идея двоичного поиска состоит в том, что мы всегда знаем, что если элемент  $T$  вообще есть в массиве  $X [1 \dots N]$ , то он должен быть в некоторой области, принадлежащей  $X$ . Будем использовать запись *Должно Быть (область)* для обозначения того, что если элемент  $T$  есть в массиве, то он должен быть в этой области. Мы можем использовать такую запись для преобразования приведенного выше описания процедуры двоичного поиска в набросок программы:

Начальная установка области от 1 до  $M$

Цикл

( Инвариант: ДолжноБыть (область) )

Если область пуста,

    выход, так как  $T$  нет в массиве

Вычислить  $M$  - середину области

Использовать  $M$  как "зонд" для сокращения области

    Если  $T$  найдено в процессе сокращения,

        сообщить его позицию

Основополагающей частью этой программы является инвариант цикла, который заключен в фигурные скобки. Это утверждение о состоянии программы названо инвариантом потому, что оно верно в начале и в конце каждого прохода по циклу и формализует приведенную выше интуитивную запись.

Углубимся теперь в тонкости программы и постараемся, чтобы все действия не нарушали инвариантности. Первая проблема, с которой мы должны столкнуться, – представление области: чтобы задать область  $L \dots U$ , будем использовать два индекса  $L$  и  $U$  (соответственно lower – нижний; upper – верхний). (Имеются и другие возможные представления области, например, можно задать ее начальную позицию и длину.) Следующий шаг – начальная установка; какие значения должны иметь  $L$  и  $U$ , чтобы запись *Должно Быть ( $L, U$ )* имела значение "истина"? Очевидный выбор:  $L = 1$  и  $U = N$ . *Должно быть ( $1, N$ )* указывает на

то, что если элемент  $T$  находится в  $X$ , то он находится и в массиве  $X[1 \dots N]$ . Это как раз то, что нам известно перед началом работы программы. Поэтому начальная установка состоит в присваивании  $L := 1$  и  $U := N$ .

Следующие шаги – проверка, пуста ли область  $L \dots U$ , и вычисление новой средней точки  $M$ . Область  $L \dots U$  пуста, если  $L > U$ ; в этом случае мы запоминаем в переменной  $P$  значение 0 и заканчиваем цикл, поэтому можно записать

```
if L > U then
    P := 0; break
```

Оператор выхода `break` прерывает выполнение цикла. А для нахождения середины области служит оператор

```
M := (L+U) div 2
```

Оператор `div` реализует деление нацело: и  $6 \text{ div } 2$ , и  $7 \text{ div } 2$  равно 3. Программа теперь выглядит так:

```
L := 1; U := N
```

Цикл

```
( Инвариант: ДолжноБыть(L,U) )
```

```
if L > U then
```

```
    P := 0; break
```

```
M := (L+U) div 2
```

Использовать  $M$  как "зонд" для сокращения области

Если  $T$  найдено в процессе сокращения,

сообщить его позицию и завершить работу

Подробная запись трех последних строк будет включать сравнение  $T$  с  $X[M]$  и выполнение соответствующего действия, чтобы сохранить значение инварианта. Эта часть программы будет иметь следующий вид:

case

$X[M] < T$ : Действие А

$X[M] = T$ : Действие В

$X[M] > T$ : Действие С

Что касается действия В, то мы знаем, что позиция Т совпадает с М, поэтому мы пересылаем М в Р и осуществляем выход (break) из цикла. Так как остальные два случая симметричны, сосредоточимся на первом и положимся на то, что третий будет логически выведен из симметрии (это одна из причин того, что в следующем разделе мы будем тщательно проверять эту программу):

Если  $X[M] < T$ , то мы знаем, что  $X[1] \leq X[2] \leq \dots \leq X[M] < T$ , поэтому Т не может быть в массиве  $X[1 \dots M]$ . С другой стороны, Т находится не вне массива  $X[L \dots U]$ , и ясно, что если Т вообще где-либо находится, он должен быть в массиве  $X[M+1 \dots U]$ . Это можно записать как *Должно Быть* ( $M+1, U$ ). Переопределим инвариант *Должно Быть* ( $L, U$ ), заменив L на M+1. Добавление этого оператора выбора к полученному ранее наброску программы дает окончательный вариант программы:

```
L := 1; U := N
loop
  ( ДолжноБыть(L,U) )
  if L > U then
    P := Ø; break
  M := (L+U) div 2
  case
    X[M] < T:  L := M+1
    X[M] = T:  P := M; break
    X[M] > T:  U := M-1
```

Это короткая программа: десять строк команд и одно утверждение инвариантности. Основная методика проверки программы – точно сформулировать инвариант и внимательно следить за сохранением инварианта при написании каждой строки программы – очень помогла нам в превращении наброска алгоритма в текст на псевдоязыке. Этот

процесс внушает некоторое доверие к программе, но мы никоим образом не уверены в ее правильности. Чтобы убедиться в том, что программа выполняет правильные действия, затратьте несколько минут перед тем, как продолжить чтение.

#### 4.3. РАЗБОР ПРОГРАММЫ

Когда я сталкиваюсь с тонкой программистской задачей, то пытаюсь довести программу до такого уровня детализации, который мы только что видели. Чтобы убедиться в том, что в ней нет ошибок, я пользуюсь методиками верификации. Тестирование на таком уровне мы будем применять в гл. 8, 10 и 12.

В этом разделе мы рассмотрим доказательство правильности программы двоичного поиска на высшем уровне детализации (на практике я выполняю гораздо менее формальный анализ). В нижеприведенный вариант программы включены пространные (даже слишком) утверждения, которые формализуют интуитивную запись, использованную нами при первоначальном написании программы.

В то время как разработка программы шла сверху вниз (от общей идеи через ее детализацию к конкретным строкам программы), анализ ее правильности будет вестись снизу вверх: мы будем рассматривать, как отдельные строки программы совместно работают для решения задачи.

#### *ПРЕДУПРЕЖДЕНИЕ.*

*Впереди скучный материал.*

*При признаках сонливости  
перейти к разд. 4.4.*

Начнем со строк 1 – 3. Утверждение в строке 1 *Должно Быть* (1,  $N$ ) верно по определению: если элемент  $T$  находится в массиве, он должен быть в массиве  $X[1 \dots N]$ . Поэтому операции присваивания в строке 2:  $L := 1$  и  $U := N$  обеспечивают справедливость утверждения в строке 3: *Должно Быть* ( $L, U$ ).

Теперь мы подошли к сложному фрагменту программы – циклу в строках 4 – 27. Доказательство его правильности состоит из трех частей, каждая из которых тесно связана с инвариантом цикла.

*Начальная установка.* Инвариант верен при первом выполнении цикла.

*Сохранение.* Если инвариант верен в начале итерации и выполняются операторы цикла, то значение инварианта сохраняется истинным после завершения выполнения операторов цикла.

*Завершение.* После завершения цикла результат сохраняется (в данном случае требуется, чтобы переменная Р имела соответствующее значение). Для демонстрации этого мы будем использовать факты, установленные посредством инвариантности.

Отмечаем для случая начальной установки, что в строке 3 утверждается то же самое, что и в строке 5. Для доказательства двух других свойств мы должны логически проанализировать программу от строки 5 до строки 27. При рассмотрении строк 9 и 21 (операторы выхода break) мы устанавливаем свойства завершения, а при прохождении всего пути до строки 27 можно установить свойство сохранения, так как строка 27 такая же, как строка 5.

```
1.      { ДолжноБыть (1,N) }
2.      L := 1; U := N
3.      { ДолжноБыть (L,U) }
4.      loop
5.          { ДолжноБыть (L,U) }
6.          if L > U then
7.              { L > U и ДолжноБыть (L,U) }
8.              { Т в массиве нет }
9.              P := Ø; break
10.         { ДолжноБыть (L,U) и L <= U }
11.         M := (L+U) div 2
12.         { ДолжноБыть (L,U) и L <= M <= U }
13.         case
14.             X[M] < T:
15.                 { ДолжноБыть (L,U) и НеМожетБыть (1,M) }
16.                 { ДолжноБыть (M+1,U) }
17.                 L := M+1
18.                 { ДолжноБыть (L,U) }
19.             X[M] = T:
20.                 { X[M] = T }
```



```

21.                P := M; break
22.                X[M] > T:
23.                { ДолжноБыть (L,U) и НеМожетБыть (M,N) }
24.                { ДолжноБыть (L,M-1) }
25.                U := M-1
26.                { ДолжноБыть (L,U) }
27.                { ДолжноБыть (L,U) }

```

Выполнение проверяемого в строке 6 условия приводит к утверждению в строке 7: если элемент T вообще есть в массиве, то он должен быть между L и U, а  $L > U$ . Из этих утверждений следует утверждение строки 8: элемента T в массиве нет. Таким образом, мы корректно завершили цикл в строке 9, установив переменную P в 0.

Если условие строки 6 не выполняется, то переходим к строке 10. Инвариант по-прежнему сохраняется (мы ничего не делали для его изменения), и так как проверяемое условие не выполнено, то  $L \leq U$ . В строке 11 переменной M присваивается среднее арифметическое для L и U значение, округленное до ближайшего целого числа. Так как оно всегда находится между этими двумя величинами и округление не может сделать его меньше L, получаем утверждение, приведенное в строке 12.

При анализе оператора case в строках 13 – 27 рассматривается каждый из трех возможных случаев. Для анализа проще всего второе условие в строке 19. Вследствие утверждения строки 20 мы вправе установить значение переменной P равным M и завершить цикл. Это второе из двух мест программы, где завершается цикл, и в обоих случаях он заканчивается корректно. Таким образом, мы установили правильность завершения цикла.

Перейдем теперь к двум симметричным ветвлениям оператора case. Так как при разработке программы основное внимание мы уделили первому переходу, обратимся теперь к строкам 22 – 26. Рассмотрим утверждение в строке 23. Его первая составляющая – инвариант, который программа не меняет. Второй оператор истинен, так как  $T < X[M] \leq X[M+1] \leq \dots \leq X[N]$ . Следовательно, известно, что индекс элемента T в массиве не может превышать M – 1. Это выражено с помощью сокращенной записи *НеМожетБыть (M,N)*. Логика подсказывает нам, что если элемент T должен находиться между позициями L и U и не может совпадать с M или иметь индекс, превышающей M, то он должен находиться между L и M – 1 (если он вообще есть в X); отсюда следует утверждение строки 24. Выполнение

строки 25 при условии, что утверждение строки 24 верно, сохраняет истинность строки 26 – это определение операции присваивания. Таким образом, эта ветвь оператора case переопределяет инвариант в строке 27.

Доказательство для строк 14 – 18 имеет в точности такой же вид, как и доказательство для всех трех переходов по оператору case. Один из них корректно завершает цикл, а два других сохраняют инвариант.

Анализ программы показывает, что при завершении цикла переменная  $P$  имеет правильное значение. Однако цикл может оказаться бесконечным; действительно, это было наиболее распространенной ошибкой в программах, написанных профессиональными программистами.

В доказательстве конечности алгоритма используется другая часть области  $L \dots U$ . Первоначально эта область состоит из некоторого конечного числа элементов ( $N$ ), а строки 6 – 9 гарантируют, что цикл завершится, когда в области будет меньше одного элемента. Поэтому, чтобы доказать конечность алгоритма, нужно показать, что область сокращается при каждом проходе по циклу. В строке 12 нам сообщается, что индекс  $M$  всегда находится в текущей области. Оба зацикленных ветвления в операторе case (строки 14 и 22) исключают элемент, имеющий индекс  $M$ , из текущей области, и поэтому уменьшают ее размер по крайней мере на 1, следовательно, программа должна завершиться.

#### 4.4. РЕАЛИЗАЦИЯ ПРОГРАММЫ

До сих пор мы работали с программой на псевдоязыке высокого уровня. Наше желание использовать подходящие структуры управляющей логики позволяло нам игнорировать особенности какого-либо конкретного языка программирования и сосредоточить внимание на сути проблемы. Однако в конечном счете нам придется писать программу на реально существующем языке. Именно для того, чтобы вы не думали, что я выбрал язык, облегчающий задачу, двоичный поиск был реализован на Бейсике. Хотя некоторые задачи решать с помощью программ, написанных на Бейсике, и хорошо, бедность его управляющих структур, а также глобальные (и обычно короткие) имена переменных являются существенным препятствием для создания реальных программ.

Но даже несмотря на эти проблемы приведенный выше псевдокод может быть легко выражен на диалекте языка Бейсик. В строках 1010 – 1045 содержится краткая спецификация подпрограммы.

1000 '

1010 ' ДВОИЧНЫЙ ПОИСК T В МАССИВЕ X(1..N)

1020 ' ВХОД: X(1..N) - ОТСОРТИРОВАН В НЕУБЫВАЮЩЕМ ПОРЯДКЕ

```

1030 '   ВЫХОД: P=Ø => T ОТСУТСТВУЕТ В X(1...N)
1040 '           P>Ø => P<N+1 и X(P)=T
1045 '   ПОБОЧНЫЕ ЭФФЕКТЫ: L, U и M ИЗМЕНЯЮТСЯ
1050 '
1060 L=1: U=N
1070 ' ОСНОВНОЙ ЦИКЛ
1080 'ИНВАРИАНТ: ЕСЛИ T ВОООБЩЕ ЕСТЬ В МАССИВЕ,
1090 '           ,           ТО ОН ДОЛЖЕН БЫТЬ В ПОДМАССИВЕ X(L..U)
1100 IF L>U THEN P=Ø: RETURN
1110 M=CINT((L+U)/2)
1120 IF X(M)<T THEN L=M+1: GOTO 1070
1130 IF X(M)>T THEN U=M-1: GOTO 1070
1140 '   ,   X(M)=T
1150           P=M: RETURN

```

Ввиду того, что я перевел эту программу на Бейсик с тщательно проверенного псевдокода, имеются достаточные основания считать, что она верна. Однако перед тем, как использовать ее в прикладных задачах (или опубликовать), все-таки надо ее проверить. Поэтому я написал простую программу из 25 строк на Бейсике, имеющую следующую структуру.

```

declare x[1...11]
for I := Ø to 11 do X[I] := I
for N := Ø to 10 do
  print N
  for I := 1 to N do
    Assert(BSearch(I)    = I)
    Assert(BSearch(I-.5) = Ø)
    Assert(BSearch(I+.5) = Ø)
  Assert(BSearch(Ø)      = Ø)
  Assert(BSearch(N+1)    = Ø)

```

Подпрограмма Assert ничего не делает, если ее аргумент имеет значение "истина", но "шумно выражает" недовольство, если он имеет значение "ложь". Первая же версия программы прошла этот тест без инцидентов. С помощью тестов была протестирована большая часть программы. Было проверено, успешно или нет выполняется поиск для каждой точки области, учитывая случай, когда элемент в массиве, но вне заданных для поиска границ. Проверки для  $N = 0, \dots, 10$  включали пустой массив, те размерности, для которых наиболее типичны ошибки, т. е. размерности 1, 2 и 3, ряд степеней числа 2, а также многие числа, не являющиеся степенью двух. Проверка была бы смертельно скучна (и поэтому могла содержать ошибки) при выполнении ее вручную, но при использовании компьютера она необременительна, так как на нее затрачивается незначительное время.

Мое мнение о том, что программа на Бейсике верна, сформировали многие факторы: я использовал разумные принципы для получения псевдокода, применил аналитическую методику для проверки его корректности, а потом дал компьютеру проявить себя в том, в чем он хорош, и засыпал программу текстовыми примерами.

#### 4.5. ОСНОВНЫЕ ПРИНЦИПЫ

Рассмотренное выше упражнение отразило многие основные моменты верификации программ: задача важна и требует тщательного написания программы; идеи верификации служат руководством при разработке программ, а при анализе правильности используются соображения общего характера. Основной недостаток этого упражнения – уровень детализации; на практике я работал бы на менее формальном уровне. К счастью, эти детали иллюстрируют ряд следующих общих принципов.

*Утверждения.* Взаимосвязь между входной информацией, переменными программы и ее выходом описывает "состояние" программы; утверждения позволяют программисту сформулировать эту взаимосвязь точно. Их роль в течение всей жизни программы рассматривается в следующем разделе.

*Последовательные структуры управляющей логики.* Простейшая структура, управляющая выполнением программы, имеет вид "выполнить сначала этот, а потом тот оператор". Чтобы разобраться в таких структурах, мы помещаем между ними утверждения и анализируем отдельно каждый шаг программы.

*Выбор структур управляющей логики.* В эти структуры в различных видах входят операторы *if* и *case*; во время выполнения программы выбирается одно из нескольких ветвлений. Для доказательства правильности такой структуры каждое из этих ветвлений рассматривается

отдельно. Факт выбора одного из условий перехода позволяет нам сделать некоторое утверждение в доказательстве; например, если мы выполняем оператор, следующий из условия  $I > J$ , то можем утверждать, что  $I > J$ , и использовать этот факт для вывода следующего связанного с этим утверждения.

**Итеративные управляющие структуры.** Доказательство корректности цикла состоит из трех шагов:

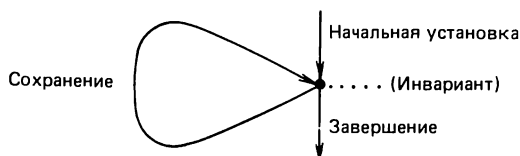


Рис. 4.1

Сначала мы убеждаемся, что инвариант цикла выполняется при его начальном определении, а потом показываем, что его истинность сохраняется на каждой итерации. На этих двух шагах с помощью математической индукции демонстрируется, что инвариант имеет значение "истина" перед и после каждого прохождения цикла. Третий шаг заключается в том, чтобы показать, что при любом завершении цикла требуемый результат верен. А все три шага служат для доказательства того, что если цикл когда-нибудь закончится, то это будет сделано корректно. То, что он закончится, нужно доказывать другими средствами (типичная аргументация использована при доказательстве конечности двоичного поиска).

**Подпрограммы.** Чтобы проверить подпрограмму, сначала точно определим ее назначение. Исходное состояние – это состояние, которое должно быть истинным перед вызовом подпрограммы, а *конечное состояние* – это состояние, которое подпрограмма обеспечивает при своем завершении. Примеры описания состояний приведены в программе двоичного поиска на Бейсике из разд. 4.4. Указанные условия – в большей степени соглашения, чем констатация фактов: ими констатируется, что если при вызове подпрограммы исходные условия удовлетворены, то выполнение подпрограммы установит итоговые условия. Доказав один раз, что подпрограмма обладает этим свойством, можно пользоваться установленной взаимосвязью между исходными и итоговыми условиями, не рассматривая каждый раз то, как выполняется подпрограмма.

#### 4.6. ЗАЧЕМ НУЖНА ВЕРИФИКАЦИЯ ПРОГРАММ

Когда один программист пытается убедить другого, что фрагмент программы верен, основным средством для этого является тестовый пример: выполнить программу вручную для некоторых входных дан-

ных. Это мощное средство, оно хорошо подходит для обнаружения ошибок, его легко использовать, и оно понятно. Ясно, однако, что программисты должны глубоко понимать программы, иначе им никогда этих программ не написать. Одно из основных достоинств верификации программ в том, что она дает программистам язык, на котором они могут выражать свое понимание программы.

Далее в этой книге, особенно в гл. 8, 10 и 12, мы будем использовать методику верификации при разработке сложных программ и применять язык верификации для объяснения каждой строки программы в момент ее написания. Это особенно полезно при составлении наброска инварианта для каждого из циклов. Важнейшие пояснения включаются в текст программы как утверждения; решение о том, какие именно утверждения включать, — это умение, которое приходит только с практикой.

Оператор контроля Assert, продемонстрированный при проверке программы двоичного поиска, позволяет проверять утверждения во время тестирования, как это сделано в решении задачи 10 из гл. 12. Если встречается ложное утверждение, то об этом выдается сообщение и работа завершается (во многих системах можно отключить проверку утверждений, если это слишком накладно по времени выполнения). Все программы, которые встретятся в этой книге, были испытаны подобно программе из разд. 4.4. Подробности этих проверок приведены в июльском номере журнала Communications of the ACM за 1985 г.

Язык верификации часто используется после написания первого варианта программы во время "прогулок" по ее тексту. Во время тестирования путь к локализации ошибок указывает нарушение истинности операторов Assert, а анализ вида нарушения показывает, как устранить ошибку и не внести при этом новые. Во время отладки обнаружьте ложные утверждения и не меняйте текст программы; всегда стремитесь разобраться в программе и сопротивляйтесь отвратительному побуждению исправлять программу до тех пор, пока она не заработает. Утверждения всегда имеют решающее значение при сопровождении программы; когда вы копаетесь в кодах, которые никогда раньше не видели и в которые никто не заглядывал годами, утверждения, описывающие состояние программы, могут оказать неоценимую помощь в ее понимании.

Ранее я упомянул, что эти методики составляют только небольшую часть задачи написания правильных программ; обычно ключом к правильности является простота текста программы. С другой стороны, некоторые профессиональные программисты, знакомые с этими методиками, рассказывали мне о своем практическом опыте, который очень близок к моей собственной практике программирования; в разработанной программе "тяжелые" части работают с первого раза, а ошибки обнаружива-

ются в "легких" частях. Приступая к "тяжелой" части программы, программисты мобилизуются и успешно используют мощные формальные методы. А в "легких" частях они возвращаются на старый путь, который приводит к старым же результатам. Я не верил в этот феномен, пока такое не случилось со мной; эти затруднения служат хорошим стимулом для использования описанных методов как можно чаще.

#### 4.7. ЗАДАЧИ

1. Как ни утомительно было доказательство правильности программы двоичного поиска, по некоторым стандартам оно все еще не завершено. Как бы вы доказали, что программа свободна от ошибок, которые проявляются при выполнении (такие как деление на 0, переполнение разрядной сетки, выход переменных за указанные для них диапазоны или выход индексов за границы массивов)? Зная основы дискретной математики, можете ли вы формализовать свое доказательство в виде логической системы?
2. Если исходный вариант программы двоичного поиска слишком прост для вас, попытайтесь реализовать вариант, который возвращает ячейке  $P$  адрес первого местонахождения элемента  $T$  в массиве  $X$  (этот элемент встречается в массиве несколько раз, и исходный алгоритм выберет один из них случайным образом). Число сравнений с элементами массива в вашей программе должно выражаться логарифмом от  $N$ , существует возможность сделать эту работу за  $\log_2 N$  таких сравнений.
3. Напишите и проверьте рекурсивную программу двоичного поиска. Какие фрагменты текста и доказательств останутся прежними, что и в итеративной версии, а какие части изменятся?
4. Добавьте фиктивные временные переменные к своей программе двоичного поиска, чтобы подсчитать количество выполняемых ею сравнений, и используйте методику верификации программы для доказательства того, что время ее работы в самом деле имеет логарифмическую зависимость от длины массива.
5. Докажите, что эта программа завершится, если на входе задать положительное число:

```
read X
while X  $\neq$  1 do
    if Even(X) then X := X/2 else X := 3*X+1
/* Функция Even имеет значение true, если
   X - четное; и false в противоположном случае */
```

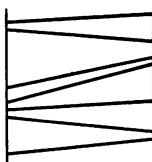
6. [К. Шолтен] Д. Гриз в своей книге "Наука программирования" (David Gries. Science of Programming) назвал эту задачу "Задачей кофейной банки". Вам дана банка из-под кофе, в которой содержится некоторое количество белых и черных бобов, а также дополнительно имеется большая куча черных бобов. Пока в банке не останется последний боб, вы повторяете следующие действия.

Случайным образом выбираете из банки два боба. Если они одинакового цвета, выбрасываете их и добавляете в банку черный боб; если разного, выбрасываете черный боб, а белый кладете обратно в банку.

Докажите, что этот процесс завершится. Что вы можете сказать о цвете последнего оставшегося боба как о функции числа белых и черных бобов, которые были в банке в начальный момент?

7. Коллега столкнулся со следующей проблемой в программе, предназначенной для черчения линий на экране дисплея с побитовым представлением изображения. Массив из  $N$  пар действительных чисел  $(a_i, b_i)$  определяет  $N$  линий  $y_i = a_i x_i + b_i$ . Эти линии упорядочены на интервале  $[0, 1]$  в том смысле, что  $y_i < y_{i+1}$  для всех значений  $i = 1, \dots, N-1$  и всех значений  $x$  на интервале  $[0, 1]$ :

Рис. 4.2



Менее формально эти линии не имеют общих точек между заданными вертикальными отрезками. Программист хотел определить для заданной точки  $(x, y)$ , где  $0 \leq x \leq 1$ , те две линии, между которыми лежит эта точка. Как быстро решить эту задачу?

8. Двоичный поиск существенно быстрее последовательного: при поиске в таблице из  $N$  элементов производится примерно  $\log_2 N$  сравнений, в то время как при последовательном поиске – примерно  $N/2$ . Часто такой скорости достаточно, но в некоторых случаях требуется сделать двоичный поиск еще более быстрым. И хотя в лучшем случае число сравнений все равно будет подчиняться логарифмической зависимости, можно ли переписать текст программы двоичного поиска, чтобы она стала работать быстрее? Для определенности предположим, что нужно осуществить поиск в упорядоченной таблице из  $N = 1000$  целых чисел.



9. В качестве упражнения по верификации программ определите поведение на входе и на выходе следующих фрагментов программ и покажите, что их тексты соответствуют их спецификациям. Первая программа выполняет сложение векторов  $A := B + C$ :

```
I := 1
while I <= N do
    A[I] := B[I] + C[I]
    I := I+1
```

В следующем фрагменте вычисляется максимальная величина в векторе X:

```
I := 2
Max := X[1]
while I <= N do
    if X[I] > Max then Max := X[I]
    I := I+1
```

Следующая программа последовательного поиска возвращает позицию первого найденного в векторе X [1 . . . N] элемента T. Оператор and в операторе while является условным, как и все другие операторы and и or, приведенные далее в этой книге: если первый оператор имеет значение false (ложь), то значение второго не вычисляется.

```
I := 1
while I <= N and X[I] ≠ T do
    I := I+1
if I > N then P := ∅ else P := I
```

В следующей программе за время, пропорциональное логарифму N, вычисляется N-я степень числа X. Текст этой рекурсивной программы "прямолинеен", проверка проста. Итеративный вариант программы более сложный и предлагается в качестве дополнительной задачи

```
function Exp(X,N)
    pre   N >= 0    /* На входе */
    post  result = X**N /* На выходе */
```

```

if N = 0 then
    return 1
else if Even(N) then
    /* Even - проверка четности */
    return Square(Exp(X,N/2))
    /* Square - возведение в квадрат */
else
    return X*Exp(X,N-1)

```

10. Введите ошибки в программу двоичного поиска и посмотрите, как их можно выловить при верификации и тестировании.

#### 4.8. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Идею разработки программы параллельно с доказательством ее правильности защищал Дейкстра в начале 70-х годов. Блестящим введением в эту область является книга "Наука программирования" Д. Гриза (David Gries. Science of Programming, опубликованная издательством Springer-Verlag в 1981 г.). Книга начинается с курса логики, переходит к формальному описанию разработки и верификации программ. В заключении книги обсуждается программирование на распространенных языках. В этой главе я попытался дать беглый очерк потенциальных выгод, которые дает верификация. Единственный путь к эффективному использованию верификации для большинства программистов – изучить книгу, подобную книге Гриза.

#### 4.9. ВЕРИФИКАЦИЯ ПРОГРАММ ПРИ ИХ "ПРОМЫШЛЕННОМ ПРОИЗВОДСТВЕ"

##### (ДОПОЛНЕНИЕ)

Описанная в этой главе методика верификации может оказать немедленное влияние на любого программиста: тщательно определите состояния на входе и на выходе каждого разрабатываемого вами модуля, а потом используйте неформальные средства для написания текста и "верификации" его правильности. Помните, что верификация – только одно из многих действий, гарантирующих, что вы поставили заказчику правильную, "крепкую" программу; важную роль в любой реальной системе играют тестирование и отладка. Улучшению качества ваших программ помогают книги, подобные книге Гриза.

Х. Миллз описывает в специальном выпуске журнала IBM Systems Journal (Volume 19, Number 4, 1980), посвященном разработке программного обеспечения, воздействие, которое оказали методики верификации программ на отдел федеральных систем фирмы IBM. Верификация составляет существенную часть курса, пройти который требуется всем программистам этого отдела. Основой этого курса является книга "Структурное программирование" Лингера, Миллза и Уитта (Linger, Mills, Witt. Structured Programming), опубликованная издательством Addison-Wesley в 1979 г. Миллз описывает, каким образом методики, основанные на верификации, играют важную роль в своевременной поставке этим отделом большого по объему программного продукта. Он описывает продукт, содержащий три миллиона слов команд и данных (на его разработку затрачено 200 чел.-лет), разработка которого началась своевременно и уложилась в бюджет. Более подробно об этом и других достижениях фирмы IBM читайте в этом выпуске журнала Systems Journal.

Хотя системы верификации программ в большинстве случаев еще не настолько готовы, чтобы их можно было использовать в производственных условиях, когда-нибудь, возможно, они станут самым обычным средством, помогающим разрабатывать некоторые виды программного обеспечения. Блестящие работы в этой области выполнены в ряде исследовательских центров; типичной для этих исследований является система Gypsy, разработанная в Университете шт. Техас (г. Остин) группой, руководимой Д. Гудом.

Система Gypsy – это методология составления спецификаций программ, их реализация и доказательства правильности. Ее основой является среда верификации Gypsy, которая предоставляет набор средств для применения этой методологии при построении программы. Программист составляет спецификацию и пишет текст, а система следит за различными аспектами программного обеспечения (спецификациями, текстом и доказательством) и помогает доказывать большинство теорем. Когда эта книга была подписана в печать, система Gypsy была использована для разработки двух значительных программ: "модулятора потока сообщений", который отфильтровывает из потока сообщений от одной машины к другой неверные сообщения (556 строк выполняемых операторов), и интерфейса для компьютерной сети (4211 строк операторов, выполняемых параллельно на двух компьютерах). Обе эти программы много раз тестировались, причем ошибки обнаружены не были.

Приведенные факты следует понимать в следующем контексте: только для коротких программ доказывалось, что они "абсолютно

верны". Верификация более длинной программы показывает, что программа обладает некоторыми свойствами (например, никогда не выдает неуместные сообщения). Эта программа может неверно работать в других случаях, но такое доказательство гарантирует, что определенные ошибки не могут быть сделаны. Еще один недостаток – это высокая стоимость: производительность одного программиста составляла только несколько команд в день (две – в коротких программах, четыре – в длинных). Дальнейшие исследования должны повысить производительность, но даже такая высокая стоимость приемлема для прикладных задач, где нужна высокая надежность или от которых зависит жизнь людей. Относительно перспектив верификации программ для таких прикладных задач я настроен оптимистически. Чтобы больше узнать о системе Gypsy, читайте статью Гуда (Good. Mechanical proofs about computer programs) в Phil. Trans. R. Soc. London A 312, p. 389 – 409 (1984).

## ЧАСТЬ II

### ЭФФЕКТИВНОСТЬ

Задачей-максимум для программиста (это и подчеркивалось в четырех предыдущих главах) является создание простой, но мощной программы, доставляющей удовольствие пользователям и не раздражающей своего создателя.

Теперь обратим наше внимание на другой специфический аспект совершенных программ – эффективность. Неэффективные программы огорчают своих пользователей долгим ожиданием результатов работы и большими расходами. Поэтому в этой части книги описаны некоторые способы повышения эффективности программ.

В следующей главе содержится обзор этих способов и описание того, как они влияют друг на друга. В трех последующих главах рассматриваются три метода уменьшения времени работы программы в том порядке, в котором они обычно применяются: в гл. 6 показано, как оценочные расчеты, выполненные на раннем этапе разработки, могут гарантировать достаточную эффективность основной структуры системы; в гл. 7 рассказывается о методах разработки алгоритмов, которые иногда разительно сокращают время работы модуля; в гл. 8 обсуждается оптимизация программы, которая обычно осуществляется на позднем этапе реализации системы. В завершение ч. II в гл. 9 рассматривается еще один аспект эффективности – эффективности по памяти.

Есть две существенные причины для исследования эффективности. Первая – важность, характерная для многих прикладных задач. По оценке одного моего знакомого руководителя разработки программного обеспечения, половина выделенного на разработку бюджета уходит на обеспечение эффективности; менеджер, занимающийся вводом в эксплуатацию средств обработки данных, должен был купить вычислительную машину стоимостью миллион долларов для решения проблем, связанных с эффективностью. Во многих системах предъявляются требования к скорости работы; сюда относятся программы реального времени, гигантские базы данных и машины, на которых работает одна-

единственная программа. Вторая причина исследования эффективности – чисто учебная. Не говоря уже о практической выгоде, эффективность является превосходной основой для обучения. Данные главы основываются на всевозможных идеях – от теории алгоритмов до обычного здравого смысла (как для оценочных вычислений). Основной темой является гибкость мышления, особенно это относится к гл. 5, где рассматривается задача с разных точек зрения.

Подобные уроки можно извлечь, обсуждая и многие другие темы. Эти главы можно было построить на изучении интерфейса с пользователем, "живучести" систем, надежности или точности ответов. Эффективность имеет то преимущество, что ее можно измерить: все могут согласиться с тем, что одна программа в 2.5 раза быстрее, чем другая, в то время как дискуссии, например, об интерфейсе с пользователем часто увязают в трясине спора о вкусах.

Глава 5 появилась в ноябрьском, гл. 6 – в мартовском, гл. 7 – в сентябрьском, гл. 8 – в февральском, а гл. 9 – в майском номерах журнала Communications of the ACM за 1984 г.

## Глава 5. ПРОИЗВОДИТЕЛЬНОСТЬ В ПЕРСПЕКТИВЕ

В следующих трех главах описываются три различных подхода к эффективности с точки зрения времени выполнения программ. В данной главе мы рассмотрим, как эти части объединились в единое целое: каждая методика применима на одном из нескольких *этапов разработки* компьютерной системы. Сначала мы исследуем одну частную программу, а потом обратимся к более систематическому рассмотрению этапов разработки.

### 5.1. РАЗБОР ПРИМЕРА

А. Аппел описал "эффективную программу моделирования для задачи множества тел" (Andrew Appel. An efficient program for many-body simulations) в январьском номере за 1985 г. журнала "SIAM Journal on Scientific and Statistical Computing" 6, 1, с. 85 – 103. Совершенствуя свою программу на различных уровнях, он сократил время ее работы с одного года до одного дня.

Эта программа решает "задачу  $N$  тел" – классическую задачу вычисления взаимодействий в гравитационном поле. Она моделирует движение  $N$  объектов в трехмерном пространстве, если даны их массы, начальные позиции и скорости; под объектами понимаются планеты, звезды или галактики. В двухмерном случае входные данные могли бы выглядеть так:

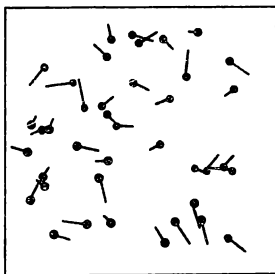


Рис. 5.1

В статье Аппела описываются две астрофизические задачи, в которых  $N = 10\,000$ ; изучая результаты моделирования, физики могут проверить, насколько хорошо теория соответствует астрономическим наблюдениям.

В очевидном варианте моделирующей программы время делится на маленькие "шаги" и вычисляется перемещение каждого объекта на очередном шаге. Так как для любого объекта вычисляется его притяжение каждым из всех других объектов, затраты на шаг приращения времени пропорциональны  $N^2$ . Аппел оценил, что такой алгоритм для  $N = 10\,000$  потребовал бы примерно один год на компьютере VAX-11/780 или один день на компьютере Gray-1.

В окончательном варианте этой программы, который уже был использован несколькими физиками, такая задача решается менее чем за один день на VAX-11/780 (время выполнения сокращено в 400 раз). В приведенном ниже кратком описании этой программы опущены многие важные детали, которые можно найти в статье Аппела; существенная особенность данной работы: громадное увеличение скорости было достигнуто в результате работы на различных этапах разработки.

*Алгоритмы и структуры данных.* Наивысший приоритет Аппел отдал сокращению затрат на шаг приращения времени с  $O(N^2)$  до  $O(N \log N)$ <sup>1</sup>. Поэтому он представил физические объекты как листья на двоичном дереве; более высокие узлы соответствуют группам объектов. Силу, действующую на отдельный объект, можно аппроксимировать силой, создаваемой большими группами объектов. Аппел показал, что такая аппроксимация не искажает результаты моделирования. Это дерево имеет примерно  $\log N$  уровней, и соответствующий алгоритм похож на

<sup>1</sup> Запись  $O(N^2)$  можно понимать, как "пропорционально  $N^2$ ". Например  $15N^2 + 100N$  и  $N^2/2 - 10$  — это  $O(N^2)$ . Неформально говоря,  $f(N) = O(g(N))$  означает, что  $f(N) < cg(N)$  для некоторой константы  $c$  и достаточно больших значений  $N$ . Формальное определение такой записи можно найти в большинстве учебников по разработке алгоритмов или дискретной математике, а в разд. 7.5 иллюстрируется уместность этой записи при разработке программ.

алгоритм из разд. 7.3. Указанное изменение сокращает время выполнения программы в 12 раз.

*Оптимизация алгоритма.* В простом алгоритме всегда используются шаги с маленьким приращением времени, чтобы обработать те редкие случаи, когда две частицы находятся близко друг от друга. Древовидная структура данных позволяет распознавать такие пары и обрабатывать их с помощью специальной процедуры. При этом удваивается размер шага приращения времени, поэтому вдвое сокращается время выполнения программы.

*Реорганизация структуры данных.* Дерево, соответствующее начальному набору объектов, становится совершенно вырожденным при представлении последующих наборов. Реконфигурация структуры данных на каждом шаге занимает мало времени, зато при этом уменьшается объем локальных вычислений, и поэтому вдвое сокращается время работы.

*Оптимизация программы.* Благодаря повышенной точности вычислений, которая обеспечивается древовидной структурой, 64-разрядные числа двойной точности с плавающей точкой оказалось возможным заменить 32-разрядными числами одинарной точности. Эта замена вдвое сократила время работы. Профилирование программы показало, что 98 % времени работы приходится на одну процедуру. Перепрограммирование ее на языке ассемблера увеличило бы скорость выполнения этой процедуры в 2.5 раза.

*Вычислительный комплекс.* После всех вышеприведенных изменений для выполнения программы все еще требовалось два дня работы на компьютере VAX-11/780, и желательно было пропустить ее несколько раз. Поэтому Аппел перенес программу на аналогичный компьютер, оснащенный устройством, ускоряющим выполнение операций с плавающей точкой, что вдвое сократило время работы.

Сокращения времени работы для всех описанных выше изменений перемножаются, давая суммарное ускорение в 400 раз. Окончательный вариант программы Аппела осуществляет моделирование для 10 000 тел примерно за один день. Однако достичь такого ускорения было нелегко. Простой алгоритм мог быть реализован программой из нескольких десятков строк, а для быстро работающей программы потребовалось 1200 строк на Паскале. На разработку и реализацию быстрой программы Аппел затратил несколько месяцев. Факторы ускоряющие работу программы, подытожены в следующей таблице:



Этап разработки	Множитель	Модификация
Алгоритмы и структуры данных	12	Двоичное дерево сокращает время с $O(N^2)$ до $O(N \log N)$
Оптимизация алгоритма	2	Использование больших шагов приращения времени
Реорганизация структуры данных	2	Создание групп объектов, хорошо согласующихся с "древовидным" алгоритмом
Оптимизация программы, не зависящая от системы	2	Замена чисел с плавающей точкой двойной точности на числа одинарной точности
Оптимизация программы, зависящая от системы	2.5	Перепрограммирование на ассемблере критической по времени выполнения процедуры
Вычислительный комплекс	2	Использование устройства, ускоряющего выполнение операций с плавающей точкой
Всего	400	

В этой таблице представлено несколько видов зависимости эффективности от факторов, ускоряющих работу программы. Первостепенным фактором является древовидная структура данных, которая открыла возможности для трех следующих изменений. Два последних фактора: (переход к программе на ассемблере и использование устройства, ускоряющего выполнение операций с плавающей точкой) в данном случае не зависели от "древовидной" структуры. Эта "древовидная" структура имела бы меньший эффект на компьютере Cray-1, конвейерная структура которого хорошо подходит для простого алгоритма. Таким образом, мы видим, что ускорение, достигаемое за счет алгоритма, может зависеть от типа вычислительного комплекса.

## 5.2. ЭТАПЫ РАЗРАБОТКИ

Компьютерная система проектируется на многих уровнях – от структуры программного обеспечения на верхнем уровне до транзисторов в устройствах компьютера на нижнем. Задача приведенного ниже обзора – дать только интуитивные описания этапов разработки, так что не рассчитывайте найти здесь их формальные определения и классификацию.

*Постановка задачи.* Борьба за быстродействующую систему может быть выиграна или проиграна при постановке задачи, которую она должна решать. В день, когда я писал этот абзац, продавец сказал мне, что не может осуществить поставку, так как заказ на покупку затерялся где-то между фирмой, где я работаю, и отделом поставок моей компании. Отдел поставок засыпан одинаковыми заказами; только в моей фирме разместили индивидуальные заказы 50 человек. Дружеская

беседа между администрацией фирмы и отделом поставок имела результатом объединение этих 50 заказов в один большой заказ. В дополнение к упрощению административной работы это изменение повысило скорость работы одной небольшой компьютерной системы в 50 раз. Хороший системный аналитик следит за такими улучшениями как во время ввода системы в эксплуатацию, так и в процессе ее работы.

Иногда хорошая спецификация дает пользователю несколько меньше, чем требовалось по их мнению. В гл. 1 мы видели на примере программы сортировки, как учет нескольких важных факторов, влияющих на входную информацию, сократил на порядок и время работы, и длину текста программы на порядок. Спецификация задачи может оказать трудноуловимое влияние на ее эффективность, например, хорошая диагностика ошибок в компиляторе может несколько замедлить его работу, но, как правило, сокращает общее время компиляции за счет уменьшения количества компиляций.

*Структура системы.* Вероятно, разбиение большой системы на модули – это единственный и самый главный фактор, определяющий ее производительность. Приведем две различные структуры системы, выдающей ответы на запросы:

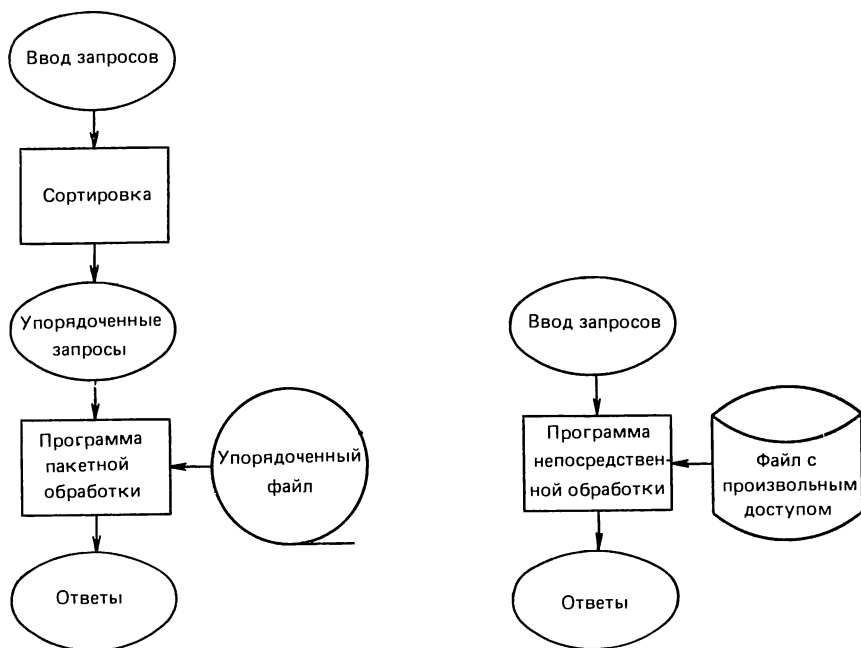


Рис. 5.2

Эти структуры представляют собой два канонизированных способа сбора взаимосвязанной информации. Они встречаются также в программах, обновляющих файлы, и в программах проверки правописания (к таким структурам мы вернемся в гл. 13). В задаче 1 показано, что каждая структура в некоторых случаях может быть обработана быстро, а в некоторых – медленно. После разбиения системы на модули разработчик должен сделать простую оценку, чтобы быть уверенным в достаточной производительности системы. Такие вычисления – тема гл. 6. Так как эффективность гораздо проще учесть при разработке новой системы, чем вводить задним числом в уже существующую, анализ производительности является решающим при проектировании системы.

*Алгоритмы и структуры данных.* Ключом к получению быстрого действия модуля обычно являются структуры его данных и алгоритмы, обрабатывающие эти данные. Самое существенное повышение быстрого действия программы Апела последовало после замены алгоритма  $O(N^2)$  алгоритмом  $O(N \log N)$ ; в гл. 2 и 7 описаны примеры аналогичного увеличения быстрого действия.

*Оптимизация программы.* Апел достиг ускорения в 5 раз, внося небольшие изменения в текст программы; этой теме посвящена гл. 8.

*Системное программное обеспечение.* Иногда проще заменить программные средства, с помощью которых реализована система, чем саму систему. Например, в разд. 6.4 описано, как замена интерпретирующего языка компилирующим увеличила эффективность в несколько сотен раз. К другим доступным средствам ускорения относятся оптимизация компилятора (включая исполнительную систему языка программирования), а также модификация операционной системы и базы данных.

*Технические средства.* Имеется много способов, с помощью которых более быстродействующий комплекс технических средств может увеличить эффективность. Иногда компьютер общего назначения обеспечивает нужную скорость; ускорения можно достичь с помощью более быстродействующей версии компьютера такой же архитектуры, мультипроцессорных систем или суперкомпьютеров. Иногда, с точки зрения отношения стоимость-эффективность, лучше создать специальное устройство для решения частной задачи; например, специальные микросхемы для синтеза речи дают возможность разговаривать недорогими игрушками и бытовым приборам. Решение Апела использовать устройство, ускоряющее выполнение операций с плавающей точкой, находится между этими двумя крайностями.

### 5.3. ОСНОВНЫЕ ПРИНЦИПЫ

Поскольку профилактика лучше лечения, мы должны запомнить наблюдение, сделанное Г. Беллом из фирмы Encore Computer Corporation:

*Самые дешевые, быстрые и надежные компоненты компьютерной системы – это те, которых нет.*

Эти отсутствующие компоненты являются также самыми точными (они никогда не допускают ошибок), наиболее защищенными (в них нельзя вторгнуться) и самыми простыми в разработке, документировании, тестировании и сопровождении. Важность простоты разработки нельзя переоценить.

Если от проблем эффективности уклониться невозможно, то помочь сконцентрировать усилия программиста могут размышления об этапах разработки.

*Если вам требуется незначительное ускорение, оптимизируйте этап, дающий наибольший эффект.* Большинство программистов имеет собственные, вошедшие в плоть и кровь представления об эффективности: чуть что, с языка срывается: "изменить алгоритм" или "оптимизировать дисциплину обслуживания очередей". Перед тем, как вы решите заняться увеличением эффективности на любом конкретном этапе, рассмотрите все этапы и выберите тот, который дает наибольшее ускорение при наименьших усилиях.

*Если вам требуется большое ускорение, оптимизируйте несколько этапов.* Огромное увеличение скорости, как в случае Аппела, достигается только при наступлении на задачу на нескольких фронтах и требует обычно очень больших усилий. Если изменения на одном уровне не зависят от изменений на других уровнях (что бывает часто, но не всегда), то достигаемые на каждом уровне увеличения скорости перемножаются.

В гл. 6 – 8 увеличение быстродействия достигается на трех различных этапах разработки; при рассмотрении каждого способа ускорения обращайте внимание на их место среди других способов.

#### 5.4. ЗАДАЧИ

1. На рис. 5.1 в разд. 5.2 показаны два возможных варианта построения простой системы, выдающей ответы на запросы. Предположим, что каждая из 1 млн записей в файле идентифицируется ключом и что при каждом запросе происходит обращение (с помощью ключа) к единственной записи. Предположим далее, что оба файла хранятся на диске в виде блоков по 100 записей в каждом и что произвольный блок может быть прочитан с диска за 50 мс. Однако блоки можно читать и последовательно, затрачивая на каждое чтение только 5 мс. Программа пакетной обработки читает весь файл, а алгоритм опера-

тивного доступа читает только относящиеся к запросам блоки, но зато читает некоторые блоки многократно. В каком случае метод пакетной обработки более эффективен, чем метод оперативного доступа?

2. Рассмотрите, какое увеличение скорости можно достичь на различных этапах разработки для некоторых нижеприведенных задач: игра "Жизнь" Конуэя, разложение на множители целых чисел, состоящих из 100 десятичных цифр, анализ Фурье, моделирование сверхбольших интегральных схем и поиск в большом текстовом файле на диске заданной последовательности символов. Исследуйте взаимозависимость предложенных методов ускорения.
3. Аппел обнаружил, что переход от двойной точности к одинарной удвоил скорость выполнения его программы. Подберите подходящий пример для проверки и измерьте аналогичное возрастание скорости в вашей системе.
4. В этой главе внимание сосредоточено на эффективности по времени выполнения. Другие распространенные показатели эффективности включают устойчивость к отказам, надежность, защищенность, стоимость, отношение стоимость-эффективность, точность и устойчивость по отношению к ошибкам пользователя. Как на различных этапах разработки подступиться к каждой из этих проблем?
5. Обсудите стоимость использования технологий различного уровня сложности на различных этапах проектирования. Включите все относящиеся сюда аспекты стоимости, в том числе время разработки (календарное и чел.-ч), удобство сопровождения и цену в долларах.
6. В одной старой популярной поговорке утверждается, что "эффективность вторична по отношению к правильности, т. е. скорость работы программы не важна, если ее ответы неправильны". Верно это или нет?
7. Рассмотрите различные решения обыденных проблем, таких как оценка ущерба от автомобильных аварий.

## 5.5. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Я изучил тему этой главы по статье Р. Редди и А. Ньюэлла "Мультипликативное возрастание скорости систем" (Raj Reddy, Allen Newel. Multiplicative speedup of systems, в книге Perspectives on Computer Science под редакцией А. К. Джонса, опубликованной издательством Academic Press в 1977 г.). Их побудили написать эту статью задачи искусственного интеллекта. Они считают, что работа систем, обрабатывающих речь и изображения, может быть ускорена в миллион раз. В их статье описывается увеличение скорости работы, достигаемое на различных этапах проектирования, статья особенно богата примерами ускорения, обусловленного техническими средствами и системным программным обеспечением.

Статья Б. Лампсона "Советы по разработке компьютерных систем" (Butler Lampson. Hints for Computer Systems Design) появилась в январском номере журнала IEEE Software 1, 1 за 1984 г. Многие из этих советов касаются эффективности; особая важность этой статьи заключается в интегрированном подходе (оборудование-программное обеспечение) к разработке систем.

## Глава 6. ПРЕДВАРИТЕЛЬНЫЕ ОЦЕНКИ

Во время непринужденной беседы о разработке программного обеспечения Б. Мартин спросил меня: "Сколько воды вытекает из Миссисипи за день?" Так как до этого момента, судя по его замечаниям, мне казалось, что он глубоко вник в суть предмета, я вежливо сдержал свою реакцию и сказал: "Извините?" Когда он повторил вопрос снова, я понял, что мне придется ублажать бедного парня, который, очевидно, свихнулся от переутомления, руководя большим отделом программного обеспечения в фирме Bell Labs.

Мой ответ был примерно таким. Я прикинул, что вблизи устья река имеет ширину около мили и глубину, вероятно, 20 футов (или примерно 1/250 мили). Я предположил, что скорость потока 5 миль/час или 120 миль/день. Формула

$$1 \text{ миля} \times 1/250 \text{ мили} \times 120 \text{ миль/день} \approx 1/2 \text{ миль}^3/\text{день}$$

показывает, что река изливает примерно половину кубической мили воды в день с точностью до порядка. Ну и что?

В этот момент Мартин взял со своего стола предложения по компьютеризованной почтовой системе, которую компания AT&T разработала для летних Олимпийских игр 1984 г., и проделал похожую последовательность вычислений. Хотя его цифры были более точными, так как брались прямо из этого предложения, вычисления были слишком просты и выглядели гораздо обескураживающе. Они показывали, что при благоприятных предположениях предлагаемая система сможет работать только в случае, если 1 мин будет равна не менее 120 с. Днем ранее он отослал эти предложения обратно, на доработку. (Эта беседа имела место в начале 1983 г., окончательный вариант системы уже работал во время Олимпиады без сбоев.)

Таким эксцентричным способом Б. Мартин использовал инженерную методику "предварительных оценок", которую очень любят в школах, готовящих инженеров, где она является хлебом насущным для боль-

шинства инженеров-практиков. К несчастью, ею очень часто пренебрегают в сфере применения компьютеров.

## 6.1. ОСНОВНЫЕ НАВЫКИ

Эти основные напоминания могут оказаться довольно полезными при выполнении предварительных вычислений.

*Два ответа лучше, чем один.* Когда я спросил П. Уэйнбергера, сколько воды вытекает из Миссисипи за один день, он ответил: "Столько, сколько втекает". После этого он прикинул, что бассейн Миссисипи примерно 1000 на 1000 миль, и что количество осадков составляет за год около 1 фута (или 1/5000 мили). Это дает

$$1000 \text{ миль} \times 1000 \text{ миль} \times 1/5000 \text{ мили/год} \approx 200 \text{ миль}^3 / \text{год}$$

$$200 \text{ миль}^3 / \text{год} / 400 \text{ дней в году} \approx 1/2 \text{ мили}^3 / \text{день}$$

т. е. несколько больше, чем половина кубической мили в день. Важное значение имеет двойная проверка всех вычислений, особенно таких быстрых.

*Третья контрольная проверка.* В календаре сообщается, что из реки вытекает 640 000 кубических футов за 1с. Исходя из этого, получим

$$640\,000 \text{ фут}^3/\text{с} \times 3600 \text{ с/ч} \approx 2.3 \times 10^9 \text{ фут}^3/\text{ч}$$

$$2.3 \times 10^9 \text{ фут}^3/\text{ч} \times 24 \text{ ч/день} \approx 6 \times 10^{10} \text{ фут}^3/\text{день}$$

$$6 \times 10^{10} \text{ фут}^3/\text{день} / (5000 \text{ фут/миля})^3 \approx$$

$$\approx 6 \times 10^{10} \text{ фут}^3/\text{день} / (125 \times 10^9 \text{ фут}^3/\text{миля}^3) \approx$$

$$\approx 60/125 \text{ мили}^3/\text{день} \approx 1/2 \text{ мили}^3/\text{день}.$$

Близость двух оценок друг к другу и особенно к ответу, полученному из календаря, является хорошим примером случайного везения.

*Быстрая проверка.* Пойа посвятил три страницы своей книги "Как решать задачу" проверке размерностей, которую он описывает как "хорошо известное, быстрое и эффективное средство проверки геометрических и физических формул". Первое правило: размерности слагаемых должны быть одинаковыми, размерность суммы будет такой же. Вы можете складывать футы, чтобы получить футы, но вы не можете прибавлять к футам секунды. Второе правило: размерность произведения равна произведению размерностей. Приведенные выше примеры подчиняются обоим правилам; выражение

$$(\text{миля} + \text{миля}) \times \text{миля} \times \text{миля/день} = \text{миля}^3/\text{день}$$

имеет правильный вид (без учета коэффициентов).

В сложных выражениях, подобных приведенным выше, вам поможет следить за размерностями простая таблица. Чтобы выполнить расчет Уэйнбергера, мы вначале выпишем три исходные величины:

Рис. 6.0

1000 миль	1000 миль	1 миля
		5000 лет

После этого мы упрощаем выражение, сокращая члены, что дает на выходе  $200 \text{ миль}^3/\text{год}$ .

Рис. 6.1

<del>1000 миль</del>	<del>1000 миль</del>	<del>1 миля</del>	200 миль <sup>3</sup>
		<del>5000 лет</del>	

Теперь мы делим на 400 – почти точное число дней в году.

Рис. 6.2

<del>1000 миль</del>	<del>1000 миль</del>	<del>1 миля</del>	200 миль <sup>3</sup>	лет
		<del>5000 лет</del>		400 дней

Сокращая, получаем (теперь уже знакомый) ответ –  $1/2 \text{ миль}^3/\text{день}$ .

<del>1000 миль</del>	<del>1000 миль</del>	<del>1 миля</del>	200 миль <sup>3</sup>	<del>лет</del>	1
		<del>5000 лет</del>		<del>400 дней</del>	2

Рис. 6.3

Такие табличные вычисления помогут вам следить за размерностями.

Контролируя размерности, мы проверяем вид выражения. Правильность умножения и деления проверяйте с помощью старого приема времен логарифмической линейки. Отдельно друг от друга вычисляйте значащие цифры и порядки. Имеется несколько быстрых способов проверки сложения:

$$\begin{array}{r} 3142 \\ 2718 \\ +1123 \\ \hline 983 \end{array}$$

$$\begin{array}{r} 3142 \\ 2718 \\ +1123 \\ \hline 6982 \end{array}$$

$$\begin{array}{r} 3142 \\ 2718 \\ +1123 \\ \hline 6973 \end{array}$$



В первой сумме слишком мало цифр, а вторая содержит ошибку в последней значащей цифре. Способом "отбрасывания девяток" обнаруживается ошибка в третьем примере: сумма цифр слагаемых равна 8 по модулю 9, а сумма цифр в ответе равна 7 по модулю 9 (в случае правильного суммирования суммы цифр должны быть равны после отбрасывания групп цифр, сумма которых равна 9).

Кроме того, не забывайте о здравом смысле: относитесь с подозрением к любым вычислениям, которые показывают, что из Миссисипи вытекает 100 галлонов воды в день.

## 6.2. БЫСТРЫЕ ВЫЧИСЛЕНИЯ ПРИ РАЗРАБОТКЕ КОМПЬЮТЕРНЫХ СИСТЕМ

Кард, Моран и Ньюэлл на страницах 9 и 10 своей книги "Психология человекомашинного взаимодействия" (Card, Moran, Newell. Psychology of Human-Computer Interaction, публикация Erlbaum, 1983) нарисовали следующую претенциозную картину получения оценок.

Разработчик системы – руководитель маленькой группы. Составляя спецификации для настольной системы календарного планирования, он делает выбор: заставить пользователя вводить каждую команду с клавиатуры или отмечать ее в меню с помощью светового пера. На чистом листе он перечисляет некоторые типичные задания, которые пользователь должен выполнить в этой системе. Шаги, выполняемые при вводе команд с клавиатуры, и задание опций меню записывает в два столбца. Из справочника он выбирает время для каждого шага и суммирует эти времена, чтобы получить общее время для задания. В системе, где команды вводятся с клавиатуры, требуется меньше времени, но ненамного. Выполняя анализ с помощью другого раздела справочника, разработчик вычисляет, что систему с меню можно быстрее освоить; фактически она может быть освоена за время, в 2 раза меньшее. Ранее он оценил, что для эффективной системы с меню потребуется более дорогой процессор, где на 20 % больше объем ОЗУ, на 100 % больше памяти для микропрограмм, и более дорогой видеотерминал. Являются ли эти дополнительные затраты оправданными? Еще несколько минут вычислений, и он понял поразительный факт, что при ожидаемых сейчас объемах производства стоимость обучения для системы с вводом команд с клавиатуры превысит стоимость производства одного экземпляра системы! Увеличение стоимости вычислительного комплекса было бы с избытком уравновешено сокращением стоимости обучения, даже если не рассматривать ожидаемое увеличение рынка сбыта для системы, которая проще в освоении. Есть ли у системы с вводом команд с клавиатуры преимущества по другим параметрам? Он продолжил анализ, рассматривая нагрузку

на память пользователя, возможность ошибок пользователя и вероятность утомления. А в соседней комнате праздно гудел неиспользуемый компилятор Паскаля, ожидая решения руководителя.

Далее в своей книге авторы переходят к разработке научной базы в области психологии, которая является необходимым предшественником описанного справочника.

Этот сценарий показывает, как простые оценки могут позволить разработчику системы сделать рациональный выбор между несколькими привлекательными альтернативами. Это совершенно другое их применение, отличное от вычислений Мартина для олимпийской почтовой системы: его анализ единственного варианта обнаружил ее фатальный изъян (подобные вычисления в разд. 2.4 показали бессмысленность простого алгоритма поиска анаграмм). В обоих случаях короткая последовательность вычислений давала ответ, достаточный для рассматриваемого вопроса, дополнительные вычисления мало бы что добавили.

На раннем этапе разработки системы быстрые расчеты могут вывести разработчика из опасных вод в надежную гавань, а если вы не воспользовались ими вовремя, показать, что проект был обречен на неудачу. Эти вычисления как правило тривиальны, в них используется математика на уровне средней школы. Труднее запомнить, что прибегать к ним надо достаточно часто.

### 6.3. "ЗАПАС ПРОЧНОСТИ"

Результат любого расчета точен настолько, насколько точны его входные данные. С точными данными простые вычисления могут дать точные ответы, которые иногда бывают довольно полезными. В 1969 г. Д. Кнут написал текст программ для сортировки на диске только для того, чтобы обнаружить, что для работы этого пакета требуется в 2 раза больше времени, чем предсказывали расчеты. Тщательная проверка вскрыла упущение: из-за ошибки в программном обеспечении диск, эксплуатируемый один год, работал в течение всей своей "жизни" со скоростью, составляющей только половину от объявленной для него фирмой-производителем. Когда устранили эту ошибку, пакет для сортировки стал вести себя, как было предсказано, и все другие связанные с диском программы также стали работать быстрее. Однако часто для попадания в цель достаточно и "небрежных" входных данных. Если вы гадаете о 20 % здесь и 50 % там и тем не менее обнаружили, что система в 100 раз быстрее или медленнее технических требований, дополнительная точность не нужна. Но прежде чем допустить ошибку в пределах 20 %, обдумайте следующий совет В. Висоцки:

”Большинство из вас, вероятно, могут воскресить в памяти вид ”Галлопирующей Герти” – моста, построенного Такомой Нэрроу, который разломался во время бури в 1940 г.<sup>1</sup> Ну ладно, подвесные мосты разрывались сами по себе таким образом в течение восьмидесяти лет или около того до ”Галлопирующей Герти”. Это было проявлением аэродинамической подъемной силы, а чтобы выполнить соответствующий инженерный расчет таких сил, содержащих существенные нелинейности, вы должны использовать для моделирования спектров вихрей математические методы и концепции Колмогорова. До середины 50-х годов никто не знал в детелях, как это сделать правильно. Так, а почему Бруклинский мост не развалился подобно ”Галлопирующей Герти”?

Потому, что Джон Рoubлинг имел достаточно здравого смысла, чтобы знать то, чего он не мог знать. Еще сохранились его заметки и письма, относящиеся к Бруклинскому мосту, и они являются замечательными примерами хорошего инженерного понимания пределов своего знания. Он знал об аэродинамической подъемной силе, действующей на подвесные мосты, он наблюдал ее. И знал также, что его знаний недостаточно для того, чтобы смоделировать эту силу. Поэтому он заложил в проект прочность ферм проезжей части Бруклинского моста в 6 раз больше, чем потребовалось бы по расчетам для известных статических и динамических нагрузок. И чтобы сделать более жесткой всю структуру моста, точно определил сеть диагональных растяжек, сбегających вниз к проезжей части. Взгляните на этот мост, он почти уникален.

Когда Рoubлинга спрашивали, не разрушится ли предложенный им мост подобно многим другим, он говорил: ”Нет, потому что я сконструировал его в 6 раз прочнее, чем требуется для того, чтобы предотвратить происшествия”.

”Рoubлинг был хорошим инженером и построил хороший мост, используя огромный запас прочности, чтобы компенсировать свое незнание. Поступаем ли мы так? Я утверждаю, что при расчете эффективности наших программных систем реального времени мы должны считать, что параметры хуже в 2, 4, 6 раз, чтобы компенсировать наше незнание. Закладывая параметры надежности-готовности, мы должны в 10 раз усилить требования, которые, как нам кажется, мы можем удовлетворить, чтобы компенсировать наше незнание. Оценивая размер, стоимость и график работ, мы должны быть осторожны и ввести коэффициенты 2 или 4, чтобы скомпенсировать наше незнание. Мы должны вести разработку таким образом, как это делал Джон Рoubлинг, а не таким,

---

<sup>1</sup> Дополнительную информацию об этом случае можно почерпнуть в разд. 2.6.1 книги Брауна ”Дифференциальные уравнения и их приложения” (Braun. Differential Equations and Their Applications), второе издание опубликовано издательством Springer-Verlag в 1978 г.

как его современники – насколько я знаю, ни одного из подвесных мостов, построенных современниками Роублинга, не осталось в Соединенных Штатах, а четверть всех мостов любого типа, построенных в США в 1870-х годах, разрушилась в течение десяти лет после их создания. "Такие ли мы инженеры, как Джон Роублинг? Я сомневаюсь."

#### 6.4. РАЗБОР ПРИМЕРА

Чтобы конкретизировать рассмотренные выше соображения, я опишу, как (или почти как) я пользовался ими при разработке системы для одной компании 1982 г. Подробности можно найти в техническом отчете университета Карнеги-Меллон (Carnegie-Mellon University Computer Science Technical Report CMU-CS-83-108). Здесь я приведу только краткое описание.

Эта система подготавливала несколько отчетов в течение дня, чтобы обобщить данные, представляемые в виде 1000 записей по 80 столбцов каждая. Каждый из отчетов был длиной около 80 страниц. Предыдущий вариант такой системы работал на большой ЭВМ. В мою задачу входила реализация системы на персональном компьютере с использованием интерпретатора языка Бейсик.

На начальном этапе разработки системы я выполнил простые расчеты, чтобы удостовериться в том, что персональный компьютер пригоден для этой прикладной задачи. Анализ пригодности по памяти был прост: я вычислил размер нескольких самых больших таблиц и обнаружил, что для них требуется половина из 48 Кбайт памяти компьютера. Анализ пригодности по времени был сосредоточен на двух основных фазах.



Рис. 6.4

Я не особенно беспокоился за время фазы 1: предыдущая система на ЭВМ IBM Systems/360 (модель 25) выполняла ее за 1 мин, а микропроцессор в персональной ЭВМ был более мощным, чем мой старый ящик. Вместо этого я уделил основное внимание фазе 2, скорость выполнения которой, как я думал, будет ограничена устройством печати, выдающим 60 строк/мин. Каждая страница отчета содержала около 30 строк, так что общее время в 40 мин хорошо укладывалось в ограничения. После этого короткого анализа кампания купила три персональных компьютера, а я осуществил разработку.

Первый вариант программы открыл много нового. Для хранения программы на Бейсике потребовалось около 20 Кбайт ОЗУ, чем я пренебрег в своих расчетах; положение спас коэффициент надежности, равный двум. Оценка времени печати, равная 40 мин, точно "попала в цель". К сожалению, я сильно ошибся в оценке времени чтения записей и построения таблиц. Вместо 1 мин это заняло 14 ч (!), поэтому системе было чрезвычайно тяжело подготавливать несколько отчетов в день. Проблема состояла в том, что я сравнил программу на ассемблере в старой System/360 с интерпретатором Бейсика на персональном компьютере, проигнорировав тот факт, что интерпретатор Бейсика выполняет программу в несколько сотен раз медленней, чем работает программа на ассемблере.

Теперь я сделал более тщательные оценки. Использование описанных выше параметров (1000 записей по 80 столбцов каждая) и некоторых предположений о значениях других параметров (50 операторов Бейсика на столбец, 100 операторов Бейсика в 1 с) дало следующий результат:

1000 записей	80 столбцов	50 операторов	1 с
	запись	столбец	100 операторов

Рис. 6.5

После умножения и сокращения мы видим, что

<del>1000 записей</del>	<del>80 столбцов</del>	<sup>50</sup> <del>операторов</del>	<sup>1 с</sup> <del>100 операторов</del>	40000
	запись	столбец		

Рис. 6.6

т. е. требуется 40 000 с. Поделив на 3600 с/ч, получим оценку – примерно 11 ч. Другой вариант оценки: я знал, что старая ЭВМ затрачивала на задачу 1 мин и выполняла одну команду за 10 мкс. Увеличение времени выполнения до 10 мс дает коэффициент замедления, равный 1000; который, умноженный на предыдущее значение (1 мин), составляет примерно 17 ч.

Если бы я знал цену этого подхода перед созданием программы, то использовал бы более быстрый язык. Вместо этого у меня была готовая программа из 600 строк, и не было иного выхода, кроме оптимизации команд программы с помощью методов гл. 8. На 70 строк программы на фазе 1 приходилось более 90 % времени работы: на выполнение только трех из них уходило 11 ч (менее 1 % программы выполнялось за 75 %

времени). Я потратил 40 ч на замену 70 строк на Бейсике 110 строками на Бейсике и 30 строками на ассемблере, что сократило время выполнения фазы 1 с 14 ч до 2 ч 20 мин. Это было достаточно хорошо для данной конкретной системы, но дольше, чем могло быть, если бы я выполнил быстрые расчеты заблаговременно и выбрал бы для реализации более эффективный язык.

## 6.5. ОСНОВНЫЕ ПРИНЦИПЫ

Используя оценки, непременно вспоминайте известный совет Эйнштейна.

*Все должно быть сделано настолько просто, насколько это возможно, но не проще.*

Мы знаем, что простые вычисления перестают быть чересчур простыми, когда мы вводим "запас прочности", чтобы компенсировать наши ошибки в оценке параметров и наше незнание рассматриваемой задачи.

## 6.6. ЗАДАЧИ

1. При каких расстояниях курьер на велосипеде, везущий магнитную ленту, будет более быстрым переносчиком информации, чем телефонная линия со скоростью передачи 56 000 бит/с?
2. Сколько времени вам потребовалось бы для заполнения диска, если бы вы вводили символы с клавиатуры?
3. В каких случаях эффективно с точки зрения затрат устанавливать терминал у программистов дома?
4. Предположим, что все процессы в мире замедлились в 1 млн раз. Сколько времени потребуется вашему компьютеру для выполнения команды? Диску, чтобы совершить один оборот? Устройству перемещения магнитных головок для поиска по всему диску? Вам, чтобы ввести с клавиатуры свое имя?
5. В каком случае будет выполнен наибольший объем работы: за 1 с на суперкомпьютере, за 1 мин на мидикомпьютере, за 1 ч на микрокомпьютере или за 1 день с использованием языка Бейсик на персональном компьютере?
6. Предположим, что система выполняет 100 обращений к диску при обработке запроса (хотя некоторым системам необходимо меньшее количество, в других требуется несколько сотен обращений к диску для обработки запроса). Сколько запросов в час может выполнить система?
7. Чтобы ускорить выполнение программы на 10 %, программист затратил календарный день и 1 ч процессорного времени на ЭВМ со стои-

мостью процессорного времени 100 дол./ч. Когда программа работает на ЭВМ одна, ей требуется обычно 1 мин процессорного времени. За сколько времени окупится это ускорение, если программа работает 100 раз в день? Если коэффициент ускорения равен 2 или 10?

8. [Р. Пайк] Во многих трансляторах имеются оптимизаторы, которые формируют более эффективную программу. Если в имеющемся у вас компиляторе есть такая возможность, измерьте, насколько возрастает время компиляции и насколько более быстродействующим становится полученный объектный модуль. Когда стоит запускать оптимизатор? В каких случаях стоит его разрабатывать?
9. Меня однажды попросили написать программу для передачи данных из одного персонального компьютера на другой, архитектура которого очень сильно отличалась от первого. Так как файл состоял только из 400 записей по 20 цифр в каждой, я посоветовал повторно ввести данные с распечатки, которую легко получить. Оцените затраты, связанные с каждым подходом, включая время программиста, необходимость вложить деньги в оборудование и стоимость передачи.
10. В статье июльского номера журнала Communications of the ACM за 1984 г. (с. 652) утверждается, что "система обрабатывает в среднем 7 328 764 запросов в день". Комментарии есть?
11. С помощью быстрых расчетов оцените время выполнения для разработок, описанных в этой книге:

1) сделайте оценку разработок для задач 1.7, 2.7 и 5.1 и в разделах 2.2, 2.4, 5.2, 13.1 и 13.3;

2) арифметику с "О-большим" можно рассматривать как формализацию для быстрых вычислений – она отслеживает скорость возрастания, но игнорирует множители-константы. Используйте вычисленное с точностью до "О-большого" время работы алгоритмов в гл. 5, 7, 10 – 12, чтобы оценить время работы реализованных на их основе программ. Сравните ваши оценки с экспериментами, о которых рассказано в этих главах.

## 6.7. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Раздел Д. Хофштадтера "Темы метамагии" (Douglas Hofstadter. *Magical Themas*) в майском номере Scientific American за 1982 г. имеет подзаголовок: "Безмолвие чисел или почему неумение считать так же опасно, как и неумение читать?" Она перепечатана в виде постскрипума в его книге "Темы метамагии" (*Magical Themes*), опубликованной издательством Basic Books в 1985 г. Это хорошее введение в методы оценок и выразительное изложение их важности.

Физики хорошо знают этот предмет. После того, как данная глава появилась в журнале Communications of the ACM, Я. Волитзки написал:

Я часто слышал, что метод оценок называют "аппроксимацией Ферми" по имени этого физика. Предание гласит, что Энрико Ферми, Роберт Опенгеймер и другие руководители Манхаттанского проекта в ожидании взрыва первого ядерного устройства находились за невысокой защитной стеной на расстоянии в несколько тысяч ярдов от него. Ферми порвал несколько листов бумаги на маленькие кусочки и подбросил их в воздух, когда увидел вспышку. После того, как прошла ударная волна, он измерил шагами расстояние, на которое переместились клочки бумаги, быстро выполнил приближенные вычисления и получил мощность взрыва бомбы, которая была подтверждена значительно позже дорогостоящим контрольно-измерительным оборудованием.

Эдвард Перселл редактирует ежемесячный раздел в журнале American Journal of Physics, озаглавленный "Оценочные вычисления". В нем много таких очаровательных вопросов, как "Сколько баррелей нефти потребляет 60-Вт лампа за год?" или "Сколько времени потребовалось бы для передачи по видеоканалу генома человека (приблизительно 1 м молекулы ДНК)?"

#### 6.8. БЫСТРЫЕ ВЫЧИСЛЕНИЯ В ПОВСЕДНЕВНОЙ ЖИЗНИ (ДОПОЛНЕНИЕ)

Публикация этой главы в журнале Communications of the ACM вызвала много интересных писем. Один читатель сообщил об услышанном рекламном заявлении, что один коммивояжер проехал на новом автомобиле 10 000 миль за год. Читатель попросил своего сына проверить истинность этого утверждения. Вот быстрый ответ: в году 2000 рабочих часов (50 недель по 40 ч в неделю), коммивояжер мог проезжать в среднем 50 миль/ч, что не учитывает времени, затраченного на продажу, но при умножении дает результат, совпадающий с утверждением. Поэтому такому заявлению можно верить, но с натяжкой.

Каждодневная жизнь предоставляет нам много возможностей для оттачивания наших навыков быстрых вычислений. Например, сколько денег вы истратили в прошлом году на еду в ресторанах? Я был однажды шокирован, услышав, как один житель Нью-Йорка быстро подсчитал, что он и его жена тратят каждый месяц на такси денег больше, чем платят за квартиру. Вопрос к читателям из Калифорнии (которые, возможно, не знают, что такое такси), сколько времени займет заполнение плавательного бассейна водой при помощи шланга для поливки сада?



Некоторые читатели отметили, что быстрым вычислениям желательнее выучиться в раннем возрасте. Р Пинкем из Стивенсоновского технологического института написал:

Я учитель и пытался в течение ряда лет обучать оценочным вычислениям всех желающих. Просто удивительно, до чего безуспешно. Такое впечатление, что здесь требуется скептический склад ума.

Мой отец вдолбил это в меня. Я родом с побережья шт. Мэн. Маленьким ребенком я услышал беседу моего отца и его друга Поттера. Он утверждал, что две леди из шт. Коннектикут наловили за день раков общим весом в 200 фунтов. Мой отец сказал: "Давай разберемся. Если ты вытягиваешь ловушку каждые 15 мин и, предположим, в ловушке 3 рака, то ты выловишь 12 раков в час или 100 раков в день. Я не верю этому!"

"Да, это правильно!", – ругнулся Поттер. – "Ты никогда ничему не веришь!"

Отец не поверил этому, а двумя неделями позже его друг сказал: "Ты помнишь этих двух леди, Фред? Они вылавливают только 20 фунтов в день".

Очень снисходительно отец проворчал: "Вот теперь я верю".

Некоторые читатели советуются, как обучать этому детей. Популярные вопросы имели такой вид: "Сколько времени потребовалось бы тебе, чтобы дойти до г. Вашингтона, округ Колумбия?" или "Сколько листьев мы сгребли в этом году?" Правильно поставленные такие вопросы, по-видимому, будят в детях фантазию на долгое время, изгоняя из них чертенка.

## Глава 7. МЕТОДЫ РАЗРАБОТКИ АЛГОРИТМОВ

В гл. 2 рассматривается общечеловеческий аспект влияния, которое может оказать на программистов разработка алгоритмов: "алгоритмический" взгляд на проблему позволяет проникнуть в ее суть, что может упростить понимание и написание программы. В этой главе мы рассмотрим область, где это влияние проявляется гораздо реже, но зато более впечатляюще: сложные алгоритмические методы иногда приводят к разительному увеличению эффективности.

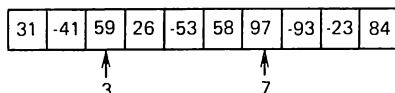
Данная глава основана на одной небольшой задаче с упором на алгоритмы ее решения и методы, использованные для разработки этих алгоритмов. Некоторые из этих алгоритмов довольно сложны, но тому

есть оправдание. В то время как первая программа, которую мы рассмотрим ниже, затрачивает на решение задачи размерностью 10 000 39 дней, последняя программа решает такую же задачу менее чем за 1 с.

### 7.1. ЗАДАЧА И ПРОСТОЙ АЛГОРИТМ ЕЕ РЕШЕНИЯ

Эта задача возникла при распознавании образов в одномерном случае; ее историю я опишу позже. На входе дан вектор  $X$ , содержащий  $N$  действительных чисел; выходом является максимальная сумма элементов среди всех непрерывных подвекторов входного вектора. Например, если входным вектором является вектор

Рис. 7.1



то программа возвращает на выходе сумму  $X[3 \dots 7]$  или 187. Задача проста, если все числа положительны — максимальным подвектором является весь входной вектор. Затруднения возникают, когда некоторые из чисел отрицательны: должны ли мы включать отрицательные числа в надежде на то, что положительные числа компенсируют их отрицательное влияние? Для того чтобы завершить постановку задачи, мы условимся, что, когда все числа на входе отрицательны, подвектор с максимальной суммой — пустой вектор, сумма элементов которого равна нулю.

Очевидный вариант программы для этой задачи перебирает все пары целых чисел  $L$  и  $U$ , удовлетворяющих условию  $1 \leq L \leq U \leq N$ , для любой пары вычисляет сумму элементов вектора  $X[L \dots U]$  и проверяет, больше ли эта сумма, чем максимальная сумма, найденная до сих пор. Программа на псевдоязыке для алгоритма 1 имеет вид

```

MaxSoFar := 0.0
/* Очистка ячейки для максимальной суммы */
for L := 1 to N do
  for U := L to N do
    Sum := 0.0
    for I := L to U do
      Sum := Sum + X[I]
    /* В Sum теперь содержится сумма X[L...U] */
  MaxSoFar := max(MaxSoFar, Sum)

```

Эта программа коротка, проста, ее легко понять. К несчастью, у нее очень серьезный недостаток – она медленно работает. Например, на компьютере, которым я обычно пользуюсь, эта программа выполнялась бы 1 ч, если  $N = 1000$ , и 39 дней, если  $N = 10\,000$ . До подробного разбора временных характеристик мы доберемся в разд. 7.5.

Такое время выполнения программы невероятно; оценим эффективность алгоритма интуитивно, с помощью записи "О-большое", описанной в разд. 5.1. Операторы самого внешнего цикла выполняются ровно  $N$  раз, а операторы среднего цикла выполняются не более  $N$  раз для каждого шага по внешнему циклу. Перемножение этих двух сомножителей показывает, что четыре строки, входящие в средний цикл, выполняются  $O(N^2)$  раз. Цикл, имеющийся в этих четырех строках, никогда не выполняется более  $N$  раз, так что на него затрачивается  $O(N)$ . Умножение затрат внутреннего цикла на число раз его выполнения показывает, что общие затраты на всю программу пропорциональны  $N^3$ , поэтому будем называть этот алгоритм кубическим.

Приведенный пример иллюстрирует анализ времени работы методом "О-большое" и показывает многие сильные и слабые стороны этого метода. Его главным недостатком является то, что мы на самом деле все еще не знаем, сколько времени потребуется программе для любых конкретных входных данных; мы только знаем, что число шагов его выполнения равно  $O(N^3)$ . Этот недостаток часто компенсируется двумя существенными достоинствами такого метода. Анализ методом "О-большое" обычно легко выполнить (как показано выше), и эта асимптотическая оценка времени работы часто бывает пригодной для прикидочных вычислений, позволяющих решить, достаточно ли эффективна программа для данной прикладной задачи.

В нескольких следующих разделах асимптотическая оценка времени работы используется в качестве единственной меры эффективности программ. Если это будет вас раздражать, загляните в разд. 7.5, где показано, что для данной задачи такой анализ чрезвычайно информативен. Однако перед тем, как читать дальше, потратьте минутку, чтобы попытаться найти быстрее работающий алгоритм.

## 7.2. ДВА КВАДРАТИЧНЫХ АЛГОРИТМА

Большинство программистов одинаково реагируют на алгоритм 1: "Есть очевидный способ намного увеличить его быстродействие". Однако есть два очевидных способа, и если один очевиден для данного программиста, то другой – часто нет. Оба алгоритма квадратичны – они выполняются за  $O(N^2)$  шагов для входных данных размерностью  $N$ , и в обоих алгоритмах такое быстродействие достигнуто с помощью вычисления суммы элементов вектора  $X[L \dots U]$  за постоянное число шагов,

а не за  $U - L + 1$  шагов, как в алгоритме 1. Однако в этих двух квадратичных алгоритмах используются совершенно разные методы вычисления суммы за постоянное время.

В первом квадратичном алгоритме быстрое вычисление суммы основано на том факте, что сумма элементов вектора  $X[L \dots U]$  имеет тесную взаимосвязь с ранее вычисленной суммой для вектора  $X[L \dots U - 1]$ . Использование этой взаимосвязи приводит к алгоритму 2.

```
MaxSoFar := 0.0
/* Очистка ячейки для максимальной суммы */
for L := 1 to N do
    Sum := 0.0
    for U := L to N do
        Sum := Sum + X[U]
    /* В Sum теперь содержится сумма X[L...U] */
    MaxSoFar := max(MaxSoFar, Sum)
```

Операторы внутри первого цикла выполняются  $N$  раз, а внутри второго — не более чем  $N$  раз для каждого шага внешнего цикла, поэтому общее время работы составляет  $O(N^2)$ .

Альтернативный квадратичный алгоритм вычисляет сумму во внутреннем цикле, обращаясь к структуре данных, созданной до начала выполнения внешнего цикла. В элементе  $I$  массива CumArray содержится кумулятивная сумма элементов вектора  $X[1 \dots N]$ , так что сумма элементов вектора  $X[L \dots U]$  может быть найдена с помощью вычисления разности  $CumArray[U] - CumArray[L - 1]$ . Отсюда следует алгоритм 2b:

```
CumArray[0] := 0.0
for I := 1 to N do
    CumArray[I] := CumArray[I-1] + X[I]
MaxSoFar := 0.0
for L := 1 to N do
    Sum := CumArray[U] - CumArray[L-1]
    /* Теперь в Sum содержится сумма X[L...U] */
    MaxSoFar := max(MaxSoFar, Sum)
```

Этой программе требуется время, равное  $O(N^2)$ ; анализ точно такой же, как и анализ алгоритма 2.

В алгоритмах, которые мы изучали до сих пор, рассматриваются все возможные пары начальных и конечных элементов при выборе подвектора и вычисляются суммы элементов этих подвекторов. Так как количество подвекторов составляет  $O(N^2)$ , время работы любого алгоритма должно иметь по крайней мере квадратичную зависимость от  $N$ . Можете ли вы придумать способ обойти эту проблему и получить алгоритм с меньшим временем работы?

### 7.3. АЛГОРИТМ "РАЗДЕЛЯЙ И ВЛАСТВУЙ"

Первый из "субквадратных алгоритмов" (с зависимостью ниже квадратичной) сложен: если вы увязните в его деталях, то немного потеряете, пропустив его и перейдя к следующему разделу. Данный алгоритм основан на такой схеме:

чтобы решить задачу размерности  $N$ , надо рекурсивно решить две подзадачи размерностью  $N/2$  и объединить их решения.

В этом случае в исходной задаче рассматривается вектор размерности  $N$ , поэтому наиболее естественный способ разделить ее на подзадачи – создать два вектора примерно одинакового размера, которые мы назовем  $A$  и  $B$ :

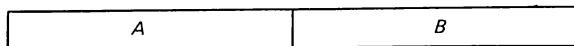


Рис. 7.2

После этого мы рекурсивно найдем максимальные подвекторы векторов  $A$  и  $B$ , которые назовем  $M_A$  и  $M_B$ :

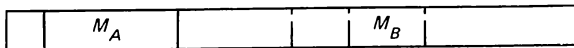


Рис. 7.3

Хотелось бы думать, что мы решили задачу, так как подвектором с максимальной суммой должен быть либо  $M_A$ , либо  $M_B$ , и это почти верно. Фактически максимум или целиком в векторе  $A$ , или целиком в векторе  $B$ , или пересекает границу между векторами  $A$  и  $B$ ; в последнем случае мы назовем его  $M_C$ :

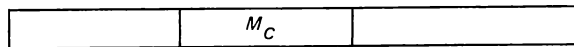


Рис. 7.4

Таким образом наш алгоритм "разделяй и властвуй" рекурсивно вычислит суммы векторов  $M_A$  и  $M_B$ , каким-то другим способом вычислит сумму вектора  $M_C$ , а затем вернет максимальное среди них значение.

Этого описания почти достаточно, чтобы написать программу. Все, что осталось описать – как мы будем обрабатывать маленькие векторы и вычислять  $M_C$ . Первое сделать несложно: максимум для вектора из одного элемента – это или значение этого элемента, или нуль, если значение элемента отрицательно; максимум для вектора с нулевым количеством элементов был определен ранее равным нулю. При вычислении  $M_C$  отметим, что его компонент в векторе A – это максимальный подвектор, начинающийся на границе и простирающийся в A, аналогично – для его компонента в векторе B. Объединение вместе этих векторов приводит к следующей программе для алгоритма 3, которая запускается обращением к процедуре:

```
Answer := MaxSum(1,N)
```

```
recursive function MaxSum(L, U)
```

```
  if L > U then    /* Вектор из 0 элементов */
```

```
    return 0.0
```

```
  if L = U then    /* Вектор из 1 элемента */
```

```
    return max(0.0, X[L])
```

```
  M := (L+U)/2    /* X[L...M] это A; X[M+1...U] это B */
```

```
  /* Найти максимум влево от точки разбиения M */
```

```
    Sum := 0.0; MaxToLeft := 0.0
```

```
    for I := M downto L do
```

```
      /* downto - цикл с уменьшением индекса */
```

```
        Sum := Sum + X[I]
```

```
        MaxToLeft := max(MaxToLeft, Sum)
```

```
  /* Найти максимум вправо от точки разбиения M */
```

```
    Sum := 0.0; MaxToRight := 0.0
```

```
    for I := M+1 to U do
```

```
      Sum := Sum + X[I]
```

```
      MaxToRight := max(MaxToRight, Sum)
```

```

MaxCrossing := MaxToLeft + MaxToRight
MaxInA := MaxSum(L,M)
MaxInB := MaxSum(M+1,U)
return max(MaxCrossing, MaxInA, MaxInB)

```

Эта программа сложна, в ней просто сделать ошибку, но она решает задачу за время  $O(N \log N)$ . Есть несколько способов доказательства этого факта. Неформальным доводом является следующее наблюдение: алгоритм выполняет объем работы  $O(N)$  на каждом из  $O(\log N)$  уровней рекурсии. Этот довод можно уточнить, используя рекуррентные соотношения. Если обозначить время решения задачи размерности  $N$  через  $T(N)$ , тогда  $T(1) = O(1)$  и

$$T(N) = 2T(N/2) + O(N).$$

В задаче 11 показано, что это рекуррентное соотношение имеет решение  $T(N) = O(N \log N)$ .

#### 7.4. СКАНИРУЮЩИЙ АЛГОРИТМ

Сейчас мы воспользуемся простейшим из алгоритмов, работающих с массивами: он начинает с левой границы массива (элемент  $X[1]$ ) и осуществляет сканирование до его правой границы (элемент  $X[N]$ ), отслеживая подвектор с максимальной найденной к данному моменту суммой. В начальный момент максимум равен 0. Предположим, что мы решили задачу для вектора  $X[1 \dots I - 1]$ . Как можно распространить это решение на первые  $I$  элементов? Наши рассуждения будут аналогичны тем, которые были выполнены для алгоритма "разделяй и властвуй": максимальная сумма первых  $I$  элементов равна максимальной сумме первых  $I - 1$  элементов (которую мы назовем *MaxSoFar*) или максимальной сумме подвектора, заканчивающегося в позиции  $I$  (которую мы назовем *MaxEndingHere*).

	<i>MaxSoFar</i>		<i>MaxEndingHere</i>
--	-----------------	--	----------------------

Рис. 7.5

Вычисление переменной *MaxEndingHere* каждый раз заново с нуля при использовании программы, похожей на программу из алгоритма 3, приводит еще к одному квадратичному алгоритму. Мы можем избежать

этого, применяя метод, который привел нас к алгоритму 2: вместо того, чтобы вычислять максимальный подвектор, оканчивающийся в позиции I, заново, мы используем максимальный подвектор, оканчивающийся в позиции I – 1. Таким образом, получаем алгоритм 4:

```
MaxSoFar := 0.0
MaxEndingHere := 0.0
for I := 1 to N do

    /* Инвариант: MaxEndingHere и MaxSoFar вычислены точно
       для X[1..I-1] */

    MaxEndingHere := max(MaxEndingHere + X[I], 0.0)
    MaxSoFar := max(MaxSoFar, MaxEndingHere)
```

Ключом к пониманию этой программы является переменная MaxEndingHere. Перед первым оператором присваивания в цикле переменная MaxEndingHere содержит сумму элементов максимального подвектора, оканчивающегося в позиции I – 1. Оператор присваивания изменяет ее так, чтобы она содержала сумму элементов максимального подвектора, оканчивающегося в позиции I. Этот оператор увеличивает ее на значение элемента X[I] до тех пор, пока она остается положительной; когда переменная становится отрицательной, оператор устанавливает ее в нуль, так как максимальный подвектор, оканчивающийся в позиции I, – пустой вектор. Хотя эта программа довольно сложная, но короткая и быстродействующая: время ее работы составляет O(N), так что назовем данный алгоритм линейным алгоритмом. Д. Гриз методично вывел и исследовал этот алгоритм в своей статье "Заметки по поводу стандартной стратегии разработки циклов и инвариантов циклов" (David Gries. A Note on the Standart Strategy for Development Loop Invariants and Loops) в журнале Science of Computer Programming 2, с. 207 – 214.

## 7.5. ПРОВЕРКА НА ПРАКТИКЕ

До сих пор мы бегло и достаточно свободно развлекались с "О-большими", теперь пришло время рассказать об истинном времени работы этих программ. Я реализовал четыре основных алгоритма (все, за исключением алгоритма 2b) на языке Си для компьютера VAX 11/750, измерил время их работы и экстраполировал результаты измерений, чтобы получить следующую таблицу:



Алгоритм		1	2	3	4
Число строк на языке СИ		8	7	14	7
Время работы, мкс		$3.4N^3$	$13N^2$	$46N \log_2 N$	$33N$
Время решения задачи размерности	$10^2$	3.4 с	.13 с	.03 с	.003 с
	$10^3$	.94 ч	13 с	.45 с	.033 с
	$10^4$	39 дней	22 мин	6.1 с	.33 с
	$10^5$	108 лет	1.5 дней	1.3 мин	3.3 с
	$10^6$	108 млн лет	5 мес	15 мин	33 с
Максимальная размерность задачи, решаемой за	с	67	280	2000	30000
	мин	260	2200	82000	2000000
	ч	1000	17000	3500000	120000000
	день	3000	81000	73000000	2800000000
Если N умножить на 10, время возрастет в (раз)		1000	100	10+	10
Если время умножить на 10, N увеличится в (раз)		2.15	3.16	10—	10

Эта таблица доказывает ряд положений. Наиболее важное из них то, что правильная разработка алгоритма может привести к большой разнице во время работы программы. Это положение иллюстрируют строки в середине таблицы. Две последние строки показывают, как увеличение размерности задачи связано с увеличением времени работы.

Другим важным моментом является то, что, когда мы сравниваем кубические, квадратичные и линейные алгоритмы друг с другом, коэффициенты (константы программы) не имеют большого значения. (Исследование алгоритма  $O(N!)$  в разд. 2.4 показывает, что коэффициенты имеют еще меньшее значение для функций со скоростью возрастания выше, чем полиномиальная.) Чтобы подчеркнуть этот момент, я провел эксперимент, в котором постарался сделать постоянные множители двух алгоритмов настолько различными, насколько это возможно. Чтобы получить огромный постоянный множитель я реализовал алгоритм 4 с помощью интерпретатора Бейсика на микрокомпьютере TRS-80, модель III. Для получения другого крайнего случая Э. Гроссе и я реализовали алгоритм 1 на суперкомпьютере Cray-1 с использованием хорошо оптимизирующего компилятора Фортрана. Мы добились различия, которого хотели: измерение времени работы кубического алгоритма дало  $3.0N^3$  нс, тогда, как время работы линейного алгоритма было  $19.5N$  мс или  $19\,500\,000$  нс. В приведенной таблице показано соответствующее время работы алгоритмов для различных размерностей задачи.

N	Cray-1 Фортран Кубический алгоритм	TRS-80 Бейсик Линейный алгоритм
10	3.0 мкс	200 мс
100	3.0 мс	2.0 с
1000	3.0 с	20 с
10 000	49 мин	3.2 мин
100 000	35 дней	32 мин
1 000 000	95 лет	5.4 ч

Различие в постоянных множителях, составляющее 6.5 млн, позволяет кубическому алгоритму стартовать с лучшим временем, но линейный алгоритм обязан вырваться вперед. Переломный момент наступает при  $N = 2500$ , где каждый из этих алгоритмов затрачивает около 50 с:

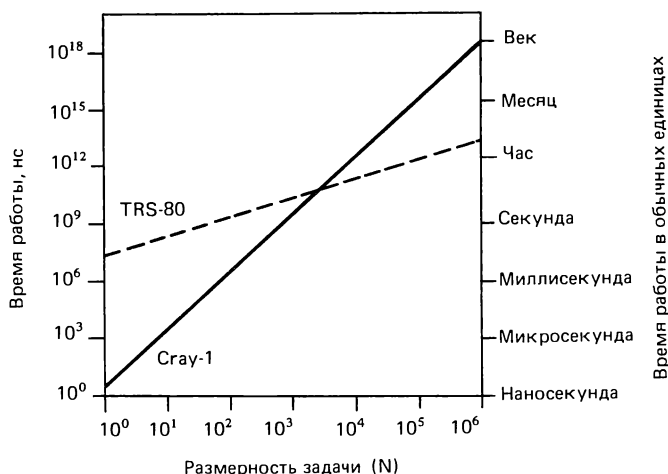


Рис. 7.6

## 7.6. ОСНОВНЫЕ ПРИНЦИПЫ

История этой задачи проливает свет на методы разработки алгоритмов. Задача возникла в процедуре сопоставления с эталоном, разработанной У. Гренандером из Университета Брауна, и описана для двумерного случая в задаче 7, где подмассив с максимальной суммой являлся оценкой с максимальной вероятностью некоторого эталонного изображения, представленного в виде цифр. Так как для решения двумерной задачи потребовалось слишком много времени, Гренандер свел ее к одномерной, чтобы лучше понять ее структуру.

Он отметил, что кубическая зависимость времени выполнения алгоритма 1 делала его чересчур медленным и вывел алгоритм 2. В 1977 г. он описал эту задачу М. Шамосу из фирмы UNILogic, Ltd. (позднее он работал в университете Карнеги-Меллон), который разработал за ночь алгоритм 3. Когда Шамос показал мне решение задачи, мы подумали, что это, вероятно, лучшее, чего можно достичь. Исследователи как раз продемонстрировали, что в ряде похожих задач требуется время, пропорциональное  $N \log N$ . Несколькими днями позже Шамос рассказал об этой задаче и ее истории на семинаре в университете Карнеги-Меллон, на котором присутствовал статистик Дж. Кадан, придумавший за 1 мин алгоритм 4, и более быстро работающего алгоритма нет: время работы любого правильного алгоритма не должно превышать  $O(N)$ .

Однако, хотя одномерная задача полностью решена, исходная двумерная задача Гренандера оставалась открытой даже через 8 лет после того, как она была поставлена (т. е. когда эта книга была подписана в печать). Вследствие вычислительной дороговизны всех известных алгоритмов Гренандер вынужден был отказаться от этого подхода к проблеме сопоставления с эталоном. Читателям, чувствующим, что алгоритм с линейной зависимостью времени для одномерной задачи очевиден, предлагается найти очевидный алгоритм для задачи 7!

Алгоритмы, о которых рассказано в этой истории, никогда не были включены в какую-либо систему, но они иллюстрируют важные методы разработки алгоритмов и оказали существенное влияние на проектирование многих систем (см. разд. 7.9).

*Сохраняйте значения переменных, чтобы избежать повторных вычислений.* Динамическое программирование в простом виде имело место в алгоритмах 2 и 4. Используя дополнительную память для хранения результатов, мы исключаем затраты времени на их повторное вычисление.

*Проводите предварительную обработку информации для построения структур данных.* Структура массива `SumArray` в алгоритме 2b позволяет вычислять сумму подвектора, используя только пару операций.

*Алгоритмы "разделяй и властвуй".* В алгоритме 3 используется этот принцип в простом виде. В учебниках по разработке алгоритмов приведены более сложные виды.

*Сканирующие алгоритмы.* Задачи с массивами часто можно решить, задав себе вопрос: "Как я могу расширить решение, полученное для вектора  $X[1 \dots I-1]$ , на случай вектора  $X[1 \dots I]$ ?" В алгоритме 4 запоминается как предыдущий результат, так и некоторые дополнительные данные для вычисления нового результата.

*Кумулятивные суммы.* В алгоритме 2b используется таблица кумулятивных сумм, в которой элемент  $I$  содержит сумму первых  $I$  элементов вектора  $X$ . Такие таблицы часто используют, имея дело с рядами. Например, в прикладных задачах обработки данных в сфере бизнеса можно определить объем продаж с марта по октябрь, если вычесть из объема продаж с начала года по октябрь объем продаж с начала года по февраль.

*Нижний предел.* Разработчик алгоритма мирно спит только тогда, когда знает, что его алгоритм лучший из возможных; чтобы иметь такую уверенность, он должен доказать соответствие нижнему пределу. Задача 9 посвящена нахождению линейного нижнего предела; для более сложных зависимостей найти нижний предел может оказаться гораздо труднее.

## 7.7. ЗАДАЧИ

1. Алгоритмы 3 и 4 реализуются сложными программами, в которых легко допустить ошибки. Чтобы доказать корректность этих программ воспользуйтесь методами верификации программ из гл. 4; тщательно определите инварианты циклов.
2. При анализе четырех алгоритмов рассматривалась только функция "О-большое". Проанализируйте функции  $\Theta$  в каждом из этих алгоритмов с максимально возможной точностью. Дает ли это упражнение понимание того, каково истинное время работы программ? Какой объем памяти требуется для реализации каждого алгоритма?
3. Мы определили, что максимальный подвектор массива отрицательных элементов равен нулю, т. е. сумме элементов пустого вектора. Предположим, что вместо этого мы определили максимальный подвектор, равный значению наибольшего элемента. Как вы измените программы?
4. Предположим, что нам нужно найти подвектор не с максимальной суммой, а с суммой, наиболее близкой к нулю. Какой самый эффективный алгоритм вы можете разработать для этой задачи? Какие методы разработки алгоритмов здесь применимы? А что, если мы захотим найти подвектор с суммой, наиболее близкой к некоторому заданному действительному числу?
5. Сеть платных дорог состоит из  $N - 1$  дорог между  $N$  пунктами оплаты. За проезд по каждой дороге назначена своя цена. Какова стоимость проезда между любыми двумя пунктами? Тривиальное решение: сообщать цену проезда между любыми двумя пунктами с временем задержки ответа, составляющим  $O(N)$ , используя только массив этих цен, или с постоянным временем задержки, используя таблицу из

$O(N^2)$  элементов. Опишите структуру данных, для которой требуется  $O(N)$  ячеек памяти, но она позволяет вычислять цену любого маршрута за постоянное время.

6. После записи начальных нулевых значений в массив  $X[1 \dots N]$  выполняются следующие  $N$  операций:

$$\text{for } I := L \text{ to } U \text{ do}$$
$$X[I] := X[I] + V$$

где  $L$ ,  $U$  и  $V$  являются параметрами операции ( $L$  и  $U$  – целые числа, удовлетворяющие выражению  $1 \leq L \leq U \leq N$ , а  $V$  – действительное число). После этих  $N$  операций значения элементов выводятся по порядку. Описанный метод требует  $O(N^2)$  времени). Можете ли вы найти более быстрый алгоритм?

7. В задаче определения максимума подмассива нам дан массив действительных чисел размерностью  $N \times N$ , и мы должны найти максимальную сумму, содержащуюся в любом из прямоугольных подмассивов. В чем сложность этой задачи?
8. Модифицируйте алгоритм 3 (алгоритм "разделяй и властвуй") так, чтобы он имел в наихудшем случае линейную зависимость времени работы от  $N$ .
9. Докажите, что любой правильный алгоритм вычисления максимального подвектора должен обработать все  $N$  входных чисел. (Алгоритмы для некоторых задач могут игнорировать отдельные данные на входе, не нарушая корректности решения. Рассмотрите алгоритм Сакса в решении задачи 2 гл. 2 и алгоритм поиска фрагмента строки Бойера и Мура в октябрьском номере журнала CACM за 1977 г.)
10. Даны целые числа  $M$  и  $N$  и вектор действительных чисел  $X[1 \dots N]$ . Найти целое число  $I$  ( $1 \leq I \leq N - M$ ), для которого сумма  $X[I] + \dots + X[I + M]$  ближе всего к нулю.
11. Каково решение рекуррентного соотношения  $T(N) = 2T(N/2) + CN$ , где  $T(1) = 0$ ,  $N$  равно степени числа 2? Докажите полученный результат с помощью математической индукции. Что будет, если  $T(1) = C$ ?

## 7.8. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Только в процессе всестороннего обучения можно в совершенстве освоить методы разработки алгоритмов. Большинство программистов могут достичь этого только с помощью учебников по алгоритмам. Книга "Структуры данных и алгоритмы" Ахо, Хопкрофта и Ульмана (Aho, Hopcroft, Ulman. Data Structures and Algorithms), опубликованная изда-

тельством Addison-Wesley в 1983 г. является блестящим учебником для изучающих этот предмет. Теме данной главы особенно близка гл. 10 этой книги "Методы разработки алгоритмов" (Algorithm Design Technics).

#### 7.9. ВЛИЯНИЕ АЛГОРИТМОВ НА КАЧЕСТВО ПРОГРАММ (ДОПОЛНЕНИЕ)

Хотя задача, рассмотренная в этой главе, иллюстрирует ряд важных примеров, в действительности она является игрушкой, никогда не включаемой в реальную систему. Сейчас мы дадим обзор нескольких реальных задач, в которых методы разработки алгоритмов доказали свою ценность.

*Численный анализ.* Обычным примером эффективности алгоритмической разработки является дискретное быстрое преобразование Фурье (БПФ). Его структура, основанная на методе "разделяй и властвуй", сокращает время, требуемое анализом Фурье, с  $O(N^2)$  до  $O(N \log N)$ . Так как в задачах обработки сигналов и анализа временных рядов обрабатываются входные последовательности длиной  $N = 1000$  или больше, этот алгоритм ускоряет программу более, чем в 100 раз.

В разд. 10.3.С своей книги "Численные методы, программное обеспечение и анализ" (John Rice. Numerical Methods, Software and Analysis), выпущенной издательством McGraw-Hill в 1983 г., Дж. Райс привел хронику развития алгоритмов решения трехмерных эллиптических уравнений в частных производных. Такие задачи возникают при моделировании СБИС, нефтяных скважин, ядерных реакторов, крыльев самолетов. Частично хронология развития алгоритмов приведена в следующей таблице. В графе "Время работы" содержится количество операций с плавающей точкой, требуемых для решения задачи на сетке  $N \times N \times N$ .

Метод	Год	Время работы
Исключение Гаусса	1945	$N^7$
ПВР-итерация:		
субоптимальные параметры	1954	$8N^5$
оптимальные параметры	1960	$8N^4 \log_2 N$
Циклическое приведение	1970	$8N^3 \log_2 N$
Мультисетка	1978	$60N^3$

ПВР означает "последовательная верхняя релаксация". Время порядка  $O(N^3)$  для метода мультисетки является, с точностью до постоянного множителя, оптимальным, так как задача имеет такое же количество входных данных. Для задачи типичной размерности ( $N = 64$ ) коэффициент увеличения скорости равен 0.25 млн. В колонке "Жемчужины

программирования" ноябрьского номера журнала Communications of the ACM за 1984 г. (с. 1090 – 1091) приведены данные, подтверждающие утверждение Райса, что ускорение, достигнутое за счет совершенствования алгоритмов с 1945 по 1970 гг. превышало рост скорости ЭВМ за этот период.

*Алгоритмы работы с графами.* При обычном методе построения цепей интегральной схемы разработчик описывает электрическую цепь в виде графа, по которому позже конструируется микросхема. В принятом подходе к разработке топологии микросхем, чтобы разделить всю электрическую схему на компоненты, используется задача разбиения графа. Для разработанных в начале 70-х годов эвристических алгоритмов разбиения графа при разбиении схемы, имеющей в сумме  $N$  элементов и проводников, требуется время, пропорциональное  $O(N^2)$ . Фидуччи и Мэттиас описали "Эвристику с линейным временем для улучшения разбиения цепей" (Fiduccia, Mattheyses. A linear-time heuristic for improving network partition). Так как в типичных случаях приходится иметь дело с несколькими тысячами элементов, их метод сокращает время трассировки микросхемы с нескольких часов до нескольких минут.

*Геометрические алгоритмы.* На последних этапах проектирования интегральной схемы получают ее топологию, согласно которой в конечном счете протравливается кристалл. Системы проектирования обрабатывают эту топологию, например, для отслеживания электрических цепей и сравнения с цепями, заданными проектировщиком. Когда интегральные схемы состояли из  $N = 1000$  геометрических элементов, которые соответствовали 100 транзисторам, алгоритмы, сравнивающие все пары геометрических элементов за время  $O(N^2)$ , могли решить задачу за несколько минут. Теперь, когда СБИС содержит миллионы геометрических элементов, квадратичным алгоритмам для решения такой задачи потребовались бы месяцы. С помощью алгоритмов типа "разделяй и властвуй" или "сканирования строки" удалось уменьшить время работы программ до  $O(N \log N)$ , так что теперь схемы могут быть обработаны за несколько часов. В докладе Жимански и Ван Уайка "Эффективные по памяти алгоритмы для анализа топологии СБИС" (Szimanski, Van Wyk. Space efficient algorithms for VLSI artwork analysis) на 20-й конференции по автоматизации проектирования описаны эффективные алгоритмы для решения таких задач, использующие ОЗУ объемом только  $O(\sqrt{N})$  (более поздний вариант их статьи опубликован в июньском номере журнала IEEE Design and Test за 1985 г.).

В программе Аппела, описанной в разд. 5.1, используется древовидная структура данных для отображения точек трехмерного простран-

ва, и поэтому алгоритм с временем работы  $O(N^2)$  сводится к алгоритму с временем работы  $O(N \log N)$ . Это было первым шагом на пути сокращения времени всей программы с одного года до одного дня.

## Глава 8. ОПТИМИЗАЦИЯ ПРОГРАММЫ

Некоторые программисты уделяют слишком много внимания эффективности: чересчур заботясь о маленьких "улучшениях", они создают крайне изощренные программы, которые очень тяжело сопровождать. Другие уделяют эффективности слишком мало внимания: они выдают прекрасно структурированные программы, которые совершенно неэффективны и поэтому бесполезны. Хорошие программисты отводят эффективности соответствующее место, и это – только одна из многих задач при разработке программного обеспечения, но иногда она очень важна.

В предыдущих главах рассматривались вопросы эффективности на верхних уровнях: постановка задачи, структура системы, разработка алгоритма и выбор структуры данных. Эта глава посвящена нижнему уровню. При оптимизации программы сначала локализуют неэффективные части, а потом вносят в программу небольшие изменения, чтобы улучшить ее характеристики. Это не тот подход, которому надо следовать всегда, и он редко оказывается действенным, но иногда может существенно изменить производительность программы.

### 8.1. ТИПИЧНАЯ ИСТОРИЯ

Однажды после полудня К. Ван Уайк и я вели дружескую беседу о методах оптимизации программ. Потом он ушел, чтобы испытать их на практике. К концу дня он сократил время работы программы из 3000 строк вдвое. Для получения изображения по заданному в виде текста описанию изображения эта программа выдает команды для фотонаборного устройства. Программа работает около 10 мин, чтобы нарисовать чрезвычайно сложное изображение, хотя для типичной картинки время работы намного меньше. Первым шагом Ван Уайка было *профилирование* программы с помощью задания компилятору ключа, который заставляет систему сообщать, сколько времени было затрачено на каждую процедуру (формат выдачи подобен второму столбцу в решении задачи 10). Прогон программы на 10 тестовых изображений показал, что почти 70 % всего времени приходится на программу распределения памяти.



Следующим его шагом было изучение подпрограммы распределения памяти. Несколько команд, добавленные для подсчета, показали, что запрос на размещение наиболее часто встречающейся записи выполнялся 68 000 раз, а для следующей по частоте записи – только 2000 раз. Предположим, вы знаете, что основное время программы тратится на просмотр памяти с целью поиска записи единственного типа; как бы вы изменили программу, чтобы повысить ее быстродействие?

Ван Уайк решил свою задачу, применив принцип кэширования: к данным, которые требуются наиболее часто, должен быть обеспечен самый "дешевый" доступ. Он модернизировал свою программу, объединив свободные записи наиболее часто встречающегося типа в связанный список. После этого стало возможным быстро обрабатывать наиболее часто встречающиеся запросы, обращаясь к этому списку, вместо того, чтобы вызывать универсальную программу распределения памяти. Это сократило общее время работы его программы как раз на 45 % относительно предыдущего значения (таким образом, для подпрограммы распределения памяти теперь требуется 30 % общего времени). Дополнительным выигрышем было то, что модифицированная подпрограмма распределения памяти снизила степень фрагментации памяти, вследствие чего повысилась эффективность использования ОЗУ по сравнению с исходным вариантом программы.

Эта история иллюстрирует лучшие стороны искусства оптимизации программ. Затратив несколько часов и добавив около 20 строк в программу из 2000 строк, Ван Уайк удвоил ее быстродействие, не изменив программу с точки зрения пользователя и не усложнив ее сопровождение. Он использовал обычные средства повышения быстродействия: профилирование выявило "узкие места" в его программе, а кэширование сократило затраты времени.

## 8.2. ПЕРВАЯ ПОМОЩЬ

Сейчас мы обратимся к трем задачам, являющимся фрагментами более сложных задач. Каждая из них иллюстрирует типичную проблему, которая появляется в различных прикладных задачах, но при решении этих проблем используются общие принципы.

Первая задача возникает при обработке текстов, в компиляторах, при обработке макроопределений и в интерпретаторах команд.

*Задача 1 – классификация символов.* Дана последовательность 1 млн символов. Определить для каждого из них, является ли он строчной буквой, прописной буквой, цифрой или чем-то другим.

В очевидном решении для каждого символа выполняется сложная последовательность проверок. Если символы записаны в коде ASCII, то,

чтобы определить, что данная буква имеет тип "что-то другое", при таком подходе требуется 6 сравнений, в то время как в коде EBDIC для получения этого же результата потребуется 14 сравнений. Можете ли вы предложить лучшее решение?

В одном из подходов используется двоичный поиск – подумайте об этом. Более быстрое решение получается, если рассматривать символ как индекс в массиве типов символов. В большинстве языков программирования есть способ осуществить это. В качестве примера предположим, что код символа 8-битовый, так что каждый символ можно рассматривать как целое число в диапазоне от 0 до 255. В массив TypeTable [0 ... 255] записываются начальные значения, например, так, как в приведенном ниже фрагменте (хотя знающие программисты могут попытаться выполнить эту работу, используя циклы).

```
for I := 0 to 255 do TypeTable[I] := Other
TypeTable['a'] := ... := TypeTable['z'] := LCLetter
TypeTable['A'] := ... := TypeTable['Z'] := UCLetter
TypeTable['0'] := ... := TypeTable['9'] := Digit
/* Обозначение типов:
      LCLetter - строчные буквы; UCLetter - прописные буквы;
      Digit - цифры; Other - другие; */
```

После этого тип символа C может быть определен с помощью массива TypeTable [C], что заменяет сложную последовательность сравнений символов одним обращением к массиву. Такая структура данных обычно сокращает время классификации символов на порядок.

Следующая проблема возникает во многих прикладных задачах – от отображения состояний до теории кодирования.

*Задача 2 – подсчет битов.* Дана последовательность из 1 млн 32-разрядных слов; определите, сколько битов установлено в единицу в каждом слове.

В очевидной программе используется цикл для выполнения 32 сдвигов логических операций AND. Можете ли вы придумать лучший способ?

Принцип построения алгоритма тот же, что и раньше: таблица, в которой элемент I содержит число битов, установленных в единицу в двоичном представлении I. К несчастью, большинство компьютеров имеют недостаточный объем памяти, чтобы хранить таблицу из  $2^{32}$  элементов, а время, необходимое на установку начальных значений 4 млрд. элементов, также может оказаться препятствием. Эти сложности

можно обойти, введя несколько дополнительных операций и используя таблицы меньшего размера. Мы зададим таблицу, содержащую число единичных битов для всех 8-битовых байтов, а потом ответим на вопрос о 32-разрядных словах, суммируя ответы на четыре вопроса о 8-разрядных словах. Таблица CountTable, содержащая число единичных битов, устанавливается в исходное состояние двумя циклами, которые по своему действию соответствуют следующим операторам присваивания (см. задачу 2).

```
CountTable[0] := 0;      CountTable[1] := 1;
CountTable[2] := 1;      CountTable[3] := 2;
...
CountTable[254] := 7;     CountTable[255] := 8;
```

Для подсчета битов во всех байтах слова W мы могли бы использовать четырехпроходный цикл, но так же просто и, вероятно, несколько эффективнее развернуть этот цикл:

```
WordCount := CountTable[ W          and 11111111B]
              + CountTable[(W rshift 8) and 11111111B]
              + CountTable[(W rshift 16) and 11111111B]
              + CountTable[(W rshift 24) and 11111111B]
```

Программа выделяет байты посредством сдвига и последующей операции AND, которая устанавливает в нуль все биты, кроме восьми младших. Эта операция зависит от языка и типа ЭВМ. В то время как в исходном варианте решения требуется около 100 машинных команд, вышеприведенный подход обычно можно реализовать, используя около дюжины операторов. Нередко такое изменение приводит (опять) к ускорению времени выполнения программы на порядок. (Некоторые другие подходы к подсчету числа битов рассматриваются Рейнгоулдом, Нивергильтом и Део в разд. 1.1 книги "Комбинированные алгоритмы: теория и практика" (Reingold, Nievergelt, Deo. Combinatorial Algorithms: Theory and Practice), опубликованной издательством Prentice-Hall в 1977 г.)

Последняя проблема типична для прикладных задач, имеющих дело с географическими и геометрическими данными.

*Задача 3 – вычисление расстояний на сфере.* Первая часть входных данных – множество S из 5 тыс. точек на поверхности шара. Каждая точка представляется своей широтой и долготой. После того, как эти

точки записаны в виде выбранной нами структуры данных, программа читает вторую часть входных данных: последовательность из 20 тыс. точек, любая из которых представляется широтой и долготой. Для каждой точки этой последовательности программа должна сообщить, какая точка множества  $S$  к ней ближе всего при условии, что расстояние измеряется как угол между лучами, проведенными из центра шара к этим двум точкам.

М. Райт из Станфордского университета столкнулась с подобной задачей при подготовке географических карт, обобщающих данные всемирного распределения определенных генетических особенностей. В ее простом решении множество  $S$  соответствовало массиву широт и долгот. Ближайший сосед к каждой точке последовательности находился с помощью вычисления расстояния от этой точки до каждой точки множества  $S$  с использованием сложных тригонометрических формул, включающих 10 функций SIN и COS. Хотя текст программы был прост и для маленьких наборов данных получались прекрасные карты, чтобы получить большие карты, требовалось несколько часов на большой ЭВМ, что сильно превышало запланированные расходы.

Так как раньше я имел дело с геометрическими задачами, Райт попросила меня приложить руку и к этой. Затратив большую часть уик-энда, я разработал несколько затейливых алгоритмов и структур данных для ее решения. К счастью (как оказалось впоследствии), для реализации каждого потребовалась бы программа из многих сотен строк, так что я не пытался запрограммировать ни один из них. Когда я описал эти структуры данных Э. Аппелу из университета Карнеги-Меллон, он обратил внимание на главное: вместо того, чтобы подходить к этой задаче на уровне структур данных, почему бы не использовать простые структуры данных для хранения точек в массиве, но оптимизировать программу, чтобы уменьшить затраты на вычисление расстояния между точками? Как бы вы воспользовались этой идеей?

Затраты можно резко сократить, изменив представление точек: вместо того, чтобы использовать широту и долготу, мы будем представлять положение точки на поверхности шара координатами  $X$ ,  $Y$  и  $Z$ . Таким образом, структура данных представляет собой массив, в котором содержится широта и долгота каждой точки (они могут понадобиться для других операций), а также три ее декартовы координаты. При обработке каждой точки последовательности несколько тригонометрических функций преобразуют ее широту и долготу в координаты  $X$ ,  $Y$  и  $Z$ , а потом определяется расстояние от нее до каждой точки множества  $S$ . Расстояние вычисляется как квадрат разностей этих трех координат, что обычно дешевле по времени, чем вычисление одной тригонометрической функции, не говоря уже о десяти. Этот метод дает правильный

ответ, так как величина угла между двумя точками монотонно возрастает в соответствии с квадратом евклидова расстояния между ними.

Хотя такой подход требует дополнительной памяти, но дает существенный выигрыш: когда Райт внесла это изменение в программу, время работы для сложных карт сократилось с нескольких часов до полминуты. В этом случае оптимизацией программы задача была решена с помощью пары дюжин строк, в то время как изменение алгоритма и структур данных потребовали бы написания многих сотен строк.

### 8.3. "ОСНОВНОЕ ЛЕЧЕНИЕ" – ДВОИЧНЫЙ ПОИСК

Сейчас мы перейдем от "оказания первой помощи" к "основному лечению" программы. Данный случай является одним из самых сложных примеров оптимизации программы, которые я знаю. Подробности взяты из задачи 8 гл. 4: мы должны выполнить двоичный поиск в таблице, содержащей 1000 целых чисел. Выполняя это упражнение, помните, что такие меры, как правило, не нужны в этой программе – алгоритм двоичного поиска настолько эффективен, что оптимизация программы часто бывает излишней. Поэтому в гл. 4 мы игнорировали возможности микроскопического повышения эффективности и сосредоточились на получении простой правильной программы, которую легко сопровождать.

Мы реализуем двоичный поиск в виде последовательности из четырех программ. Они довольно сложны, но есть хороший повод, чтобы их внимательно изучить: итоговая программа работает обычно в 2 – 3 раза быстрее, чем программа двоичного поиска, описанная в разд. 4.2. Можете ли вы перед тем, как продолжить чтение, обнаружить очевидные недостатки в этой программе?

```
L := 1; U := N
```

```
loop
```

```
/* Инвариант: если T есть в X, то T находится в X[L...U] */
```

```
if L > U then
```

```
    P := Ø; break
```

```
M := (L+U) div 2
```

```
case
```

```
    X[M] < T: L := M+1
```

```
    X[M] = T: P := M; break
```

```
    X[M] > T: U := M-1
```

Нашу разработку программы быстрого двоичного поиска мы начнем с модификации задачи локализации первого местонахождения целого числа  $T$  в массиве целых чисел  $X [1 \dots N]$ . Вышеприведенная программа может выдать на выходе любое из нескольких местонахождений элемента  $T$ . Основной цикл нашей программы аналогичен циклу в вышеприведенной программе; мы будем следить за индексами массива  $L$  и  $U$ . Позиция элемента  $T$  находится между этими индексами. Но в данном случае мы будем использовать следующее условие инвариантности:  $X [L] < T$ ,  $X [U] \geq T$  и  $L < U$ . Будем полагать, что  $X [0] < T$  и  $X [N + 1] \geq T$ , но программа никогда не будет обращаться к этим элементам. Вот эта программа:

```

L := 0; U := N+1
while L+1  $\neq$  U do
    /* Инвариант: X[L] < T и X[U]  $\geq$  T и L < U */
    M := (L+U) div 2
    if X[M] < T then
        L := M
    else
        U := M
    /* Утверждение: L+1 = U и X[L] < T и X[U]  $\geq$  T */
P := U
if P > N or X[P]  $\neq$  T then P := 0

```

Первый оператор устанавливает начальное значение инварианта. При повторении цикла инвариант сохраняется с помощью оператора `if`. Легко проверить, что операторы перехода сохраняют инвариант. При завершении мы знаем, что если элемент  $T$  находится в массиве, то его первое местонахождение – в позиции  $U$ ; этот факт определен более строго в комментарии, содержащем утверждение. В двух последних операторах в  $P$  засылается индекс первого местонахождения элемента  $T$  в массиве  $X$ , если  $T$  есть в  $X$ , или 0, если  $T$  нет в  $X$ . (В последнем операторе должна использоваться конструкция "условное или", которая не оценивает второе выражение, если первое выражение истинно; см. задачу 2 гл. 10.)

Хотя эта программа двоичного поиска решает более сложную задачу, чем предыдущая программа, она потенциально более эффективна: при

каждом проходе по циклу в ней выполняется только одно сравнение элемента  $T$  с элементами массива  $X$ . В предыдущей программе иногда требовалось делать две такие проверки.

В следующем варианте программы используется другое представление области: вместо представления  $L \dots U$  с помощью нижнего и верхнего значений мы будем задавать ее нижним значением  $L$  и приращением  $I$ , так что  $L + I = U$ . Программа будет работать так, чтобы  $I$  всегда было степенью числа 2. Это состояние легко поддерживать, если оно уже установлено, но его тяжело получить в первый раз (так как массив имеет размерность  $N = 1000$ ). Поэтому, чтобы гарантировать, что область, в которой ведется поиск, имеет исходную размерность, равную 512 (максимальная степень числа 2, меньшая 1000), самой программе предшествуют оператор присваивания и оператор условного перехода. Таким образом,  $L$  и  $L + I$  имеют значения либо  $0, \dots, 512$ , либо  $489, \dots, 1001$ . Преобразование предыдущей программы для нового представления области приводит к следующим командам:

```
I := 512
if X[512] >= T then
    L := 0
else
    L := 1000+1-512
while I ≠ 1 do
    /* Инвариант: X[L] < T и X[L+I] >= T и I = 2**j */
    NextI := I div 2
    if X[L+NextI] < T then
        L := L + NextI; I := NextI
    else
        I := NextI
/* Утверждение: I = 1 и X[L] < T и X[L+I] >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

Ход доказательства правильности этой программы в точности совпадает с доказательством для предыдущей программы. Эта программа работает

обычно медленнее своей предшественницы, но она открывает возможности для дальнейшего ускорения.

Следующая программа является упрощенным вариантом вышеприведенной. Она включает в себя некоторые возможности оптимизации, которые мог бы осуществить "разумный" компилятор. Упрощен первый оператор if, устранена переменная NextI, и из внутреннего оператора if удалены операторы присваивания с использованием переменной NextI.

```
L := 0
if X[512] < T then L := 1000+1-512
    /* Утверждение: X[L] < T и X[L+512] >= T */
if X[L+256] < T then L := L+256
    /* Утверждение: X[L] < T и X[L+256] >= T */
if X[L+128] < T then L := L+128
if X[L+64] < T then L := L+64
if X[L+32] < T then L := L+32
if X[L+16] < T then L := L+16
if X[L+8] < T then L := L+8
if X[L+4] < T then L := L+4
if X[L+2] < T then L := L+2
    /* Утверждение: X[L] < T и X[L+2] >= T */
if X[L+1] < T then L := L+1
    /* Утверждение: X[L] < T и X[L+1] >= T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0
```

Хотя доказательство правильности этой программы имеет ту же структуру, что и выше, теперь мы можем понять ее работу на более интуитивном уровне. Если первая проверка не выполнена и L осталось равным нулю, программа вычисляет P побитно, слева направо, причем старший бит вычисляется первым.

Окончательный вариант этой программы не для слабонервных. В нем посредством развертывания цикла устранены накладные расходы, связанные с организацией цикла и делением I на 2. Другими словами, так как I принимает в этой конкретной программе только несколько



различных значений, мы все их можем явно выписать, избежав тем самым вычисления их снова и снова во время выполнения программы.

```

I := 512; L := 0
if X[512] < T then L := 1000+1-512
while I ≠ 1 do
    /* Инвариант: X[L] < T и X[L+1] ≥ T и I = 2**j */
    I := I div 2
    if X[L+1] < T then
        L := L + I
/* Утверждение: I = 1 и X[L] < T и X[L+1] ≥ T */
P := L+1
if P > 1000 or X[P] ≠ T then P := 0

```

В этой программе можно разобраться, если вставить строки с утверждениями подобно тем, которые окружают оператор проверки  $X[L + 256]$ . Если вы, чтобы увидеть, как "ведет себя" оператор if, выполнили анализ для двух случаев, то вам понятно, что все остальные операторы if "ведут себя" аналогично.

Я сравнил программу двоичного поиска из разд. 4.2 с этой хорошо оптимизированной программой на ряде вычислительных систем и получил следующие результаты:

ЭВМ	Язык	Оптимизация	Единицы	Медленная программа	Быстрая программа	Ускорение
MIX	Ассемблер	Максимальная	Время	18,0	4,0	4,5
TRS-80	Бейсик	Нет	Мс	43,6	14,6	3,0
PDP-10 (KL)	Паскаль	"	Мкс	16,4	5,5	3,0
	Ассемблер	Максимальная		4,5	0,9	5,0
VAX-11/750	Си	Нет	Мкс	33,8	13,3	2,5
		Исходный текст		22,5	12,2	1,8
		Компилятор		29,2	12,8	2,3
		Текст + компилятор		19,7	12,2	1,6

Первая строка показывает, что программа из разд. 4.2, реализованная Кнудом на языке ассемблера, выполняется примерно за  $18 \log_2 N$  циклов компьютера MIX, тогда как его реализация быстроработающей програм-

мы из этого раздела в 4.5 раз быстрее. Коэффициент ускорения зависит от многих факторов, но во всех вышеприведенных случаях он значителен.

Этот пример является идеализированным рассказом об оптимизации программы в ее экстремальном виде. Мы заменили очевидный вариант программы двоичного поиска (он не производил впечатление слишком "объемного") на "короткую" версию, которая в несколько раз быстрее<sup>1</sup>. Средства верификации программ, описанные в гл. 4, играли в этой задаче решающую роль. Так как мы ими воспользовались, можем считать, что окончательный вариант программы верен. Когда я впервые увидел окончательный текст, представленный без проверки, я смотрел на него в течение месяца как на чудо.

#### 8.4. ОСНОВНЫЕ ПРИНЦИПЫ

Самым важным принципом касающимся оптимизации программ, является то, что оптимизацией нельзя злоупотреблять. Это радикальное общее правило объясняется следующим.

*Роль эффективности.* Многие другие характеристики программного обеспечения важны так же, как эффективность, если не больше. По наблюдениям Д. Кнута, непродуманная оптимизация является причиной многих несчастий; при этом может подвергаться опасности корректность, работоспособность и удобство сопровождения программы.

Оставьте заботы об эффективности для тех случаев, когда это действительно необходимо.

*Профилирование.* Когда эффективность важна, первый шаг – профилировать систему, чтобы обнаружить, где теряется время. В решении задачи 10 приведены два результата профилирования. Такие данные обычно показывают, что большая часть времени уходит на небольшое количество "узких мест", а остальная часть программы почти никогда не выполняется (например, в разд. 5.1 на одну процедуру приходится 98 % времени работы). Профилирование укажет критические области, а для других частей программы мы последуем мудрому правилу: "Не укрепляй то, что и так не ломается".

*Этапы разработки.* Как мы видели в гл. 5, есть много путей для эффективного решения задачи. Перед тем, как оптимизировать коман-

---

<sup>1</sup> Эта программа была известна среди программистов с начала 60-х годов. Отдельные фрагменты ее истории приведены на с. 93 колонки "Жемчужины программирования" в февральском номере журнала Communications of the ACM за 1984 г.

ды программы, мы должны убедиться, что другие подходы не дают более эффективного решения.

Ранее мы обсуждали, надо ли вообще и в каких случаях оптимизировать программу. Если мы решили так поступить, остается вопрос – как это делать? Я попытался ответить на этот вопрос в своей книге “Написание эффективных программ” (Writing Efficient Programs), содержащей список общих правил по оптимизации программ. Все рассмотренные нами примеры можно объяснить в терминах этих принципов, что я и сделаю, выделяя названия правил курсивом.

Программа построения изображений Ван Уайка. Основной стратегией в решении Ван Уайка был *учет особенностей наиболее распространенных случаев*. В его конкретном примере этот учет включал кэширование списка записей наиболее часто встречающегося типа.

Задача 1 – классификация символов. Решение с использованием таблицы, индексами в которой являются коды символов, – это *предварительное вычисление логической функции*.

Задача 2 – подсчет битов. Таблица со счетчиками числа битов, установленных в единицу в байте, близко связана с предыдущим решением. Это *хранение предварительно вычисленных результатов*.

Задача 3 – вычисление расстояний на сфере. Запоминание декартовых координат вместе с широтами и долготами – это пример *расширения структуры данных*. Использование более просто вычисляемого евклидова расстояния вместо углового базируется на алгебраической идентичности.

Двоичный поиск. *Объединение проверок выполнения условий* сокращает число сравнений элементов массива на внутреннем цикле с двух до одного, *использование алгебраической идентичности* позволяет заменить представление области с помощью верхней и нижней границ на представление с помощью нижней границы и приращения, а *развертывание цикла* увеличивает объем программы, устраняя все издержки на организацию цикла.

До сих пор мы оптимизировали программы, чтобы сократить процессорное время. Но можно заниматься оптимизацией программы и для других целей, таких как сокращение числа страниц памяти или увеличение объема кэш-памяти. Кроме сокращения времени работы, оптимизация, по всей вероятности, чаще всего используется для уменьшения объема памяти, требуемой программе. Первое представление об этом стремлении дает задача 4, а следующая глава целиком посвящена этой теме.

## 8.5. ЗАДАЧИ

1. В программе идентификации символов, приведенной в данной главе, предполагалось, что классы символов не пересекались. Как бы вы написали подпрограмму для проверки принадлежности в пересекающихся классах, таких как строчные буквы, прописные буквы, цифры и алфавитно-цифровые символы?
2. Напишите фрагмент программы, которая по заданному  $N$  (степени числа 2) устанавливает начальное значение в массиве `CountTable`  $[0 \dots N - 1]$ , как это описано в данной главе.
3. Как себя "будут вести" различные алгоритмы двоичного поиска, если их применить (вопреки спецификации) к неупорядоченным массивам?
4. На заре программирования Ф. Брукс столкнулся с проблемой представления большой таблицы на компьютере с небольшим объемом памяти. Он не мог хранить всю таблицу, так как места хватало только на то, чтобы представить каждый элемент в виде нескольких битов (фактически на каждый элемент отводилась только одна десятичная цифра – я ведь сказал, что это было очень давно!). Альтернативный подход заключался в использовании численного анализа, чтобы подобрать функцию, соответствующую этой таблице. В результате значения функции были весьма близки к табличным (ни одно значение не отличалось более чем на пару единиц от истинного). На ее реализацию потребовался крайне малый объем памяти, но в силу реальных ограничений эта аппроксимация не оказалась достаточно хорошей. Как мог бы Брукс получить требуемую точность при ограниченной памяти?
5. Типичный пример последовательного поиска для определения того, есть ли элемент  $T$  в массиве  $X[1 \dots N]$ , был приведен в задаче 9 гл. 4.

$I := 1$

**while**  $I \leq N$  and  $X[I] \neq T$  **do**  $I := I+1$

При общепринятом способе оптимизации программы ее работу ускорят, помещая элемент  $T$  на "сторожевую позицию" в конец массива.

$X[N+1] := T$

$I := 1$

**while**  $X[I] \neq T$  **do**  $I := I+1$

Исключение проверки  $I \leq N$  обычно уменьшает время работы программы на 20 – 30 %. Реализуйте эти две программы и измерьте время их работы в своей системе.

6. Как можно использовать способ, описанный в задаче 5, в программе поиска максимального элемента в массиве (см. задачу 9 гл. 4)? Как этот метод может уменьшить время поиска в наборах данных, представленных связанными списками, таблицами для случайного перемешивания и двоичными деревьями?
7. Так как программа последовательного поиска проще программы двоичного поиска, она обычно эффективней для небольших таблиц. С другой стороны, благодаря логарифмической зависимости числа сравнений, которые выполняет алгоритм двоичного поиска, он будет работать быстрее для больших таблиц, чем алгоритм последовательного поиска, выполняющий линейное число сравнений. Точка перелома зависит от того, насколько хорошо оптимизирована каждая из программ. Какие минимальное и максимальное значения вы можете получить для этой точки? Какое значение вы получите на компьютере, если обе программы одинаково оптимизированы? Является ли она функцией уровня оптимизации?
8. По наблюдениям Д. Б. Ломета из Уотсоновского исследовательского центра фирмы IBM, методом случайного перемешивания можно решить проблему поиска для 1000 целых чисел более эффективно, чем методом оптимизированного двоичного поиска. Напишите быстроработавшую программу на базе хэширования и сравните ее с оптимизированным двоичным поиском с точки зрения скорости и объема памяти.
9. В начале 60-х годов В. Береж из фирмы United Technologies Corporation обнаружил, что большая часть времени работы программ моделирования в фирме Sikorsky Aircraft уходит на вычисление тригонометрических функций. Дальнейшие исследования показали, что эти функции вычисляются только для углов, кратных 5. Как он уменьшил время работы?
10. Используйте средства профилирования, чтобы собрать статистические данные о рабочих характеристиках программы.
11. Иногда оптимизируют не команды программы, а используемые в ней математические методы. Чтобы вычислить полином
 
$$y = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0,$$
 в приведенной ниже программе выполняется  $2N$  умножений. Придумайте программу, работающую быстрее.

```

Y := A[0]; XToTheI := 1
for I := 1 to N do
    XToTheI := X*XToTheI
    Y := Y + A[I]*XToTheI

```

12. Примените описанные в этой главе методы к реальной программе, например к программе сортировки из разд. 1.4 или к программе для анаграмм из разд. 2.8.

#### 8.6. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Хотя я могу показаться нескромным, но моя любимая книга по оптимизации программ – это собственная книга "Написание эффективных программ" (Writing Efficient Programs), опубликованная в 1982 г. издательством Prentice-Hall. Ядром этой книги является обсуждение на 50 страницах упомянутых выше правил по достижению эффективности. Каждое правило формулируется в общем виде, а затем иллюстрируется примерами его применения для небольшого фрагмента программы и историями о его использовании в реальных системах. В других разделах книги рассматривается роль эффективности в системах программного обеспечения и использование этих правил в нескольких важных подпрограммах.

#### 8.7. ОПТИМИЗАЦИЯ ПРОГРАММ НА КОБОЛЕ, ИСПОЛЪЗУЕМЫХ ФЕДЕРАЛЬНЫМ ПРАВИТЕЛЬСТВОМ (ДОПОЛНЕНИЕ)

Роль оптимизации программ при обработке данных рассматривается в отчете "Улучшение прикладных систем на Коболе может выявить существенные ресурсы компьютеров" (Improving COBOL Applications Can Recover Significant Computer Resources (1 April 1982, Order Code PB82-198540, National Technical Information Service, Springfield, Virginia 22161). Эти прикладные системы разработаны министерством жилищно-го строительства и развития городов.

Сокращение процессорного времени, %	Экономия за год, дол.	Стоимость оптимизации
82	37 000	5500
45	45 000	1200
30	4400	2400
19	9000	900
9	7000	9000

Суммарная стоимость оптимизации – 19 000 дол. За минимальный (пятилетний) срок существования этих систем экономия составит около 400 000 дол. При эксплуатации программы на ЭВМ, принадлежащих вооруженным силам, один чел.-д, затраченный на оптимизацию, сокращает в типичном случае время прохождения программы с 7.5 до 2 ч и

более; такое изменение экономит 3500 дол за первый год работы и приводит к тому, что повышается надежность выдачи пользователю выходной информации.

В отчете содержится предупреждение, что оптимизация программы не должна выполняться наудачу; другим соображениям, таким как корректность и удобство сопровождения, должен быть отдан принадлежащий им по праву высокий приоритет. В докладе обращено внимание на то, что обычно существует предел оптимизации как для отдельной программы, так и для набора программ в системе: оптимизация вне этого предела может оказаться очень трудной и давать незначительный эффект.

В этом отчете также рекомендуется следующее: "Руководители федеральных агентств должны требовать периодической ревизии использования ресурсов своих ЭВМ прикладными программами на Коболе и, где это возможно, требовать действий по сокращению затрат для неблагоприятных в этом отношении прикладных программ".

## Глава 9. СОКРАЩЕНИЕ ОБЪЕМА ПАМЯТИ

Если вы похожи на некоторых моих знакомых, то первое, что придет вам в голову после прочтения заголовка этой главы будет: "Старо". Как гласит предание, в давние, плохие для программирования, времена программисты были скованы маломощностью компьютеров, но эти времена давно прошли. Теперь господствует новая философия: "Мегабайт здесь, мегабайт там – вот реальная память". И это верно вот почему: многие программисты используют большие ЭВМ и практически не беспокоятся о сокращении памяти для своих программ.

Но и упорные размышления о компактных программах могут принести пользу. Иногда в таких размышлениях рождается новый взгляд на программу, что делает ее проще<sup>1</sup>. Сокращение объема памяти дает иногда полезные побочные эффекты по времени работы: программы меньшего объема быстрее загружаются, а меньший объем обрабатываемых данных означает обычно меньшее время на их обработку. Даже при дешевой памяти ее объем может быть критичен. Многие микропроцессоры

---

<sup>1</sup> В своей статье, помещенной в июльском номере журнала Communications of the ACM за 1974 г., об операционной системе UNIX, разработанной на малых ЭВМ, Ричи и Томпсон отметили что всегда существовали довольно жесткие ограничения на систему и ее программное обеспечение. Если имеются частично антагонистические желания получить приемлемую эффективность и впечатляющие возможности, ограничения в размерах побуждают не только к экономии, но и к некоторой элегантности разработки.

имеют адресное пространство 64 Кбайта. Ограничение объема памяти побуждает не только к экономии, но и к некоторой элегантности разработки. Небрежное использование виртуальной памяти на больших ЭВМ может привести к катастрофически медленной работе.

Мы определили степень важности этой задачи, теперь рассмотрим некоторые методы сокращения объема памяти.

### 9.1. ПРОСТОТА – ЭТО КЛЮЧ К УСПЕХУ

Простота программы может способствовать функциональности, ясности, быстродействию и сокращению объема памяти программы. Ф. Брукс обнаружил это, когда в середине 50-х годов писал программу составления платежной ведомости для национальной компании. "Узким местом" программы было представление подоходного налога шт. Кентукки. Налог определялся по закону с помощью очевидной двумерной таблицы (доход – одна размерность, величина, не облагаемая налогом – другая). Хранение этой таблицы в явном виде требовало тысячи слов памяти, что превышало емкость ОЗУ ЭВМ.

Первым подходом который попытался применить Брукс, была попытка подобрать математическую функцию, аппроксимирующую таблицу, но таблица была настолько нерегулярной, что ни одна из простых функций не подходила. Зная, что таблица составлена людьми, которые не имели пристрастия к головоломным математическим формулам, он проконсультировался в течение нескольких минут с законодателями из шт. Кентукки, чтобы выяснить, какие аргументы привели к такой причудливой таблице, и обнаружил, что налог в шт. Кентукки был простой функцией дохода, который оставался после вычитания федерального налога. Поэтому его программа вычисляла федеральный налог по существующим таблицам, а потом использовала оставшийся доход и таблицу из нескольких дюжин слов памяти для того, чтобы найти налог шт. Кентукки.

После изучения контекста Бруксу удалось заменить исходную задачу более простой. Хотя казалось, что в первоначальной задаче для данных потребуются тысячи слов памяти, модифицированная задача была решена с использованием незначительного объема памяти.

Упрощения могут сократить и объем памяти, занимаемой кодами программы. В гл. 3 описываются несколько больших программ, которые были заменены меньшими программами с более подходящими структурами данных. В этих случаях более простой взгляд на программу сократил исходный текст с тысяч до сотен строк. При этом, вероятно, на порядок уменьшился и объем объектного кода.



## 9.2. ПАМЯТЬ ДЛЯ ХРАНЕНИЯ ДАННЫХ

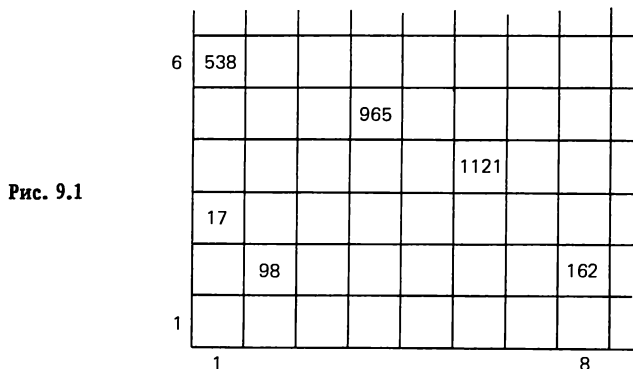
Очевидно, что упрощение – самый простой способ решить задачу, но некоторые трудные задачи не удастся упростить. В этом разделе мы рассмотрим методы, сокращающие объем памяти, требуемый для хранения данных, к которым программа осуществляет доступ. В следующем разделе мы рассмотрим сокращение объема памяти для хранения программы во время исполнения.

*Не храните данные, вычисляйте их повторно.* Память, требуемая для хранения данных о заданном объекте, может быть резко сокращена, если не запоминать их, а вычислять повторно всякий раз, когда они потребуются. На заре вычислительной техники некоторые программы хранили большие таблицы, например для функции SIN. Сегодня фактически все программы вычисляют тригонометрические функции, обращаясь к подпрограммам. Это сильно сокращает требования к памяти и вследствие успехов в численном анализе и в аппаратной реализации операций с плавающей точкой лишь немного дороже интерполяции на основе большой таблицы. Аналогично таблицу простых чисел можно было бы заменить подпрограммой проверки того, является ли заданное число простым. При использовании этого метода затрачивается больше времени за счет сокращения памяти, и он применим только тогда, когда данные об объектах, которые надо хранить, могут быть повторно вычислены по их описаниям.

Для таких целей, как сравнение эффективности или регрессионная проверка корректности, при выполнении различных программ с идентичными случайными входными данными часто используются "программы-генераторы". В зависимости от прикладной задачи случайный объект может быть файлом из сгенерированных случайным образом строк текста или графом со случайно сгенерированными ребрами. Вместо того, чтобы запоминать данные обо всем объекте, мы храним для его формирования программу и некоторое начальное число, которое определяет конкретный объект. Объект, имеющий объем несколько мегабайтов, можно задать с помощью нескольких байтов, если затратить для доступа к нему чуть больше времени.

*Разреженные структуры данных.* Изменение структуры данных может резко уменьшить объем памяти, необходимый для хранения заданной информации. Однажды я столкнулся с системой, которая позволяла пользователю осуществлять доступ к любой из 2000 точек на карте с помощью планшета для ввода. Эта программа преобразовывала физические координаты выбранной точки в пару целых чисел X (в пределах от 1 до 200) и Y (в пределах от 1 до 150). Планшет имел размеры примерно 4 × 3 фута, а разрешающая способность программы была 1/4 дюйма. Затем эта пара (X, Y) использовалась в программе чтобы сообщить, какую

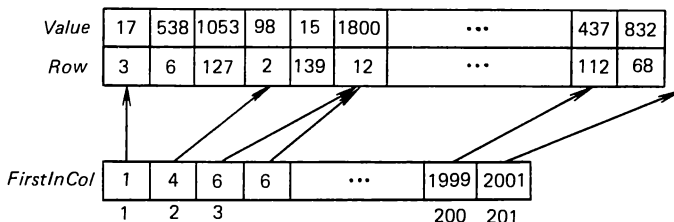
из двух тысяч точек выбрал (если вообще выбрал) пользователь. Так как никакие две точки не должны попадать в одну и ту же позицию (X,Y), программист представил карту массивом  $200 \times 150$ , содержащим идентификаторы точек (целые числа от 1 до 2000 или 0, если в этой позиции точки нет). Левый нижний угол этого массива мог бы выглядеть так (нулевые индексы точек представлены пустыми квадратами):



В соответствующей карте точка 17 имеет позицию (1, 3), точка 538 – позицию (1, 6), остальные четыре видимые позиции в первом столбце пусты.

Хотя этот массив легко реализовать и обеспечить малое время доступа, его объем  $200 \times 300 = 30\,000$  16-разрядных слов составляет более 10 % памяти компьютера с полумегабайтным ОЗУ. Когда система стала выходить за пределы отведенной ей памяти, программист понадеялся, что можно будет уменьшить объем памяти, занимаемый описанной структурой данных. Что бы вы предложили?

Я опишу наше решение в терминах языка Фортран, поскольку мы использовали именно этот язык. Если вы применяете язык с большими возможностями структурирования данных, потратьте минуту, чтобы подумать, как выразить на нем это решение. Далее показаны три массива, используемые в нашем решении. Ссылки по индексам, содержащимся в нижнем массиве, изображены стрелками:



Точки в столбце I соответствуют позициям между элементом FirstInCol[I] и элементом FirstInCol[I+1] – 1 в массивах Row и Value. Хотя в массивах только 200 столбцов, чтобы обеспечить выполнение указанного условия, определен также элемент FirstInCol[201]. На вышеприведенном рисунке в первом столбце три точки: точка 17 в позиции (1, 3), точка 538 в позиции (1, 6) и точка 1053 в позиции (1, 127). В столбце 2 – две точки, нет точек в столбце 3 и две точки в столбце 200. Чтобы определить, какая точка запомнена в позиции (I, J), мы использовали программу имеющую на псевдоязыке следующий вид:

```

for K := FirstInCol[I] to FirstInCol[I+1]-1 do
    if Row[K] = J then
        /* Точка найдена в позиции K */
        return Value[K]
/* Не найдено; (I,J) -пуст */
return 0

```

Для реализации этого метода используется намного меньший объем памяти, чем для его предшественника: два массива по 2000 элементов и один массив состоящий из 201 элемента, дают в сумме 4201 16-разрядных слов вместо 30 000. И хотя этот метод менее быстродействующий, чем предыдущий (в самом худшем случае для доступа требуется 150 сравнений, но в среднем используется полдюжины сравнений), у пользователя нет проблем по сопровождению такой программы. Ввиду хорошей модульной структуры этой системы новый вариант был включен в нее за несколько часов с помощью замены нескольких подпрограмм. Мы не заметили увеличения времени работы и высвободили 50 крайне необходимых килобайтов.

Такое решение иллюстрирует несколько общих моментов, касающихся структуры данных. Классическая задача: представление разреженных массивов (разреженный массив – это такой, в котором большинство элементов имеет одинаковое значение, обычно – нуль). Это решение концептуально просто и легко реализуемо при использовании одних только массивов. Отметим, что отсутствует массив LastInCol (последний в столбце), аналогичный массиву FirstInCol (первый в столбце). Вместо этого мы воспользуемся тем фактом, что последняя точка любого столбца расположена перед первой точкой следующего. Это тривиальный пример повторного вычисления вместо хранения. Точно также нет массива Col (столбец), аналогичного массиву Row (строка). Так как мы осу-

ществляем доступ к массиву Row только через массив FirstInCol, то всегда знаем номер текущего столбца.

Объем памяти можно сократить и многими другими методами структуризации данных. В разд. 3.1 мы сэкономили память, запоминая "рваную" трехмерную таблицу в виде двухмерного массива. Если мы используем ключ, который надо хранить в памяти, в качестве номера элемента массива, то нет необходимости хранить сам ключ, можно запомнить только относящийся к нему атрибут, например счетчик числа его повторений. Применение метода индексирования по ключу было описано в разд. 1.4 и 8.2 и в задачах 7 и 8 гл. 1. В вышеприведенном примере с разреженной матрицей использование ключа в качестве индекса в массиве FirstInCol позволило нам решить задачу без массива Col. Запоминание указателей, выделяющих фрагменты больших объектов (таких как длинные текстовые строки), устраняет затраты, связанные с хранением многих копий одного и того же объекта, хотя необходимо быть внимательным, модифицируя объект, принадлежащий многим пользователям. Этот метод применен в моем настольном календаре (с 1821 по 2080 гг.). Вместо того, чтобы иметь 260 различных календарей, используется 14 канонических календарей (с учетом високосного года) и приводится таблица, в которой дан номер календаря для каждого из 260 годов.

*Сжатие данных.* Идеи из теории информации позволяют сократить память с помощью компактного кодирования данных. В примере с разреженной матрицей мы предположили, что элементы массива Row — 16-разрядные целые числа. Так как номер строки находится в диапазоне от 1 до 150, элементы этого массива можно было бы представить 8-битовыми байтами, что сэкономило бы 1 Кбайт памяти. В системе для деловых контактов на базе микрокомпьютера я кодировал две десятичные цифры одним байтом (вместо очевидных двух), записывая целое число  $N = 10 \times A + B$ . Информация декодировалась так:

$$A := N \text{ div } 10$$

$$B := N \text{ mod } 10$$

Эта простая схема сжимала объем файла с цифровыми данными до размера одной дискеты вместо двух<sup>1</sup>. Такое кодирование может уменьшить

---

<sup>1</sup> Некоторые читатели предложили использовать такое кодирование:  $N := (A \text{ lshift } 4) \text{ or } B$  (lshift — сдвиг влево). Значения могут быть декодированы операторами  $A := \text{rshift } 4$  (rshift — сдвиг вправо) и  $B := N \text{ and } 1111_2$ . Дж. Линдерман отметил, что операция сдвига и наложения маски обычно не только осуществляется быстрее, чем умножение и деление, но стандартные служебные подпрограммы, такие, как вывод на печать в шестнадцатеричном виде, отображают закодированные таким образом данные в легко воспринимаемом виде.

объем памяти, требуемый для каждой записи, но на обработку этих коротких записей обычно затрачивается больше времени, так как сначала их надо декодировать.

Используя теорию информации, можно также сжать поток записей, передаваемых по каналу, например по линии связи или в файл на диске. Эти методы сжатия информации обычно требуют сложной реализации, но могут привести к существенной экономии: в гл. 13 вкратце описано, как файл из 30 000 английских слов был помещен в 26 000 16-разрядных слов компьютера. Подробности этих методов можно найти в справочной литературе.

*Стратегия распределения памяти.* Иногда важно не то, какой объем памяти используется, а то, как он используется. Предположим, например, что в вашей программе есть три типа записей X, Y и Z — все одинакового размера. При написании программы на некоторых из языков первым побуждением могло быть задание по сотни объектов каждого из этих трех типов. Что бы произошло, если бы вы использовали 101 запись типа X и не было бы записи типа Y и Z? Программа вышла бы за пределы отведенной ей памяти, хотя 200 ячеек памяти остались бы совершенно неиспользованными. Динамическое распределение записей устранило бы такой явный недостаток путем выделения записей по мере необходимости. Многие современные языки располагают таким средством, но даже в примитивных языках, таких как Фортран, программист может реализовать эту стратегию на уровне программы пользователя.

Динамическое распределение означает, что мы должны запрашивать что-либо, пока нам это не потребуется. Стратегия записей переменной длины означает, что, запрашивая что-либо, мы должны потребовать ровно столько, сколько нам необходимо. Во времена восьмидесятисимвольных записей обычным было то, что половина байтов на диске с библиотекой программ приходилась на дополняющие эти записи справа пробелы. В файлах с записями переменной длины конец строки обозначается символом *новая строка*, и вследствие этого емкость таких дисков увеличивается. Я однажды утроил скорость выполнения программы на микрокомпьютере, используя записи переменной длины на магнитной ленте: максимальная длина записи была 250, но в среднем использовалось только около 80 байтов.

Более совершенные методы распределения памяти описаны в справочниках. Сборка мусора переупорядочивает освободившуюся память так, что использованные однажды фрагменты становятся такими же пригодными, как и новые. Алгоритм пирамидальной сортировки в разд. 12.4 осуществляет перекрытие по памяти двух логических структур данных, используемых в разное время. Другой подход к совместному

использованию памяти применил Б. Керниган, когда написал программу для решения "задачи о коммивояжере", в которой большая часть памяти была занята двумя матрицами  $N \times N$ , где  $N = 150$ . В этих двух матрицах, которые я назову А и В, содержались расстояния между пунктами. Поэтому Керниган знал, что диагональные элементы равны нулю ( $A[I, I] = 0$ ) и они симметричны ( $A[I, J] = A[J, I]$ ), и поделил между двумя треугольными матрицами объем памяти, отведенный под одну квадратную матрицу. Фрагмент получившейся матрицы выглядит так:

Рис. 9.3

0	B[1, 2]	B[1, 3]	B[1, 4]
A[2, 1]	0	B[2, 3]	B[2, 4]
A[3, 1]	A[3, 2]	0	B[3, 4]
A[4, 1]	A[4, 2]	A[4, 3]	0

Теперь Керниган мог обращаться к элементу  $A[I, J]$  с помощью следующего оператора:

$$C[\max(I, J), \min(J, I)]$$

и аналогично к элементам матрицы В, но переставив  $\min$  и  $\max$ . Такое представление используется в различных программах очень давно. Эта методика несколько усложнила Кернигану написание программы и слегка замедлила ее, но сокращение двух матриц по 22 500 слов до одной было существенным для компьютера, имеющего 30 000 слов памяти. А для матриц размерностью  $900 \times 900$  это изменение дало бы тот же эффект на компьютере с ОЗУ емкостью 4 Мбайта

### 9.3. СОКРАЩЕНИЕ КОЛИЧЕСТВА КОМАНД

Иногда "узким местом" (в смысле объема памяти) в программе являются не данные, а размер самой программы. Например, я подписался на журнал для любителей, который публикует программы для построения графических изображений, содержащие следующие команды:

```

for I := 17 to 43 do Set(I,68)
for I := 18 to 42 do Set(I,69)
for J := 81 to 91 do Set(30,J)
for J := 82 to 92 do Set(31,J)

```

где функция Set (X, Y) высвечивает элемент изображения на экране в позиции (X, Y). Соответствующие подпрограммы, скажем Hor и Vert, для рисования горизонтальных и вертикальных линий позволили бы заметить эти команды на команды

```

Hor(17,43,68)
Hor(18,42,69)
Vert(81,91,30)
Vert(82,92,31)

```

Эту программу, в свою очередь, можно заменить интерпретатором, который читает команды из текстового файла, подобного следующему:

```

H 17 43 68
H 18 42 69
V 81 91 30
V 82 92 31

```

Если это занимает все еще слишком много места, то каждую из строк можно представить 32-разрядным словом, в котором 2 бита выделены на команды и по 10 битов на каждое из трех целых чисел в диапазоне от 0 до 1023. (Такое преобразование может быть выполнено программой.) Этот гипотетический случай иллюстрирует несколько общих методов сокращения места, занимаемого командами программы.

*Формирование подпрограмм.* Замена прямого задания команд на подпрограммы упростила вышеприведенную программу, поэтому сократила объем необходимой для нее памяти и сделала ее более понятной. Это тривиальный пример разработки программы "снизу вверх". Хотя нельзя игнорировать многие достоинства методов проектирования "сверху вниз", "гомогенный взгляд на мир", который дают хорошие подпрограммы нижнего уровня, может упростить сопровождение системы и одновременно сократить память.

Сократить занимаемую командами память можно, уменьшив число подпрограмм. В разд. 9.6 и 9.7 даны ссылки по этой теме.

*Интерпретаторы.* В вышеприведенном гипотетическом случае мы смогли заменить длинную строку текста программы командой из четырех байтов для специального интерпретатора. В разд. 3.2 описан интерпретатор для формирования писем; хотя его основная цель – упростить написание и сопровождение программы, он дает также побочный эффект сокращения объема памяти, занимаемого программой.

*Трансляция на машинный язык.* Одним из аспектов сокращения памяти, которым большинство программистов может управлять слабо, является трансляция с исходного языка на машинный. Например, некоторые незначительные изменения компилятора уменьшают в ранних версиях системы UNIX память, занимаемую командами, на 5 %. В крайнем случае программист может рассмотреть вариант трансляции большой по объему программы в текст на ассемблере вручную, но это дорогой, нередко приводящий к ошибкам способ, который обычно не приносит хорошие плоды. В неопубликованных результатах эксперимента Л. Р. Картер (работавший в то время в фирме Motorola Software Technology) обнаружил, что запись программы из 1500 строк на Паскале в виде 2900 строк на ассемблере сократила размер объектного модуля с 1600 до 1200 байтов. Эти 4 Кбайта в данном приложении были очень важны (для работы системы требовалось еще несколько Кбайт сверх 128 Кбайт, имевшихся на процессорной плате, а на кросс-плате не было места для установки дополнительной платы памяти), но на экономию каждых десяти байтов расходовался 1 ч, т. е. затраты составляли примерно 5 дол за байт.

#### 9.4. ОСНОВНЫЕ ПРИНЦИПЫ

Теперь, после того как мы рассмотрели методы сокращения требуемого объема памяти, давайте определим позицию, которую мы должны занять, приступая к этой задаче.

*Расплата за увеличение объема памяти.* Что произойдет, если программа будет использовать на 10 % больший объем памяти? Во многих системах за такое увеличение не приходится платить: пропадавшим зря битам найдется теперь хорошее применение. Возможно, в малых системах программа вообще не сможет работать: она выйдет за пределы памяти. В больших системах с разделением времени расплата не увеличивалась бы пропорционально увеличению объема памяти. В системах с виртуальной памятью могло бы резко возрасти время работы, так как программа, которая раньше помещалась в физической памяти, стала теперь многократно обращаться к диску – в задаче 4 гл. 2 описывается,



как увеличение на несколько процентов размерности задачи увеличило время работы программы на два порядка. Перед тем, как вы решите сокращать требуемый объем памяти, необходимо знать, как вы будете за это расплачиваться.

*“Узкие места” памяти.* В разд. 8.4 описано, что время работы программы обычно резко зависит от “узких мест”: на несколько команд выпадает большая часть времени работы программы. Для памяти, которую занимает программа, верно обратное: выполняется команда миллиард раз или вообще не выполняется, для ее хранения требуется одинаковый объем памяти. Память для данных может иметь “узкие места”: несколько простых записей могут занимать большую часть памяти. Например, в случае с разреженной матрицей на единственную структуру данных приходилось более 10 % памяти ЭВМ с ОЗУ емкостью 0.5 Мбайта. Замена матрицы структурой, имеющей размер, в десятки раз меньший, оказало существенное влияние на систему; сокращение структуры данных объемом 1 Кбайт в 100 раз имело бы меньшее влияние.

*Компромиссы.* Иногда программисту приходится пренебрегать эффективностью, функциональностью, простотой сопровождения, чтобы сэкономить память. Такие технические решения должны приниматься только после изучения всех альтернатив. На ряде примеров в этой главе показано, как сокращение объема памяти может иногда улучшить другие характеристики. В разд. 1.4 побитовая структура данных позволила запомнить набор записей в ОЗУ, а не на диске и тем самым сократила время выполнения программы с нескольких минут до нескольких секунд, а программу – с сотен до нескольких дюжин строк. Это произошло только потому, что первоначальное решение было не оптимальным, но, поскольку все мы как программисты еще далеки до совершенства, в наших собственных программах часто можно найти точно такие же места. Мы должны признать этот факт и, прежде чем начать жертвовать каким-либо из полезных свойств программы, искать методы, улучшающие все другие аспекты нашего решения.

*Работа со средой.* Среда, окружающая программу, может иметь существенное влияние на эффективность программы (с точки зрения памяти). Существенным является то, какие представления используются компилятором и исполнительной системой, а также стратегия распределения памяти и страничной организации. Когда вы почти вышли за объем ОЗУ, убедитесь, что указанные аспекты вам известны настолько, чтобы гарантировать, что вы не идете наперекор принципам, принятым в системе.

*Выбор подходящих для данной задачи средств сокращения объема памяти.* Мы рассмотрели четыре метода сокращения объема данных (повторное вычисление, разреженные структуры, теория информации и стратегия распределения памяти), три метода уменьшения количества

команд (формирование подпрограмм, интерпретаторы и трансляция) и один основополагающий принцип – простота. Когда нужно сэкономить память, рассмотрите все эти варианты.

#### 9.5. ЗАДАЧИ

1. В конце 70-х годов С. Фельдман создал компилятор языка Фортран 77, который едва умещался в 64 Кбайта. Чтобы сократить память, он упаковал элементы записей нескольких типов в четырехбитовые поля. Когда он распаковывал эти записи, чтобы хранить поля в байтах, то он обнаружил, что, хотя память для данных увеличилась на несколько сотен байтов, общий объем программы уменьшился на несколько тысяч байтов. Что произошло?
2. Есть ли другие простые, но эффективные (с точки зрения памяти) структуры данных для задачи с разреженной матрицей? Как бы вы написали программу для построения структуры данных, описанной в тексте? Как бы вы изменили структуру так, чтобы, не используя слишком много дополнительной памяти, ускорить поиск? Насколько еще вы можете сократить память при необходимости?
3. В качестве примеров экономии "памяти" рассмотрите хранение информации в прикладных задачах, не связанных с компьютерами, таких как календари или математические таблицы.
4. Программа на Бейсике в журнале для любителей содержит операторы DATA, которые определяют последовательность примерно из 4000 одnobайтовых целых чисел. Первые несколько чисел в этой таблице следующие:  
128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128,  
128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128,  
128, 128, 128, 128, 128, 128, 152, 166, 172, 153, 164, 128, 128, 128, 128, 128,  
128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, 128, . . .  
В этой программе числа распечатываются на четырех страницах в две колонки. Приведите более подходящее представление этих чисел.
5. При обсуждении "сжатия данных" в разд. 9.2 было упомянуто декодирование  $10 \times A + B$  с помощью операций `div` и `mod`. Рассмотрите компромисс "память-время" при замене этих операций поиском в таблице.
6. Разработайте структуру связанного списка, имеющего только один указатель на каждый узел, но позволяющую, тем не менее, перемещаться по списку вперед и назад.
7. В обычных программах при профилировании значения счетчика команд выбираются на регулярной основе (см., например, решение к задаче 10 гл. 8). Разработайте структуру данных для хранения таких

значений, эффективную с точки зрения времени и памяти, а также выдающую полезную выходную информацию.

8. Изображение лица (отменно красивого) из разд. 3.3 хранится в матрице  $48 \times 48$  байтов. Предположим, вы должны помнить много изображений лиц со сходными характеристиками; насколько вы могли бы уменьшить требования к объему памяти?
9. Программисты, работающие на Коболе, обычно отводят 6 байтов на дату (ДДММГГ), 9 байтов на номер в системе социального страхования (ДДД-ДД-ДДД) и 21 байт для имени и фамилии (12 – для фамилии, 8 – для имени и 1 – для инициала). Насколько вы могли бы сократить эти ограничения, если бы объем памяти был критичен?
10. [Ф. Аппел] Уплотните оперативно используемый словарь английских слов до минимально возможного размера. Подсчитывая объем памяти, учитывайте и файл данных, и программу, интерпретирующую эти данные.

#### 9.6. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Методы сокращения объема памяти для данных рассматриваются во многих учебниках по структурам данных. Книга "Методы структур данных" Стэндиша (Standish. Data Structures Techniques), опубликованная издательством Addison-Wesley в 1980 г. является хорошим подспорьем при изучении этой области. К этой теме относятся разд. 2.5 (управление несколькими стеками), 5.4 (восстановление памяти), 5.5 (уплотнение и сосуществование), 6.2 (динамическое распределение памяти), 7.2 (представление строк), 7.4 (представление строк переменной длины), 8.3 – 8.5 (представление массивов).

Глава 9 книги Ф. Брукса "Mythical Man Month", выпущенной издательством Addison-Wesley в 1975 г.<sup>1</sup>, озаглавлена "Десять фунтов в пятифунтовом мешке". Основное внимание в ней уделено вопросам распределения памяти в больших системах со стороны руководителей проекта. Автор поднимает такие важные вопросы, как ресурсы памяти, спецификация функций модулей и выделение памяти в зависимости от времени выполнения.

#### 9.7. ДВА ПРИМЕРА БОЛЬШОГО СОКРАЩЕНИЯ ПАМЯТИ (ДОПОЛНЕНИЕ)

В основной части этой главы дан обзор различных методов. Давайте теперь внимательно рассмотрим две критичные с точки зрения объема памяти системы, чтобы увидеть, как действуют эти методы.

---

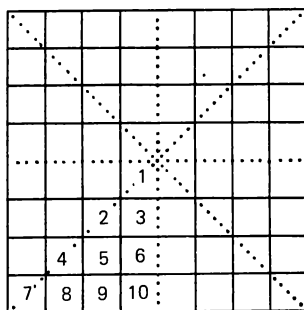
<sup>1</sup> Имеется перевод: "Как проектируются и создаются программные комплексы": Пер. с англ. – М: Наука, 1979. – Прим. перев.

К. Томпсон разработал двухэтапную программу шахматного эндшпиля<sup>1</sup> для заданных наборов фигур, например король и два слона против короля и коня. На этапе обучения программа вычисляет число ходов до матовых позиций для всех возможных расстановок фигур (т.е. для заданного набора из четырех или пяти фигур), осуществляет анализ в обратном направлении от всех возможных матовых позиций. Специалисты по компьютерам называют этот метод динамическим программированием, в то время как знатоки шахмат называют его ретроградным анализом. Получившаяся в результате база данных делает программу "всеведущей" для заданных фигур, поэтому на этапе игры она в совершенстве играет эндшпили. Ее игру знатоки шахмат называют "сложной, изменчивой, сильной, трудной" и "мучительно медленной и загадочной", она опрокидывает устоявшиеся шахматные догмы.

Запоминанию в явном виде всех возможных положений фигур препятствовали чрезмерные затраты памяти. Поэтому Томпсон использовал кодирование расположенных фигур как ключ для доступа к записям файла на диске, содержащего информацию об этих позициях. Каждая запись файла содержит 12 битов, включая число ходов до мата из этой позиции. Так как на доске 64 клетки, все позиции для пяти фигур можно закодировать пятью целыми числами в диапазоне от 0 до 63, что дает расположение каждой фигуры. Результирующий ключ из пяти битов подразумевает наличие таблицы из  $2^{30}$  или примерно 1.07 млрд 12-битовых записей в базе данных, что превышает емкость диска.

Главная находка Томпсона заключалась в том, что варианты расположения шахматных фигур, являющиеся зеркальными отображениями относительно штриховых линий на следующем рисунке, равноценны и нет необходимости дублировать их в базе данных.

Рис. 9.4



Поэтому в его программе предполагается, что белый король может находиться в одном из десяти пронумерованных полей; произвольное распо-

<sup>1</sup> Эта программа отличается от программы Belle chess machine, разработанной Дж. Кондоном и Томпсоном.

ложение фигур может быть представлено в этом виде последовательно-стью не более трех отображений. Такая нормализация сокращает файл на диске до  $10 \times 64^4$  или  $10 \times 2^{24}$  12-битовых записей. В дальнейшем Томпсон отметил, что так как черный король не может находиться в смежных полях с белым королем, то для двух королей имеется только 454 правильные позиции, в которых белый король расположен в одном из десяти помеченных на вышеприведенном рисунке полей. В результате использования этого факта его база данных сократилась до  $454 \times 64^3$ , т. е. примерно до 121 млн 12-битовых записей, что соответствовало (с запасом емкости) одному (специально выделенному) диску.

В качестве второй критичной по объему требуемой памяти системы рассмотрим компьютер Macintosh фирмы Apple. Подробности об этом компьютере и историю его разработки можно найти в февральском выпуске журнала BYTE за 1984 г. Дефицит памяти в этой системе возник в результате того, что все огромные функциональные возможности системы требовалось втиснуть в постоянное запоминающее устройство (ПЗУ) емкостью 64 Кбайта. Бригада разработчиков достигла этого тщательным формированием подпрограмм (включая обобщение операторов, слияние программ и т. п.) и кодированием вручную всего ПЗУ на ассемблере. Они оценили, что их оптимизированная до предела программа с продуманным распределением регистров и выбором команд в 2 раза меньше по размеру эквивалентной программы реализованной на языке высокого уровня.

Эти две системы во многом различаются. Задача бригады разработчиков системы для компьютера Macintosh – сжать программу размером в 128 Кбайт до объема 64 Кбайт – была вызвана тем, что они рассчитывали поставлять миллионы копий системы. Хотя Томпсон знал, что его программа будет существовать только в одном экземпляре, он должен был сократить файл с учетом емкости диска. Бригада разработчиков оптимизировала программу на ассемблере, чтобы сократить объем ПЗУ; Томпсон воспользовался фактом симметрии в структуре данных, чтобы сократить объем файла на диске.

Однако эти два достижения имеют некоторые общие черты. В обоих случаях объем памяти уменьшился ненамного (в 2 раза для ПЗУ компьютера Macintosh и в 8 раз для диска, содержащего варианты эндшпилей), что было определяющим для успеха всей системы. Сокращение памяти уменьшило также время работы обеих систем. Компактная программа на ассемблере для компьютера Macintosh может осуществлять ввод-вывод со скоростью 1 Мбит/с, а сокращение числа позиций в программе для эндшпилей уменьшает время ее обучения с одного года до нескольких недель.

## ЧАСТЬ III

### ПРОГРАММНЫЙ ПРОДУКТ

Настало время развлечений. В частях I и II была заложена основа; в следующих четырех главах этот материал мы используем для создания интересных и важных программ, которые показывают основные моменты приложения методов, изложенных в предыдущих главах, к реальным задачам.

В гл. 10, 11 и 12 описаны подпрограммы общего назначения для сортировки, поиска и обслуживания очередей с приоритетами. В гл. 13 рассказано о программе проверки правописания английских слов, в которой используются некоторые из указанных алгоритмов. Главы 11 и 13 основаны на частных задачах, а гл. 10 и 12 – на методах, являющихся базовыми для приведенных в них программ.

Глава 10 появилась впервые в апрельском номере журнала *Communications of the ACM* за 1984 г., гл. 11 – в декабре 1984 г., гл. 12 – в марте 1985 г., а гл. 13 – в мае 1985 г.

#### Глава 10. СОРТИРОВКА

Как бы вы отсортировали последовательность записей в заданном порядке? Ответ обычно прост:

*Воспользуйтесь командой сортировки, имеющийся в операционной системе.*

К несчастью, это не всегда срабатывает. В некоторых системах нет команды сортировки, а существующие стандартные средства сортировки могут оказаться недостаточно общими или недостаточно эффективными для того, чтобы решить конкретную задачу (как в разд. 1.1). В таких случаях у программиста нет другого выхода, как писать программу сортировки.

### 10.1. СОРТИРОВКА МЕТОДОМ ВСТАВОК. АЛГОРИТМ ПОРЯДКА $O(N^2)$

Сортировка методом вставок – это метод, применяемый большинством карточных игроков для сортировки своих карт. Они держат сданные карты в отсортированном порядке и, когда приходит новая карта, помещают ее в соответствующее место. Чтобы отсортировать массив в возрастающем порядке, мы начнем с отсортированного подмассива  $X[1 \dots 1]$ , а затем будем вставлять элементы  $X[2], \dots, X[N]$ , как это показано в следующей программе на псевдоязыке.

```
for I := 2 to N do
    /* Инвариант: сортируется массив X[1...I-1] */
    /* Цель: сдвинуть X[I] вниз на соответствующее
        место в X[1...I-1] */
```

В следующих четырех строках продемонстрирована обработка этим алгоритмом массива из четырех элементов. Переменной  $I$  соответствует символ  $*$ ; элементы слева от нее отсортированы, в то время как элементы справа содержатся в исходном порядке.

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

Чтобы отслеживать сдвигаемые элементы, сдвиг выполняется справа налево в цикле, в котором используется переменная  $J$ . В этом цикле элемент меняется местами со своим предшественником в массиве до тех пор, пока есть предшественник (т. е.  $J > 1$ ) и элемент не достиг своего окончательного места (т. е. он размещен неверно относительно своего предшественника). Таким образом, вся программа сортировки выглядит так:

```
for I := 2 to N do
    /* Инвариант: сортируется массив X[1...I-1] */
    J := I
    while J > 1 and X[J-1] > X[J] do
        Swap(X[J], X[J-1])
    /* Swap меняет элементы местами */
    J := J-1
```

Когда мне надо выполнить сортировку и эффективность не является определяющей, я использую эту программу; она состоит всего из пяти строк команд. (В некоторых чересчур защищенных от ошибок языках при выполнении этой программы может возникнуть ошибка; см. задачу 2.)

Если у вас под рукой нет подпрограммы Swap, можно воспользоваться следующими операторами присваивания, в которых для перестановки элементов  $X[J]$  и  $X[J-1]$  используется переменная  $T$ :

```
T := X[J];  X[J] := X[J-1];  X[J-1] := T
```

Эта программа побуждает заняться простыми упражнениями по ее оптимизации в духе гл. 8. Так как в переменную  $T$  постоянно засылается одно и то же значение (которое было в начальный момент в ячейке  $X[I]$ ), мы можем вынести из цикла операторы пересылки в переменную  $T$  и пересылки из нее и изменить оператор сравнения, как показано ниже.

```
for I := 2 to N do
  /* Инвариант: сортируется массив X[1...I-1] */
  J := I
  T := X[J]
  while J > 1 and X[J-1] > T do
    X[J] := X[J-1]
    J := J-1
  X[J] := T
```

В этой программе элементы сдвигаются вправо на свободное место, которое образуется в результате сохранения значения  $X[I]$ , и в заключение переменная  $T$  пересылается на такое свободное место, которое является ее окончательной позицией. В ней семь строк, и она более сложная, чем простой алгоритм сортировки методом вставок, но в моей системе она выполняется в 3 раза быстрее первой программы.

Данную программу легко преобразовывать в текст даже на примитивных языках, таких как Бейсик:

```
1000 ' СОРТИРОВКА X(1..N), N>1
1010 FOR I=2 TO N
1015   ' ИНВАРИАНТ: СОРТИРУЕТСЯ МАССИВ X(1...I-1)
```



```

1020 J=I
1030 T=X(J)
1040 IF J<=1 OR X(J-1)<=T THEN 1080
1050 X(J)=X(J-1)
1060 J=J-1
1070 GOTO 1040
1080 X(J)=T
1090 NEXT I
1100 RETURN

```

Сравнив время работы этой программы с "эффективной" сортировкой из учебника "1982 BASIC" (где "трусливая" логика потребовала вдвое большего числа строк), я обнаружил, что эта простая программа требует вдвое меньше времени для своей работы, чем ее более сложная предшественница.

При случайных данных и в худшем случае время сортировки методом вставок массива из  $N$  элементов пропорционально  $N^2$ . К счастью, если массив уже почти упорядочен, эта программа работает намного быстрее, так как каждый элемент сдвигается вниз только на короткое расстояние.

## 10.2. БЫСТРАЯ СОРТИРОВКА – АЛГОРИТМ ПОРЯДКА $O(N \log N)$

Этот алгоритм был описан К. А. Р. Хором в его классической статье "Быстрая сортировка" (C. A. R. Hoare. Quicksort) в апрельском номере журнала Computer Journal 5, 1 за 1962 г. (с. 10 – 15). Он использовал принцип "разделяй и властвуй" из разд. 7.3: чтобы отсортировать массив, мы делим его на две части и сортируем их рекурсивно. Например, чтобы отсортировать восьмиэлементный массив

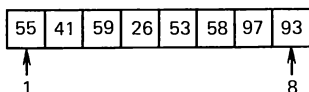


Рис. 10.1

мы разбиваем его по первому элементу (55) так, чтобы все элементы, меньшие 55, были слева от него, а большие – справа:

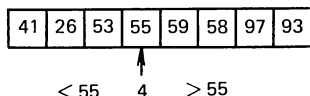


Рис. 10.2

Если мы теперь рекурсивно отсортируем подмассив с элементами 1 – 3 и подмассив с элементами 5 – 8 независимо друг от друга, то весь массив окажется отсортированным.

Среднее время работы этого алгоритма намного меньше, чем время сортировки методом вставок, т. е.  $O(N^2)$ , так как операция разбиения сильно превосходит по эффективности сортировку последовательности. После типичной операции разбиения  $N$  элементов примерно половина элементов превышает значение, относительно которого выполнено разбиение, а другая половина – меньше его. Примерно за такое же время операция сдвига при методе вставок "ухитрится" поставить очередной элемент на нужное место.

Описанная выше идея приводит к наброску рекурсивной подпрограммы. Мы будем представлять часть массива, с которой имеем дело, двумя индексами  $L$  и  $U$ , обозначая ими нижнюю и верхнюю границы. Рекурсия заканчивается, когда мы получаем массив менее чем из двух элементов. Таким образом, программа выглядит так:

```
procedure QSort(L,U)
```

```
  if L >= U then
```

```
    /* Если в массиве не более одного элемента, делать нечего */  
  else
```

```
    /* Делим массив согласно заданному значению,
```

```
       которое, возможно, расположено
```

```
       в правильном месте P */
```

```
    QSort(L, P-1)
```

```
    QSort(P+1, U)
```

Чтобы выполнить разбиение массива относительно элемента  $T$ , воспользуемся простой схемой, которую я узнал от Н. Ломута из компании Alsys, Inc. Для этой работы есть более быстроработающие программы<sup>1</sup>, но в данной процедуре легко разобраться, тяжело сделать ошибку

---

<sup>1</sup> Большинство людей, описывающих алгоритм быстрой сортировки, применяют схему разбиения, основанную на использовании двух индексов, подобных описанным в задаче 3. Хотя основная идея такой схемы проста, я обнаружил, что в деталях она сложна — однажды я потратил добрых два дня, отслеживая ошибку, спрятавшуюся в коротком цикле разбиения. Человек, прочитавший черновик, утверждал, что стандартная схема разбиения с использованием двух индексов проще, чем метод Ломута, и набросал программу в подтверждение своей точки зрения. Я прекратил рассматривать ее после того, как он обнаружил две ошибки.

и в конце концов она работает ненамного медленней. Дано значение  $T$ ; мы должны переупорядочить массив  $X[A \dots B]$  и вычислить индекс  $M$  (от слова middle – середина) так, чтобы все элементы, меньшие  $T$ , были по одну сторону от  $M$ , а все остальные элементы – по другую сторону. Мы выполним эту работу с помощью простого цикла `for`, в котором массив сканируется слева направо, при этом используются переменные  $I$  и  $M$ , чтобы обеспечить выполнение следующего инварианта в массиве  $X$ :

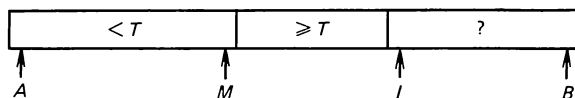


Рис. 10.3

Когда программа проверяет  $I$ -й элемент, требуется рассмотреть два случая. Если  $X[I] \geq T$ , все прекрасно – условие инвариантности выполняется. С другой стороны, когда  $X[I] < T$ , мы можем достичь инвариантности, увеличив индекс  $M$  так, чтобы он стал индексом меньшего элемента, а затем поменять местами этот элемент и элемент  $X[I]$ . Полностью программа разбиения выглядит так:

```

M := A-1
for I := A to B do
  if X[I] < T then
    M := M+1
    Swap (X[M], X[I])
  
```

В программе быстрой сортировки мы будем разбивать массив  $X[L \dots U]$  относительно значения  $T = X[L]$ , поэтому  $A$  будет равно  $L + 1$ , а  $B$  будет равно  $U$ . Таким образом инвариант цикла разбиения выглядит так:

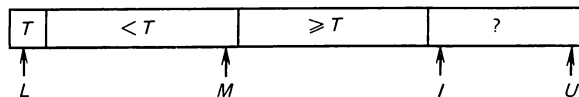


Рис. 10.4

По окончании цикла мы имеем:

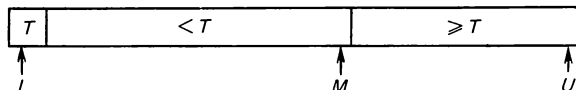
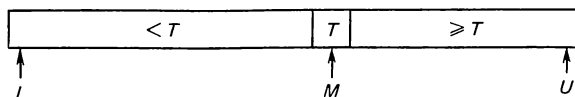


Рис. 10.5

После этого мы меняем местами элементы  $X[L]$  и  $X[M]$ , что позволяет получить<sup>1</sup>

Рис. 10.6



Теперь мы можем рекурсивно вызывать эту процедуру с параметрами  $(L, M - 1)$  и  $(M + 1, U)$ .

Вышеприведенный алгоритм всегда осуществляет разбиение относительно первого элемента массива. Такой выбор может потребовать чрезмерных времени и памяти для некоторых типичных входных данных, например для уже отсортированных массивов. Мы поступим гораздо проще, выбрав элемент для разбиения случайным образом. Для осуществления этого поменяем местами элемент  $X[L]$  и произвольный элемент из массива  $X[L \dots U]$ . Если у вас нет функции `RandInt`, которая используется в нижеприведенной программе, вы можете реализовать ее, воспользовавшись функцией `Rand`, которая осуществляет выбор случайного действительного числа с равномерным распределением плотности вероятности на интервале  $[0,1)$ , и выражением  $L + \text{int}(\text{Rand} \times (U + 1 - L))$ . В том невероятном случае, когда в вашей системе нет даже такой подпрограммы, обратитесь к книге Кнута "Получисленные алгоритмы". Но независимо от того, используете вы системную подпрограмму или напишете собственную, обратите внимание на то, чтобы функция `RandInt` возвращала число в диапазоне от  $L$  до  $U$  – значение вне этого диапазона приводит к роковой ошибке.

Окончательная программа "Быстрая сортировка 1", представлена ниже. Чтобы отсортировать массив  $X[1 \dots N]$ , мы вызываем процедуру

```
QSort(1,N)
```

```
procedure Qsort(L,U)
```

```
  if L < U then
```

```
    Swap(X[L], X[RandInt(L,U)])
```

<sup>1</sup> Заманчиво проигнорировать этот шаг и организовать рекурсию с параметрами  $(L, M)$  и  $(M + 1, U)$ ; но это приводит к бесконечному циклу, если элемент  $T$  строго больше всех других элементов в подмассиве. Попытавшись проверить условие завершения, я мог бы обнаружить ошибку, но тогда проницательные читатели догадались бы, как я в действительности сделал это открытие. М. Джакоб нашла элегантное доказательство некорректности этой рекурсии: так как элемент  $X[L]$  никогда не перемещается, сортировка может быть правильной, только в том случае, если минимальный элемент массива находится вначале в ячейке  $X[1]$ .

```

T := X[L]
M := L
for I := L+1 to U do
    /* Инвариант: X[L+1...M] < T и
       X[M+1...I-1] >= T */
    if X[I] < T then
        M := M+1
        Swap(X[M], X[I])
Swap(X[L], X[M])
/* X[L...M-1] < X[M] <= X[M+1...U] */
QSort(L, M-1)
QSort(M+1, U)

```

Большая часть доказательства корректности этой программы была дана при описании ее разработки (что является, конечно, самым подходящим для этого местом). Доказательство проводится по индукции: внешний оператор `if` правильно обрабатывает пустой массив и массив из одного элемента, а программа разбиения корректно задает массивы большего размера для рекурсивного вызова. Такая программа не может осуществить бесконечную последовательность рекурсивных вызовов, так как элемент  $X[M]$  исключается при каждом обращении; это доказательство аналогично доказательству, использованному в разд. 4.3 для демонстрации завершения двоичного поиска.

Давайте обратимся теперь к характеристикам эффективности этой программы. Я не хочу приводить здесь подробности, но время работы программы быстрой сортировки оценивается как  $O(N \log N)$ , а размер стека составляет в среднем  $O(\log N)$  для любого входного массива с различными элементами. Математические доказательства близки к приведенным в разд. 7.3, а в решении задачи 10 содержатся данные об одной из реализаций этого алгоритма. Полученные случайные значения характеристик являются результатом вызова генератора случайных чисел, а не предложением о случайном распределении входных данных. В задачах 3, 5 и 11 показаны пути повышения производительности программы быстрой сортировки для худшего случая. В большинстве учебников по алгоритмам приводится математический анализ программы быстрой сортировки,

а также доказывается, что минимальное количество сравнений для любой сортировки, основанной на сравнениях, составляет  $O(N \log N)$  сравнений, поэтому программа быстрой сортировки близка к оптимальной.

Любители методов, изложенных в гл. 8, вероятно, отметили несколько путей оптимизации программы быстрой сортировки для ускорения ее работы. Самый простой путь показан в решении задачи 10 гл. 8: мы должны явно выписать команды для процедуры *Swap* во внутреннем цикле (так как два остальных обращения к этой процедуре находятся вне внутреннего цикла, выписывание их в явном виде оказало бы пренебрежимо малое влияние на быстродействие). В моей системе этим способом время работы программы было сокращено до 2/3 по отношению к предыдущему времени выполнения. Можно заметить, что большая часть времени тратится на сортировку очень маленьких массивов. Было бы быстрее сортировать их, используя простой метод, подобный сортировке методом вставок, вместо того, чтобы использовать метод быстрой сортировки целиком.

Б. Седжуик разработал несколько более хитрую реализацию этой идеи. Когда быстрая сортировка вызывается для маленьких подмассивов (это происходит, если значения *U* и *L* близки), мы ничего не делаем. Для реализации этого приема мы изменяем первый оператор *if* в процедуре быстрой сортировки на следующий:

```
if U-L > CutOff then
```

где *CutOff* – небольшое целое число. Когда программа кончает работу, массив не отсортирован, но сгруппирован в маленькие группы случайно расположенных элементов так, что значения всех элементов в одной группе меньше, чем в любой другой группе справа от нее. Мы должны доупорядочить элементы внутри групп любым другим методом: так как весь массив почти отсортирован, метод сортировки вставками хорошо подходит для этой работы. Для сортировки всего массива напомним следующую программу:

```
qSort(1,N)
```

```
InsertionSort      /* сортировка методом вставок */
```

Для определения наилучшего значения переменной *CutOff* я прогнал программу дважды для всех значений *CutOff* от 1 до 50 при фиксированном  $N = 5000$ . На этом графике представлены результаты.

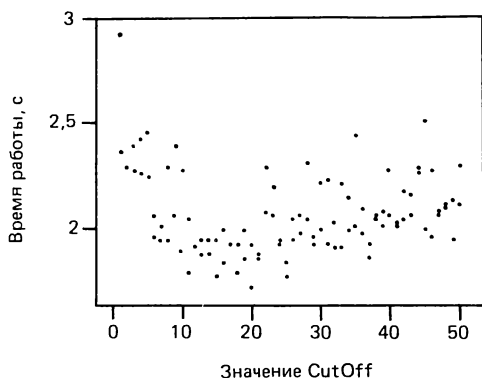


Рис. 10.7

Хорошим значением для переменной CutOff оказалось 15; значения от 10 до 20 дают экономию времени в пределах нескольких процентов по отношению к данному значению. Указанное изменение сократило время работы программы, которую мы назовем "Быстрая сортировка 2", вдвое по отношению к первоначальному времени работы, а по сравнению с программой с явно записанной процедурой Swar – на 25 %.

### 10.3. ОСНОВНЫЕ ПРИНЦИПЫ

Рассмотренные нами программы обобщены в приведенной ниже таблице. Они были реализованы на языке Си на ЭВМ VAX-11/750, измерения проводились для 32-разрядных целых чисел, логарифмы взяты по основанию 2. "Вставка 1" – первая из описанных программ, "Вставка 2" – та же программа с явно записанной процедурой Swar и с вынесенными из цикла операторами пересылки в переменную T и из нее. "Быстрая сортировка 1" – первая из описанных программ быстрой сортировки; в программе "Быстрая сортировка 2" – процедура Swar записана явно, а сортировка небольших массивов происходит обращением к программе "Вставка 2" после рекурсивного вызова ее с параметрами (1,N). "Системная сортировка" – это имеющаяся в системе UNIX подпрограмма под названием qsort. Функции для времени работы получены подгонкой известных нам формул по результатам измерений, приведенным в таблице.

Программа	Содержит строк на языке Си	Время работы, мкс	Время выполнения для данных объемом, с,		
			100	1000	10 000
Вставка 1	5	$17N^2$	0.17	17.3	1730
Вставка 2	7	$6N^2$	0.06	5.7	570
Быстрая сортировка 1	11	$63N \log_2 N$	0.05	0.63	7.8
Быстрая сортировка 2	20	$32N \log_2 N$	0.03	0.32	4.3
Системная сортировка	1	$97N \log_2 N$	0.06	1.0	13.6

Еще с одним видом сортировки порядка  $O(N \log N)$  мы познакомились в разд. 12.4.

Из этой таблицы можно извлечь несколько важных уроков, касающихся обоих методов сортировки в частности, так и программирования вообще..

Системная сортировка – простая и относительно быстрая; она медленней, чем самодельная программа быстрой сортировки только в силу своего универсального и гибкого интерфейса, в котором для каждого сравнения используется вызов процедуры. Если системная сортировка в состоянии удовлетворить ваши требования, даже не думайте о написании своей собственной программы. (В разд. 2.8 описаны два средства сортировки в системе UNIX: программа `sort` и подпрограмма `qsort`.)

Сортировку методом вставок легко запрограммировать, и для задач небольшой размерности она может быть достаточно быстроработающей. На сортировку 10 000 целых чисел программой "Вставка 2" в моей системе потребовалось ровно 10 мин. Для больших  $N$  определяющим является время работы программы быстрой сортировки –  $O(N \log N)$ . Методы разработки алгоритмов, изложенные в гл. 7, подсказали нам основные идеи для этого алгоритма, в котором применен принцип "разделяй и властвуй", а методы верификации программ из гл. 4 помогли реализовать эти идеи в виде простой, короткой и эффективной программы. Хотя большое увеличение скорости было достигнуто изменением алгоритма, методы оптимизации программы, описанные в гл. 8, увеличили быстродействие программы сортировки методом вставок в 3 раза, а программы быстрой сортировки – в 2 раза.

#### 10.4. ЗАДАЧИ

1. Подобно другим мощным средствам сортировка часто используется, когда этого не требуется, и не используется, когда это необходимо. Объясните, когда необходима и когда не нужна сортировка на примере вычисления следующих статистических данных для массива из  $N$  действительных чисел: минимум, максимум, среднее, медиана, мода.
2. Предположим, что массив  $X[1 \dots 10]$  и переменная  $T$  описаны как целые числа; что произойдет при исполнении следующей программы?

```
I := 11
```

```
if I <= 10 and X[I] < T then I:= I+1
```



Во многих языках эта программа выполняется элегантно, и переменная  $I$  остается неизменной. Однако в некоторых системах эта программа может быть завершена аварийно, так как индекс  $I$  выходит за границы массива. Что сделает ваша система? Почему данная проблема является важным моментом в различных программах сортировки методом вставок? Как можно решить эту проблему в таких программах?

3. Приведенная в тексте программа быстрой сортировки работает за время, пропорциональное  $N^2$ , если  $X[1] = X[2] = \dots = X[N]$ . Объясните, почему? Эта проблема устранена в задаче 11 и в приведенной ниже программе быстрой сортировки, адаптированной в соответствии со статьей Седжуика, которая упоминается в разд. 10.5.

```

procedure Qsort(L,U)
  if L < U then
    Swap(X[L], X[RandInt(L,U)])
    I := L; J := U+1; T := X[L]
    loop
      repeat I := I+1 until X[I] >= T
      repeat J := J+1 until X[J] <= T
      if J < I then break
      Swap(X[I], X[J])
    Swap(X[L], X[J])
    QSort(L, J-1)
    QSort(I, U)

```

В этой программе предполагается, что в массиве  $X$  нет элементов, больших  $X[N+1]$ , и используется "сигнальная метка"  $N+1$  для ускорения внутреннего цикла. Для массивов с различными элементами эта программа делает меньше перестановок, чем "Быстрая сортировка 1", и поэтому почти вдвое быстрее. Воспользуйтесь методами гл. 4, чтобы доказать, что эта программа верна. Как она будет решать задачу при совпадающих значениях элементов?

4. [Р. Седжуик] Увеличьте быстродействие метода Ломута, используя  $X[L]$  в качестве "сигнальной метки", аналогично тому, как это сделано в задаче 8.5. Покажите, как такая схема позволяет устранить процедуру Swap после цикла.

5. Хотя в программе быстрой сортировки для стека используется в среднем только  $O(\log N)$  элементов памяти, в худшем случае эта зависимость может стать линейной. Объясните, почему это происходит, и измените программу так, чтобы логарифмическая зависимость сохранилась для худшего случая.
6. [М. Д. Макилрой] Покажите, как использовать метод Ломута для сортировки строк битов переменной длины за время, пропорциональное сумме длин этих строк.
7. Реализуйте несколько программ сортировки и подытожьте их в таблице, аналогичной приведенной в тексте. В дополнение к сортировке методом вставок и быстрой сортировке вы, может быть, захотите рассмотреть сортировку методом Шелла (прекрасное быстродействие и простота программы), пирамидальную сортировку (минимальная дополнительная память и хорошее быстродействие для наихудшего случая – см. разд. 12.4) и поразрядную сортировку (применима только в особых случаях, см. решение задачи 6). Из вашей таблицы следуют такие же выводы?
8. Набросайте на одном листке процедуру, показывающую пользователю, работающему на вашей системе, как ему выбрать подпрограмму сортировки. Обеспечьте в вашем методе учет важности времени прогона, объема памяти, времени, затрачиваемого программистом на разработку и сопровождение, универсальность (вдруг я захочу сортировать строки символов, являющиеся римскими цифрами?), стабильность (элементы с одинаковыми ключами должны сохранять свой относительный порядок), особые свойства входных данных и т. д. В качестве последней проверки вашей процедуры попытайтесь подсунуть ей задачу сортировки, описанную в гл. 1.
9. Напишите программу нахождения  $k$ -го по величине элемента в массиве  $X[1 \dots N]$  за время  $O(N)$ . Ваш алгоритм может переставлять элементы в массиве  $X$ . Представьте данные о времени его работы.
10. Соберите и представьте эмпирические данные о времени работы программы быстрой сортировки.
11. Напишите подпрограмму разбиения со следующим условием на выходе:

Рис. 10.8

$< T$	$= T$	$> T$
-------	-------	-------

Как бы вы ввели эту подпрограмму в программу быстрой сортировки?

12. Рассмотрите методы сортировки, используемые в прикладных задачах, не связанных с применением компьютеров (таких как сортировка писем на почте и сортировка мелких денег).

13. В этой главе в программах быстрой сортировки элемент для разбиения выбирался случайным образом. Рассмотрите лучшие варианты, такие как выбор среднего значения по массиву.

## 10.5. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Что читать по методам сортировки, зависит от того, зачем вы читаете. Чтобы больше узнать о предмете вообще, обратитесь к учебникам по алгоритмам, которые упоминались в гл. 2 и 7. Следующие ссылки особенно полезны для программистов, пишущих программы сортировки:

Если вы хотите писать чрезвычайно эффективные программы сортировки в ОЗУ, обратитесь к статье Седжуика "Реализация программ быстрой сортировки" (Sedgewick. Implementing Quicksort Programs) в октябрьском номере журнала "Communications of the ACM" за 1978 г.

Если ваша работа состоит в написании простых программ для сортировки на диске, читайте гл. 4 в книгах Кернигана и Плodgeжа "Программные средства" или "Программные средства на Паскале".

Программисты, намеревающиеся потратить несколько месяцев на написание качественной системы, должны изучить том 3 "Сортировка и поиск" работы Кнута "Искусство программирования для ЭВМ". В статье Линдермана "Теория и практика разработки работающей подпрограммы сортировки" (Linderman. Theory and practice in the construction of a working sort routine. The Bell Laboratories Technical Journal 63, 8, ч. 2 с. 1827 – 1843) рассказано о том, как применить эти методы в реальной программе системной сортировки.

## Глава 11. ПОИСК

Небольшие программы для компьютеров часто бывают поучительными и занимательными. В этой главе рассказывается история одной крошечной программы, которая в дополнение к этим свойствам оказалась довольно полезной для небольшой фирмы.

### 11.1. ЗАДАЧА

Эта фирма только что закупила несколько персональных компьютеров. Запустив базовую систему, я стал агитировать персонал искать у себя в офисе задачи, которые можно было бы решать с помощью программ. Фирма занималась опросом общественного мнения, и одна находчивая сотрудница предложила автоматизировать задачу построения выборки, извлекая случайным образом избирательные участки из

их списка. Так как при выполнении этой работы вручную требуется провести нудный час с таблицей случайных чисел, она предложила написать следующую программу:

Мне бы хотелось получить программу, для которой пользователь вводит список названий избирательных участков и целое число  $M$ . Выходом программы является список из  $M$  избирательных участков, выбранных случайным образом. Обычно имеется несколько сотен названий избирательных участков (каждое название – алфавитно-цифровая строка, имеющая не более дюжины символов), а типичные значения  $M$  лежат между 20 и 40.

Эта была идея программы с точки зрения пользователя. Есть ли у вас какие-либо предложения по постановке задачи перед тем, как мы бросимся ее программировать?

Моей первой реакцией было: "Это отличная мысль; задача готова для автоматизации." Однако потом я отметил, что хотя ввести с клавиатуры нескольких сотен названий, вероятно, проще, чем иметь дело с длинными столбцами случайных чисел, задача остается все еще утомительной и уязвимой для ошибок. Вообще, глупо готовить уйму входных данных, если программа так или иначе игнорирует большую их часть. Поэтому я предложил следующую альтернативную программу:

Входные данные состоят из двух целых чисел  $M$  и  $N$ , где  $M < N$ . Выходом является упорядоченный список из  $M$  случайных целых чисел в диапазоне 1, . . .  $N$ , в котором ни одно число не присутствует более одного раза. Чтобы удовлетворить блюстителей теории вероятностей, мы потребуем, чтобы в отсортированной выборке не было замен и каждый элемент выбирался с равной вероятностью.

При  $M = 20$  и  $N = 200$  эта программа могла бы выдать последовательность из 20 чисел, которая начинается так: 4, 15, 17, . . . После этого пользователь составляет выборку 20 из 200 избирательных участков, последовательно просматривая список и отмечая четвертое, пятое, семнадцатое названия и т. д. (Выходные данные должны быть упорядочены, так как выданный на устройство печати список не пронумерован.)

Такая постановка задачи встретила одобрение потенциальных пользователей. После того, как программа была реализована, задачу, на решение которой раньше требовался час, удалось решить за несколько минут.

Теперь рассмотрим эту задачу с другой стороны: как бы вы реализовали эту программу? Предположим, что в вашей системе есть функция  $\text{RandInt}(I, J)$ , которая выдает на выходе случайное целое число, выбранное из диапазона  $I, \dots, J$  с равномерным распределением вероятнос-

тей, и функция RandReal (A, B), которая выдает на выходе случайное действительное число, выбранное на интервале [A, B) с равномерным распределением вероятностей.

## 11.2. ОДНО ИЗ РЕШЕНИЙ

Как только мы сформулировали задачу, я помчался за своим ближайшим экземпляром книги Кнута "Получисленные алгоритмы". Внимательно изучив эту книгу неделей раньше, я знал, что в ней содержится ряд алгоритмов для решения задач, подобных этой. Потратив минуту на рассмотрение нескольких возможных вариантов решения, которые мы вкратце разберем ниже, я понял, что алгоритм S из разд. 3.4.2 книги Кнута является идеальным решением данной задачи.

Этот алгоритм рассматривает по порядку целые числа 1, 2, ..., N и использует для выбора каждого из них проверку с соответствующим случайным условием. Поскольку числа просматриваются по порядку, гарантируется, что выход будет упорядоченным.

Чтобы понять критерий выбора, рассмотрим пример, где  $M = 2$  и  $N = 5$ . Мы должны выбрать число 1 с вероятностью  $2/5$ ; программа осуществляет это с помощью оператора, подобного следующему:

```
if RandReal(0,1) < 2/5 then ...
```

К несчастью мы не можем выбрать 2 с такой же вероятностью: если это сделать, в итоге можно получить 2 числа из 5, а можно и не получить. Поэтому мы должны изменить критерий и выбрать число 2 с вероятностью  $1/4$ , если число 1 было выбрано, или с вероятностью  $2/4$ , если число 1 не было выбрано. Вообще, чтобы выбрать S чисел из R оставшихся, мы будем выбирать следующее число с вероятностью  $S/R$ .

Эта идея приводит к программе 1

```
Select := M; Remaining := N
for I := 1 to N do
  if RandReal(0,1) < Select/Remaining then
    print I: Select := Select-1
  Remaining := Remaining-1
```

До тех пор, пока  $M < N$ , программа выбирает точно M целых чисел: она не может выбрать больше потому, что, когда переменная Select становится равной 0, числа перестают выбираться, и не может выбрать меньше, так как, когда отношение Select/Remaining становится равным 1,

число обязательно выбирается. Оператор `for` обеспечивает выдачу чисел на печать в упорядоченном виде. Вышеприведенное описание поможет вам понять, что каждое подмножество должно выбираться с равной вероятностью; Кнут дает доказательство этого с точки зрения теории вероятностей.

Руководствуясь вторым томом книги Кнута, я написал программу с легкостью. Включая заголовки, проверку границ и т. п., для окончательной программы потребовалось только 13 строк на Бейсике. Она была завершена через полчаса после постановки задачи и использовалась в течение ряда лет без каких-либо проблем.

### 11.3. ПРОСТРАНСТВО РЕШЕНИЙ

Одна часть работы программиста состоит в решении сегодняшних задач, другая и, возможно, более важная часть состоит в подготовке к решению задач завтрашних. Иногда такая подготовка включает прохождение курсов обучения или штудирования книг, подобных книгам Кнута. Гораздо чаще, однако, мы, программисты, учимся на простых умозрительных упражнениях или пытаемся ответить на вопрос, как решить задачу другим способом. Давайте сейчас исследуем пространство возможных решений для задачи построения выборки.

Когда я рассказывал об этой задаче в Уэст-Пойнте, я попросил придумать лучший подход, чем тот, что был в первой постановке задачи (ввод всех 200 названий в программу). Один слушатель предложил снять копию со списка избирательных участков, разрезать копию на бумажные полоски, перемешать полоски в бумажном пакете, а затем вытащить требуемое количество полосок. Он продемонстрировал "концептуальный прорыв", который является темой книги Адамса, упоминавшейся в разд. 1.7<sup>1</sup>.

В дальнейшем мы ограничимся в наших поисках программами, имеющими на выходе  $M$  упорядоченных целых чисел, выбранных случайным образом из последовательности  $1, \dots, N$ . Мы начнем с оценки программы 1. Ее алгоритмическая идея ясна, текст короток, она занимает всего несколько слов в памяти, а время ее работы прекрасно соответствует данной прикладной задаче. Однако время работы может оказаться проблемой в других приложениях: например, выбор дюжины целых чисел в диапазоне  $1 \dots 2^{31} - 1$  занял бы несколько часов на суперкомпьютере. Поэтому стоит затратить несколько минут нашего вре-

---

<sup>1</sup> В этой книге приведены взгляды Артура Кестлера на три вида творчества. *А-а!* Озарением он назвал оригинальность. *Ага!* Озарение — это открытие. Решение этого слушателя Уэст-Пойнта он назвал бы *ха-ха!* Озарение: ненаучный ответ на высоконучный вопрос — это акт комического вдохновения (как в решении задачи 10 гл. 1.)

мени на изучение других путей решения этой задачи. Перед тем, как продолжить чтение, сделайте столько набросков решения на самом верхнем уровне, сколько сможете; о деталях реализации пока не заботьтесь.

В одном из решений случайные целые числа заносятся в пустое первоначально множество до тех пор, пока их не наберется достаточное количество. На псевдоязыке это выглядит так:

```
Очистить массив S
Size := 0
/* Size - количество выбранных элементов */
while Size < M do
    T := RandInt(1,N)
    if T is not in S then
        /* Если T нет в S то */
        Insert T in S
        /* Занести T в S */
        Size := Size + 1
Напечатать элементы множества S в упорядоченном виде
```

Этот алгоритм не приводит к группировке вокруг одного из элементов; его выходные данные – случайные. Проблема реализации множества S у нас все еще остается в том же состоянии; подумайте о подходящей структуре данных.

Побитовую структуру данных, описанную в разд. 1.4, реализовать относительно просто. Мы представим множество S в виде массива битов, в котором I-й бит установлен в единицу тогда и только тогда, когда целое число I есть в наборе. Мы установим набор в исходное состояние с помощью подпрограммы InitToEmpty, которая сбрасывает все биты в нуль.

```
for I := 1 to N do
    Bit[I] := 0
```

Функция Member (T) сообщает, находится ли элемент T в множестве S, возвращая на выходе Bit [T], а процедура Insert (T) вставляет элемент T в множество S, присваивая Bit [T] := 1. И, наконец, подпрограмма PrintInOrder распечатывает элементы множества S:

```

for I := 1
    if Bit[I] = 1 then
        print I

```

Эти подпрограммы позволяют написать более точные псевдокоманды для программы 2.

```

InitToEmpty
Size := 0
/* Size - количество выбранных элементов */
while Size < M do
    T := RandInt(1,N)
    if not Member(T) then
        /* Если T нет в S то */
        Insert(T)
        /* Занести T в S */
        Size := Size + 1
PrintInOrder

```

Для побитового представления в Программе 2 используется  $N/b$  слов  $b$ -разрядной памяти. При очевидной реализации подпрограмм начальной очистки и печати время работы каждой из них пропорционально  $N$ , но оно может быть уменьшено до  $N/b$  при одновременной обработке всех  $b$  битов слова (это выполняется, пока  $M < N/b$ ; мы вскоре рассмотрим, что делать, когда  $M$  близко к  $b$ ). Процедура *Insert* всегда вызывает точно  $M$  раз, но число обращений к функции *Member* может быть большим, так как некоторые из получаемых с помощью функции *RandInt* целых чисел могут уже быть в наборе. В задаче 2 требуется показать, что до тех пор, пока  $M < N/2$ , ожидаемое количество проверок, выполняемых функцией *Member*, меньше  $2M$ . Как функция *Member*, так и процедура *Insert* требуют на каждую операцию постоянное время, так что общие затраты работы программы 2 составляет  $O(N/b)$ .

Хотя в этом анализе эффективности предполагается, что множество  $S$  было реализовано в виде побитового представления, в программе 2 это не отражено. В операциях *InitToEmpty*, *Member*, *Insert* и *PrintInOrder* обрабатывается множество с "абстрактным типом данных" – множество,



для которого определены такие операции, обычно называется словарем (более подробно об этом см. в разд. 12.5). Замена этих четырех подпрограмм может изменить представление множества и поэтому повлиять на эффективность программы. Приведенные ниже рисунки иллюстрируют несколько возможных структур данных в конце работы программы при  $M = 5, N = 10$  и функцией  $\text{RandInt}(1, 10)$  сформирована последовательность 3, 1, 4, 1, 5, 9.

1 0 1 1 1 0 0 0 1 0

3 1 4 5 9

1 3 4 5 9

4			
1	3	5	9

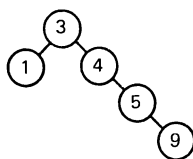


Рис. 11.1

Двоичные деревья описаны в большинстве учебников по алгоритмам и структурам данных. Так как вставка в дерево происходит в случайном порядке, довольно велика вероятность, что дерево будет далеко не сбалансированным; поэтому сложные схемы для балансировки в этой прикладной задаче не нужны. Совокупность  $M$  "карманов" можно рассматривать как разновидность хэширования, когда все целые числа в диапазоне  $1, \dots, N/M$  размещаются в первом "кармане", а целое число  $I$  направляется в "карман" с номером (округленно)  $I * M / N$ . Эти "карманы" реализованы как массив связанных списков. Так как целые числа распределены равномерно, ожидаемая длина каждого связанного списка равна единице. Средняя эффективность различных схем при  $M < N/b$  приведена в таблице.

Представление множества	O (время на операцию)				Общее время	Память в словах
	Init	Member	Insert	Print		
Битовый вектор	$N/b$	1	1	$N/b$	$O(N/b)$	$N/b$
Неупорядоченный массив	1	$M$	1	$M \log M$	$O(M^2)$	$M$
Упорядоченный массив	1	$\log M$	$M$	$M$	$O(M^2)$	$M$
Двоичное дерево	1	$\log M$	$\log M$	$M$	$O(M \log M)$	$3M$
Карманы	$M$	1	$M$	$M$	$O(M)$	$3M$

Бойтесь коэффициентов, скрытых за "О-большим": операции над массивами обычно "дешевле" по сравнению с некоторыми вариантами реализации доступа к битовому вектору, с операциями для работы с указателями в двоичном дереве, с операциями деления, используемыми при работе с "карманами". Чтобы разобраться в вопросах, связанных с эффективностью, давайте рассмотрим случай с  $N = 1\,000\,000$  и  $b = 32$ . При  $M = 5000$  "карманы" являются, вероятно, самой эффективной структурой; когда  $M = 50\,000$ , побитовое представление осуществляется быстрее и занимает меньше места; при  $M = 500\,000$  программа 1 использует намного меньше места и также работает быстрее. Однако при  $M = 999\,995$  лучше отметить пять невыбранных элементов. Любой вариант легко запрограммировать и получить быстроработающую программу.

Еще один подход к получению упорядоченного подмножества случайных целых чисел состоит в "перетасовке"  $N$ -элементного массива, в котором содержатся числа от 1 до  $N$ , а затем в упорядочении первых  $M$  элементов, которые и являются выходными данными. Алгоритм Кнута в разд. 3.4.2 перетасовывает массив  $X[1 \dots N]$ .

```
for I := 1 to N do
    Swap(X[I], X[RandInt(I,N)])
```

Э. Шеперд и А. Воронов из Хьюстонского университета отметили, что в этой программе нам требуется перетасовать только первые  $M$  элементов массива, что приводит к программе 3.

```
for I := 1 to N do X[I] := I
for I := 1 to M do
    Swap(X[I], X[RandInt(I,N)])
Sort(1, M)
```

Упорядоченный список находится в массиве  $X[1 \dots M]$ . В этом алгоритме используется  $N$  слов ОЗУ и время  $O(N + M \log M)$ , но методика, применяемая в задаче 8 гл. 1, сокращает это время до  $O(M \log M)$ . Этот алгоритм мы можем рассматривать как альтернативу программе 2, множеству выбранных элементов в нем соответствует массив  $X[1 \dots I]$ , а множеству невыбранных элементов – массив  $X[I + 1 \dots N]$ . Явно задавая невыбранные элементы, мы устраним проверку того, был ли очередной элемент выбран ранее.

В программах 1 – 3 предлагаются разные решения задачи, но они не покрывают все пространство возможных вариантов. Еще один метод состоит в генерации интервалов между следующими друг за другом элементами множества. В работе Дж. С. Виттера "Быстрые методы получения случайных выборок" (J. S. Vitter. Faster Methods for Random Sampling) в июльском номере журнала Communications of the ACM за 1984 г. генерируется последовательность из  $M$  упорядоченных случайных целых чисел за время  $O(M)$  и с использованием фиксированного объема памяти. Такие затраты ресурсов являются оптимальными (с точностью до коэффициента).

#### 11.4. ОСНОВНЫЕ ПРИНЦИПЫ

Эта глава иллюстрирует несколько важных этапов процесса программирования. Хотя далее эти этапы рассматриваются в естественном порядке, процесс разработки более сложен: мы перескакиваем с одного этапа на другой, выполняя каждый этап много раз, пока не достигнем приемлемого решения.

*Разберитесь в задаче.* Поговорите с пользователем о контексте, в котором возникла задача. В формулировке задачи часто содержатся и идеи ее решения; их (как и все скороспелые идеи) нужно рассмотреть, но не нужно им слепо следовать.

*Опишите абстрактную задачу.* Ясная, четкая постановка задачи помогает сначала решить эту задачу, а потом посмотреть, как полученное решение может быть применено к другим задачам.

*Исследуйте пространство возможных решений.* Слишком многие программисты быстро хватаются за одно конкретное решение задачи;

они думают минуту и целый день программируют вместо того, чтобы подумать час и в течение часа написать программу. Использование неформальных языков высокого уровня помогает описать решение: управляющая логика представляется на псевдоязыке, важнейшие структуры данных – с помощью "абстрактных типов данных". Неоцененное значение на этом этапе процесса разработки имеет знание литературы.

*Реализуйте только одно решение.* Если нам повезет, анализ пространства возможных решений покажет, что одна программа намного лучше всех остальных; в других случаях мы должны рассмотреть несколько вариантов, чтобы выбрать лучший. Надо стараться реали-

зовать выбранный вариант в виде краткой и простой программы<sup>1</sup>.

*Осмотритесь.* Прекрасная работа Пойа "Как решить задачу" может помочь любому программисту улучшить свои навыки решения задач. Он отмечает, что всегда можно что-нибудь сделать; изучив и поняв задачу, можно улучшить любое решение, и в любом случае мы можем улучшить наше понимание того, как решать задачу. Его советы особенно полезны при рассмотрении задач, возникающих при программировании.

### 11.5. ЗАДАЧИ

1. В разд. 11.1 требуется, чтобы все одинаковые подмножества из  $M$  элементов выбирались с одинаковой вероятностью, что является более строгим требованием, чем выбор каждого целого числа с вероятностью  $M/N$ . Опишите алгоритм, который осуществляет выбор каждого элемента равновероятно, но некоторые подмножества выбираются с большей вероятностью, чем другие.
2. Покажите, что при  $M < N/2$  ожидаемое число проверок, выполняемых в программе 2 функцией `Member` перед обнаружением элемента, отсутствующего в наборе, меньше 2.
3. Вычисление числа проверок, выполняемых функцией `Member` в программе 2, приводит к многим интересным задачам по комбинаторике и теории вероятностей. Как представить среднее число проверок, выполняемых функцией `Member`, в виде функции от  $M$  до  $N$ ? Сколько их при  $M = N$ ? Когда число проверок превысит  $M$ ?
4. В этой главе описано несколько алгоритмов для решения одной задачи. Реализуйте некоторые из них, измерьте их производительность и опишите пригодность каждого из них в зависимости от ограничений на время работы, объем памяти, время программирования и т. д.

---

<sup>1</sup> Задача 5 представляет собой упражнение, с помощью которого я оценивал стиль программирования. Большинство студентов вернули решение на одном листе и получили посредственные оценки. Два студента, потратившие предыдущее лето на участие в разработке большого программного проекта, вернули прекрасно документированные программы на пяти страницах, разбитые на дюжину процедур, каждая из которых имела тщательно разработанный заголовок. Они получили неудовлетворительные оценки. Моя работающая программа состояла из 5 строк, и достигнутый ими коэффициент увеличения размера программы (в 60 раз) был слишком велик для сдачи экзамена. Когда они выразили недовольство, так как использовали стандартные средства разработки программных систем, мне пришлось процитировать П. Зейва: "Цель техники программирования — управлять сложностью, а не создавать ее." Несколько дополнительных минут, потраченных на поиск простой программы могут сэкономить впоследствии часы, необходимые на документирование сложной программы.

5. На курсе по алгоритмам для студентов я дважды даю задачу по генерации упорядоченных подмножеств. Перед темой "Сортировка и поиск" студенты должны написать программу для случая  $M = 20$  и  $N = 400$ ; основными критериями оценки являются краткость и ясность программы – время работы не учитывается. После прохождения темы "Сортировка и поиск" они должны решить задачу повторно при  $M = 2000$  и  $N = 1\,000\,000$ , а оценка зависит в первую очередь от времени прогона.
6. [В. Висоцки] Алгоритмы для порождения комбинаторных объектов, как правило, удобно выражать в виде рекурсивных процедур. Программа 1 может быть написана так:

```

procedure RandSelect(M,N)

pre    $\emptyset \leq M \leq N$            /* На входе */

post  M различных целых чисел в диапазоне от 1 до N
      напечатаны в упорядоченном виде /* На выходе */

if M >  $\emptyset$  then
    if RandReal( $\emptyset$ ,1) < M/N then
        print N; RandSelect(M-1,N-1)
    else
        RandSelect(M,N-1)

```

Эта программа выводит на печать случайные числа в убывающем порядке; как добиться того, чтобы они появлялись в возрастающем порядке? Обоснуйте корректность получившейся в результате программы. Как бы вы использовали основную рекурсивную структуру этой программы для генерации всех подмножеств из  $M$  элементов множества  $1, \dots, N$ ?

7. Как бы вы сгенерировали случайную последовательность из  $M$  целых чисел от 1 до  $N$  с ограничением на то, что выходные данные должны следовать в произвольном порядке? Как бы вы сгенерировали упорядоченный список, если бы в нем а) допускались дубликаты; б) желательно было иметь и дубликаты, и произвольный порядок?
8. [М. И. Шамос] Имеется карта с десятью пятнами, которые скрывают случайно размещенные числа от 1 до 10. Игрок стирает с карты пятна, чтобы выявить скрытые числа. Если появляется число 3, карта проигрывает; если обнаружены 1 и 2 (в любом порядке), то

карта выигрывает. Опишите шаги, которые бы вы предприняли для вычисления вероятности того, что случайно выбранная последовательность стираемых пятен приводит к выигрышу; предположим, что вы можете использовать не более 1 ч процессорного времени

#### 11.6. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

В книге Нийенхуиса и Вилфа "Комбинаторные алгоритмы для компьютеров и калькуляторов" (Nijenhuis, Wilf. Combinatorial Algorithms for Computers and Calculators, второе издание, выпущенное издательством Academic Press в 1978 г.) представлена большая коллекция алгоритмов как в абстрактном виде, так и в виде готовых к работе программ на Фортране. В этой книге особенно много алгоритмов для порождения комбинаторных объектов, таких как случайные выборки.

### Глава 12. ПИРАМИДЫ

В этой главе рассматриваются пирамиды<sup>1</sup> — структуры данных, которые мы будем использовать для решения следующих двух задач.

*Сортировка.* Пирамидальная сортировка массива из  $N$  элементов производится за время  $O(N \log N)$  и требует только несколько дополнительных слов памяти.

*Очереди с приоритетами.* Использование пирамид обеспечивает выполнение операций по вставке нового элемента в множество элементов и удаления наименьшего элемента из множества; каждая из этих операций выполняется за время  $O(\log N)$ .

Использование пирамид в обеих этих задачах облегчает программирование и обеспечивает эффективность вычислений.

Материал этой главы излагается методом "снизу вверх": мы начнем с подробностей и перейдем к общей картине. В двух следующих разделах описаны пирамидальные структуры данных и две подпрограммы для работы с ними. В двух разделах, следующих за ними, эти структуры используются для решения упомянутых выше задач.

---

<sup>1</sup> Используемый автором термин *heap* будем переводить как "пирамида", а *heapsort* — как "пирамидальная сортировка", как это сделано в переводе книги Кнута "Сортировка и поиск", с. 177). — Прим. перев.

## 12.1 СТРУКТУРЫ ДАННЫХ

Пирамида – это структура данных для представления набора элементов<sup>1</sup>. В наших примерах они будут фигурировать как целые числа, но элементами пирамиды могут быть данные любого типа, обладающие свойством упорядоченности. Ниже показана пирамида, содержащая двенадцать целых чисел:

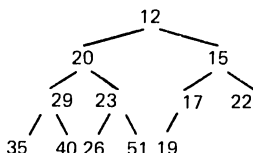


Рис. 12.1

Это двоичное дерево является пирамидой в силу следующих двух признаков. Первый признак мы будем называть *порядком*: значение элемента в любом узле меньше или равно значениям его "сыновей". Отсюда следует, что наименьший элемент является корнем дерева (в данном примере узел 12), но ничего нельзя сказать об относительном порядке левых и правых "сыновей". Второй признак пирамиды – вид; эта идея отражена в следующем рисунке:



Рис. 12.2

Объясняя словами, можно сказать, что в двоичном дереве, обладающем признаком *вида*, концевые узлы расположены не более чем на двух уровнях, причем нижний уровень сдвинут влево, насколько это возможно. В дереве нет "дырок"; если в нем содержится  $N$  узлов, то ни один узел не удален от корня на расстояние, большее чем  $\log_2 N$ . Мы вскоре увидим, что эти два признака вместе достаточно жестки, что позволяет находить минимальный элемент в наборе данных, но в то же время достаточно мягки, поэтому нужно эффективно реорганизовать структуру после вставки или удаления элемента.

Давайте от абстрактных свойств пирамид обратимся теперь к их реализации. Имеется, конечно, много возможных представлений двоичных деревьев, таких как использование записей и указателей. Мы будем использовать представление, пригодное только для двоичных деревьев,

---

<sup>1</sup> В другом контексте применительно к вычислительной технике слово "heap" означает большой сегмент памяти, в котором размещаются "узлы" произвольной длины; в данной главе мы не будем рассматривать такую интерпретацию.

обладающих признаком вида, но для этого случая оно достаточно эффективно. Дерево из 12 элементов, обладающее признаком *вида*, реализуется в виде следующего 12-элементного массива:

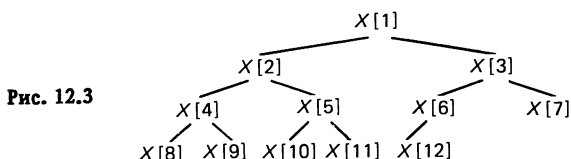


Рис. 12.3

В этом неявном представлении двоичного дерева корень находится в элементе  $X[1]$ , два его "сына" – в элементах  $X[2]$  и  $X[3]$  и т. д. Различные функции на этом дереве определены следующим образом:

Root = 1	/* Корень */
Value(I) = X[I]	/* Значение элемента I */
LeftChild(I) = 2*I	/* Левый "сын" элемента I */
RightChild(I) = 2*I+1	/* Правый "сын" элемента I */
Parent(I) = I div 2	/* "Отец" элемента I */
Null(I) = (I < 1) or (I > N)	

Неявно представленное дерево из  $N$  элементов неизбежно обладает признаком *вида*: отсутствующие элементы не предусмотрены.

На следующем рисунке показана 12-элементная пирамида и ее реализация в виде неявного представления дерева в 12-элементном массиве:

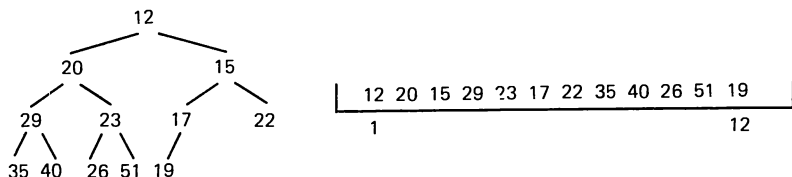


Рис. 12.4

Так как признак *вида* гарантируется при таком представлении, с этого момента мы будем использовать название "пирамида" для обозначения того факта, что значение в любом узле больше или равно значению его "отца". Точная формулировка такова: массив обладает признаком пирамидальности, если

$$\forall_{2 \leq i \leq N} X[i \text{ div } 2] \leq X[i].$$



В следующем разделе мы будем рассматривать массив  $X [L \dots U]$ , в котором соблюдается признак пирамидальности (но не соблюдается признак вида). Функция  $\text{Heap} (L, U)$  имеет значение "истина" тогда и только тогда, когда

$$\forall_{2 \leq i \leq N} X [i \text{ div } 2] \leq X [i].$$

## 12.2. ДВЕ ВАЖНЫЕ ПОДПРОГРАММЫ

В этом разделе мы рассмотрим две подпрограммы для восстановления признака *пирамидальности* в массиве, в одном из концов которого он нарушен. Обе подпрограммы эффективны: на реорганизацию пирамиды из  $N$  элементов им потребуется время, пропорциональное  $\log N$ . В соответствии с идеей изложения "снизу вверх" мы определили эти подпрограммы здесь, а использовать их будем в следующих разделах.

Если массив  $X [1 \dots N - 1]$  являлся пирамидой, то помещение произвольного элемента в позицию  $X [N]$ , вероятно, не приведет к пирамиде массив  $X [1 \dots N]$ ; восстановление признака *пирамидальности* является задачей процедуры  $\text{SiftUp}$  (сдвиг вверх). В ее имени заключена стратегия ее работы: мы сдвигаем новый элемент вверх по дереву до тех пор, пока это требуется, меняя его местами с "отцом" на всем пути. (Потратьте секунду, чтобы понять, какой путь ведет вверх: корень пирамиды находится в вершине дерева, и поэтому элемент  $X [N]$  расположен в конце массива.) Этот процесс иллюстрируется следующими рисунками, на которых показано (слева направо), как новый элемент сдвигается вверх по пирамиде до тех пор, пока он не окажется в нужной позиции и не станет правым сыном корневого элемента:

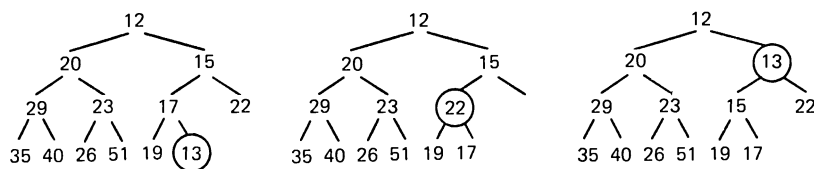


Рис. 12.5

Процесс продолжается, пока значение обведенного кругом узла не станет больше значения своего "отца" или равно ему (как в данном случае), или пока этот узел не станет корнем дерева. Если процесс начинается, когда функция  $\text{Heap} (1, N - 1)$  имеет значение "истина", то значение функции  $\text{Heap} (1, N)$  сохраняется тем же.

Имея интуитивную основу, давайте напишем программу. Сдвиг будет производиться в цикле, поэтому мы должны начать с задания

инварианта цикла. На вышеприведенных рисунках признак *пирамидальности* имеет место на дереве везде, за исключением обведенного кругом узла и его "отца". Если мы обозначим индекс этого выделенного узла через  $I$ , то сможем использовать следующий инвариант:

Цикл

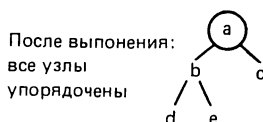
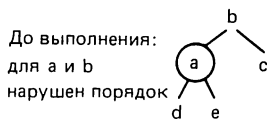
/\* Инвариант: функция  $\text{Heap}(1, N)$  истинна, за исключением,  
возможно, участка от  $I$  до его "отца" \*/

Так как в начальный момент функция  $\text{Heap}(1, N - 1)$  имеет значение "истина", мы начинаем цикл с присваивания  $I := N$ .

В цикле должно проверяться, не пора ли нам уже закончить его (либо потому, что выделенный узел оказался на вершине пирамиды, либо его значение стало равно или больше значения его "отца"), а если нет, то продвигаться вперед к завершению. Этот инвариант сообщает, что признак *пирамидальности* имеет место везде, за исключением участка от узла  $I$  до его "отца". Если выполнено проверяемое условие  $I = 1$ , то узел  $I$  не имеет "отца" и признак *пирамидальности* имеет место везде; поэтому цикл можно завершить. Если узел  $I$  имеет "отца", то мы присвоим переменной  $P$  значение индекса "отца", выполнив операцию  $P := I \text{ div } 2$ . Если  $X[P] \leq X[I]$ , то признак *пирамидальности* имеет место везде и цикл можно завершить.

С другой стороны, если узел  $I$  и его "отец" не стоят в нужном порядке, мы меняем местами элементы  $X[I]$  и  $X[P]$ . Этот шаг иллюстрируется следующими рисунками, в которых значениям узлов соответствуют буквы, а узел  $I$  обведен кругом:

Рис. 12.6



После перестановки все пять элементов расположены в соответствующем порядке:  $b < d$  и  $b < e$ , так как  $b$  находился в начальный момент выше в пирамиде<sup>1</sup>,  $a < b$ , так как условие  $X[P] \leq X[I]$  не выполняется, и  $a < c$ , что следует из совместного выполнения условий  $a < b$  и  $b < c$ . Это обеспечивает соблюдение признака *пирамидальности* везде в массиве,

<sup>1</sup> Это важное свойство не сформулировано в инварианте цикла. Кнут отметил, что инвариант должен быть усилен: "Функция  $\text{Heap}(1, N)$  имеет значение "истина", если узел  $I$  не имеет "отца"; в противном случае она имела это значение при замене элемента  $X[I]$  на  $X[P]$ , где узел  $P$  является "отцом" узла  $I$ ." Подобное уточнение должно также использоваться в цикле  $\text{SiftDown}$ , который мы вкратце рассмотрим.

за исключением, быть может, интервала между узлом I и его "отцом". Поэтому мы вновь достигнем инварианта, присваивая  $I := P$ .

Из вышеприведенных фрагментов формируется программа SiftUp, время работы которой пропорционально  $\log N$ , так как пирамида имеет  $\log N$  уровней.

```

procedure SiftUp(N)
    pre   Heap(1,N-1) и  $N > 0$     /* На входе */
    post  Heap(1,N)              /* На выходе */

    I := N
    loop
/* Инвариант: функция Heap(1,N) истинна, за исключением,
    возможно, участка от I до его "отца" */
        if I = 1 then break
        P := I div 2
        if X[P] <= X[I] then break
        Swap(X[P], X[I])
        I := P

```

Как и в гл. 4, строки pre (на входе) и post (на выходе) характеризуют эту процедуру: если условие pre имеет место перед вызовом процедуры, то условие post устанавливается после ее завершения.

Присваивание элементу  $X[1]$  нового значения при условии, что массив  $X[1 \dots N]$  является пирамидой, сохраняет истинность функции Heap(2, N); процедура SiftDown (сдвиг вниз) восстанавливает истинность функции Heap(1, N), сдвигая элемент  $X[1]$  вниз в массиве до тех пор, пока у него не окажется "сыновей" или пока он больше своих "сыновей". На следующем рисунке показан узел 18, который сдвигается вниз по пирамиде до тех пор, пока его значение не станет, наконец, меньше значения его единственного "сына" (узла 19):

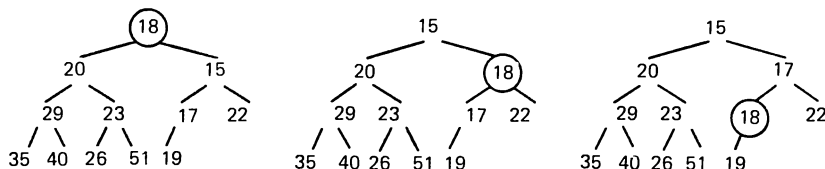


Рис. 12.7

Когда элемент сдвигается вверх, он всегда движется к корню. Перемещение элемента вниз является более сложным: элемент, расположенный не на своем месте, меняется местами с наименьшим из "сыновей".

Вышеприведенный рисунок иллюстрирует инвариант цикла процедуры SiftDown: признак *пирамидальности* имеет место везде, кроме, быть может, участка от выделенного узла до его "детей":

Цикл

```
/* Инвариант: функция Heap(1,N) истинна, за исключением,  
возможно, участка от I до его (0, 1 или 2) "сыновей" */
```

Этот цикл похож на цикл процедуры SiftUp. Сначала мы проверяем, имеет ли узел I хоть одного "сына", и завершаем цикл, если нет. Теперь наступает тонкий момент: если узел I имеет "сыновей", то мы засылаем в переменную C индекс наименьшего из "сыновей" узла I. Наконец, мы либо завершаем цикл, если  $X[I] \leq X[C]$ , либо продвигаемся дальше вниз, меняя местами элементы  $X[I]$  и  $X[C]$  и присваивая  $I := C$ .

```
procedure SiftDown(N)
```

```
    pre    Heap(2,N) и  $N > 0$       /* На входе */
```

```
    post   Heap(1,N)              /* На выходе */
```

```
    I := 1
```

```
    loop
```

```
/* Инвариант: функция Heap(1,N) истинна, за исключением,
```

```
возможно, участка от I до его (0, 1 или 2 "сыновей" */
```

```
    C := 2*I
```

```
    if C > N then break
```

```
/* C - левый "сын" I */
```

```
    if C+1 <= N then
```

```
        /* C - правый "сын" I */
```

```
        if  $X[C+1] < X[C]$  then
```

```
            C := C+1
```

```
/* C - наименьший из "сыновей" I */
```

```
    if  $X[I] <= X[C]$  then break
```

```
    Swap( $X[C]$ ,  $X[I]$ )
```

```
    I := C
```

Анализ этой процедуры, подобный тому, который был выполнен для процедуры SiftUp, показывает, что операция взаимной перестановки (Swap) сохраняет соблюдение признака *пирамидальности* везде, кроме, может быть, участка между узлом C и его "сыновьями". Подобно процедуре SiftUp, на выполнение процедуры затрачивается время, пропорциональное  $\log N$ , потому что в ней выполняется фиксированный объем работы на каждом уровне пирамиды.

### 12.3. ОЧЕРЕДИ С ПРИОРИТЕТАМИ

Любую структуру данных можно рассматривать с двух точек зрения. Взгляд "снаружи" позволяет по абстрактному описанию судить о ее функциях: очередь – это средство для работы с последовательностью элементов с помощью операций вставки и удаления. Взгляд "изнутри" позволяет разобраться в ее реализации и узнать, как выполняются эти функции – для очереди можно было бы использовать массив или связанный список. Мы начнем наше изучение очередей с приоритетами с их абстрактных свойств, а потом обратимся к реализации.

Очередь с приоритетами манипулирует множеством<sup>1</sup> элементов, которое сначала пусто. Обозначим его S. Процедура Insert (вставка) вставляет новый элемент в множество; мы можем определить ее более точно с помощью состояний на входе и на выходе.

Procedure Insert(T)

```
pre   !S! < MaxSize                               /* На входе */
post  текущее S = исходное S U {T}                /* На выходе */
```

Процедура ExtractMin удаляет наименьший элемент из множества и передает его значение в виде своего единственного параметра T:

Procedure ExtractMin(t)

```
pre   !S! > 0                                       /* На входе */
post  исходное S = текущее S U {T}
      и T = min(исходное S) /* На выходе */
```

Эта процедура могла бы быть, конечно, модифицирована для того, чтобы извлекать максимальный элемент или любой другой крайний элемент при полном упорядочении.

---

<sup>1</sup> Так как это множество может содержать несколько копий одного и того же элемента, более точно его можно назвать "множеством с повторяющимися элементами". Оператор объединения определяется так  $\{2, 3\} \cup \{2\} = \{2, 2, 3\}$ .

Очереди с приоритетами используются во многих прикладных задачах. В операционной системе такая структура может служить для представления набора задач; они поступают в произвольном порядке, а для выполнения выбирается задача с наивысшим приоритетом. При имитационном моделировании дискретных событий: в цикле моделирования извлекается событие с наименьшим значением времени, а в очередь, возможно, добавляются новые события. В обеих прикладных задачах основная очередь с приоритетами должна быть расширена, чтобы кроме элементов этого множества содержать дополнительную информацию; в нашем рассмотрении мы будем игнорировать детали реализации.

Последовательные структуры, такие как массивы и связанные списки, являются очевидными кандидатами для реализации очередей с приоритетами. Если последовательность упорядочена, то легко удалить минимальный элемент, но трудно вставить новый элемент; для неупорядоченных структур ситуация противоположна. В таблице показана эффективность этих структур для множества из  $N$  элементов:

Структура данных	Время работы		
	для вставки нового элемента	для удаления минимального элемента	для $N$ операций
Упорядоченная последовательность	$O(N)$	$O(1)$	$O(N^2)$
Пирамида	$O(\log N)$	$O(\log N)$	$O(N \log N)$
Неупорядоченная последовательность	$O(1)$	$O(N)$	$O(N^2)$

Несмотря на то, что с помощью алгоритма двоичного поиска можно найти позицию для вставляемого элемента в упорядоченном массиве за время  $O(\log N)$ , для перемещения старых элементов, чтобы освободить места для нового, может потребоваться  $O(N)$  шагов. Если вы забыли разницу между алгоритмами с временем работы  $O(N^2)$  и  $O(N \log N)$ , вернитесь к разд. 7.5: при  $N = 100\,000$  время работы таких программ составляет 1.5 дня и 1.3 мин.

Реализация очередей с приоритетами в виде пирамид является золотой серединой между экстремальными вариантами, получаемыми с помощью последовательных структур. В этом случае  $N$ -элементное множество представляется в виде массива  $X[1 \dots N]$ , обладающего признаком *пирамидальности*, где  $X$  объявлен как массив  $X[1 \dots \text{MaxSize}]$ . Мы задаем в начальный момент пустое множество, присваивая  $N := 0$ . Чтобы вставить новый элемент, мы увеличиваем  $N$  на

1 и размещаем элемент в  $X[N]$ , что приводит к истинности функции Heap (1,  $N - 1$ ). Для обработки такой ситуации была разработана подпрограмма SiftUp. Поэтому программа для вставки выглядит следующим образом:

```
procedure Insert(T)
    if N >= MaxSize then error    /* Ошибка */
    N := N+1; X[N] := T
    /* Функция Heap(1,N-1) - истинна */
    SiftUp(N)
    /* Функция Heap(1,N) - истинна */
```

Процедура ExtractMin определяет минимальный элемент множества, удаляет его и реконструирует массив, чтобы он обладал признаком *пирамидальности*. Так как этот массив является пирамидой, минимальный элемент находится в  $X[1]$ . Остающиеся в множестве  $N - 1$  элементов находятся в подмассиве  $X[2 \dots N]$ , имеющем признак пирамидальности. Мы за два шага вновь достигаем состояния, когда функция Heap (1, N) имеет значение "истина". Сначала мы пересылаем элемент  $X[N]$  в  $X[1]$  и уменьшаем N на 1; элементы множества расположены в массиве  $X[1 \dots N]$ , и функция Heap (2, N) имеет значение "истина". На следующем шаге вызывается процедура SiftDown.

Программа очень проста:

```
procedure ExtractMin(T)
    if N < 1 then error    /* Ошибка */
    T := X[1]
    X[1] := X[N]; N := N-1
    /* Функция Heap(2,N) - истинна */
    SiftDown(N)
    /* Функция Heap(1,N) - истинна */
```

Для пирамиды из N элементов на процедуры и Insert, и ExtractMin требуется время  $O(\log N)$ .

#### 12.4. АЛГОРИТМ СОРТИРОВКИ

Очереди с приоритетами позволяют реализовать простой алгоритм сортировки массива  $A[1 \dots N]$ .

```

for I := 1 to N do
    Insert(A[I])
for I := 1 to N do
    ExtractMin(A[I])

```

На  $N$  операций Insert и ExtractMin затрачивается в худшем случае  $O(N \log N)$  времени, что лучше, чем максимальное время выполнения –  $O(N^2)$  программы быстрой сортировки из разд. 10.2. К сожалению, для массива  $X[1 \dots N]$ , используемого для пирамиды, требуются дополнительные слова памяти.

В этом разделе мы рассмотрим алгоритм пирамидальной сортировки, который является улучшенным вариантом вышеописанного подхода: он реализуется более короткой программой и использует меньший объем памяти, так как ему не требуется дополнительный массив, и выполняется за более короткое время – см. задачу 2. Предположим, что подпрограммы SiftUp и SiftDown модифицированы для работы с пирамидой, в вершине которой находится наибольший элемент; это просто осуществить, поменяв местами знаки  $<$  и  $>$ .

В простом алгоритме используются два массива, один для очереди с приоритетами, а второй – для сортируемых элементов. Алгоритм пирамидальной сортировки экономит место, поскольку используется только один массив. При реализации с единственным массивом  $X$  он соответствует двум абстрактным структурам: в левой части – пирамида, в правой – последовательность элементов первоначально в произвольном порядке, а в итоге – упорядоченная. На рисунке показана эволюция массива  $X$ ; массив нарисован горизонтально, а время отложено вниз по вертикальной оси:

Рис. 12.8



Алгоритм пирамидальной сортировки состоит из двух этапов: на первых  $N$  шагах массив преобразуется в пирамиду, а на последующих  $N$  шагах удаляются элементы в убывающем порядке и образуется окончательная упорядоченная справа налево последовательность.



На первом этапе строится пирамида. Его инвариант может быть изображен так:

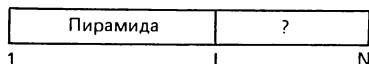


Рис. 12.9

Эта программа формирует пирамиду (1, N):

```

for I := 2 to N do
    /* Инвариант: функция Heap(1,I-1) - истинна */
    SiftUp(I)
    /* Функция Heap(1,N) - истинна */

```

На втором этапе пирамида используется для построения упорядоченной последовательности. Его инвариант может быть изображен так:

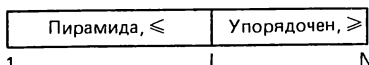


Рис. 12.10

В теле цикла для сохранения инварианта используются две операции. Так как элемент  $X[1]$  является максимальным среди первых  $I$  элементов, перестановка (Swap) его с элементом  $X[I]$  расширяет упорядоченную последовательность на один элемент. Эта перестановка подвергает риску признак *пирамидальности*, который мы восстанавливаем, сдвигая вниз новый элемент, находящийся в вершине. Программа для второго этапа выглядит так:

```

for I := N downto 2 do /* Цикл с убыванием счетчика */
    /* Функция Heap(1,I) - истинна, интервал (I+1,N) - упорядочен и
        X[1...I]  $\leq$  X[I+1...N] */
        Swap(X[1], X[I])
    /* Функция Heap(2,I-1) - истинна, интервал (I,N) - упорядочен и
        X[1...I-1]  $\leq$  X[I...N] */
        SiftDown(I-1)
    /* Функция Heap(1,I-1) - истинна, интервал (I,N) - упорядочен и
        X[1...I-1]  $\leq$  X[I...N] */

```

Для реализации алгоритма пирамидальной сортировки требуется всего 5 строк:

```
for I := 2 to N do
    SiftUp(I)
for I := N downto 2 do
    Swap(X[1],X[I])
    SiftDown(I-1)
```

Так как этот алгоритм использует только  $N - 1$  обращение к процедурам SiftUp и к SiftDown, на каждую из которых затрачивается время, не превышающее  $O(\log N)$ , время работы алгоритма, даже в худшем случае,  $O(N \log N)$ .

## 12.5. ОСНОВНЫЕ ПРИНЦИПЫ

*Эффективность.* Признак *вида* гарантирует, что все узлы пирамиды находятся на ближайших  $\log_2 N$  уровнях от корня; процедуры SiftUp и SiftDown имеют оптимальное время работы именно потому, что дерево сбалансировано. В пирамидальной сортировке за счет перекрытия двух абстрактных структур (пирамида и последовательность) при реализации их в одном массиве устранено использование дополнительной памяти.

*Корректность.* Для написания команд цикла мы вначале точно сформулировали инвариант; потом цикл успешно продвигался вперед к своему завершению, сохраняя при этом условие инвариантности. Признаки вида и порядка представляют собой другой вид инварианта: они являются инвариантом пирамиды как структуры данных. В подпрограмме работающей с пирамидой, может предполагаться, что эти признаки имеют место перед началом ее работы и она должна, в свою очередь, гарантировать их сохранение при своем завершении.

*Абстракция.* Хорошие инженеры понимают различия между тем, что делает некоторый компонент (это абстракция, которую видит пользователь), и тем, как он это делает (реализация внутри "черного ящика"). В этой главе мы рассмотрели два способа упаковки "черного ящика": процедурная абстракция и абстрактные типы данных.

*Процедурная абстракция.* Мы можем использовать процедуру сортировки (sort) для упорядочения массива, не зная, как она реализована: мы рассматриваем sort, как одну операцию. Процедуры SiftUp и SiftDown дают аналогичный уровень абстракции: когда мы создаем очередь с приоритетами и выполняем пирамидальную сортировку, мы не заботимся о том, как действуют эти процедуры, но знаем, что

они делают (восстанавливает в массиве признак *пирамидальности*, вероятно, нарушенного на одном из его концов). Выполняя разработку на хорошем уровне, мы один раз формируем черные ящики, а потом используем их для компоновки двух различных систем.

**Абстрактные типы данных.** Реализованные в языках программирования типы данных с некоторыми ограничениями абстрактно определены с помощью математических объектов и операций над этими объектами (пользователю не требуется знать, как они реализованы); "Словари" в разд. 11.3 и очереди с приоритетами в этой главе можно отнести к подобным типам данных:

	Целые числа	Очереди с приоритетами	Словари
Математическая модель	Целое число	Множество целых чисел	Множество целых чисел
Операции	Присваивание, суммирование и т. д.	Начальная очистка, Insert (вставка)	Начальная очистка, Insert, Member (проверка принадлежности)
		ExtractMin (удаление минимального элемента)	PrintInOrder (распечатка в заданном порядке)
Ограничения	Максимальный и минимальный размеры	Максимальный размер множества Размер элементов	Максимальный размер множества Размер элементов
Реализация	Дополнение до числа 2, десятичные числа со знаком	Упорядоченный массив, пирамида	Битовый вектор, "карманы", массивы, деревья

Некоторые современные языки программирования позволяют программистам определить свои собственные типы данных, такие, как очереди с приоритетами. В последующих командах можно объявить, что переменная имеет тип "очередь с приоритетом"; программа будет воспринимать только эту абстракцию и может ничего не знать о ее реализации. Программа такого типа приведена в решении задачи 10; описанная дисциплина программирования упрощает повторное использование программного обеспечения.

## 12.6. ЗАДАЧИ

1. Модифицируйте процедуру SiftDown так, чтобы она имела следующую спецификацию:

proc SiftDown(L,U)

pre Heap(L+1,U) /\* На входе \*/

post Heap(L,U) /\* На выходе \*/

Чему равно время работы программы? Покажите, как она может быть использована для построения  $N$ -элементной пирамиды за время  $O(N)$ , что ускорит выполнение программы пирамидальной сортировки, а также сократит число команд.

2. Реализуйте программу пирамидальной сортировки так, чтобы она работала с максимальным быстродействием. Сравните его с быстродействием алгоритмов сортировки, приведенных в таблице в разд. 10.3? Реализуйте программу для очередей с приоритетами на основе пирамиды так, чтобы она работала с максимальным быстродействием; при каких значениях  $N$  оно превосходит быстродействие, обеспечиваемое последовательными структурами?
3. Каким образом можно использовать очереди с приоритетами, реализованные на основе пирамид, для решения следующих задач:
  - А. Построить код Хоффмана (такие коды рассматриваются в большинстве книг по теории информации и во многих книгах по структурам данных).
  - Б. Вычислить сумму большого набора чисел с плавающей точкой.
  - В. Найти 1000 наибольших среди 10 млн чисел, хранящихся на магнитной ленте.
  - Г. Объединить много маленьких упорядоченных файлов в один большой упорядоченный файл (такая задача возникает при реализации программы сортировки слиянием на диске, подобной рассмотренной в разд. 1.3). Как изменятся ваши ответы, если входные данные упорядочены?
4. [Д. С. Джонсон] В задаче упаковки "карманов" требуется разместить набор из  $N$  весов ( $N = 0, \dots, 1$ ) в минимальном количестве "карманов" единичной емкости. Первое, что приходит в голову, рассмотреть эти веса в порядке их предъявления и разместить каждый вес в первом подходящем "кармане", просматривая "карманы" в возрастающем порядке. Покажите, как с помощью структуры, аналогичной пирамиде, можно реализовать этот метод за время  $O(N \log N)$ . (Эта задача связана с эффективными алгоритмами выделения первого подходящего фрагмента памяти, таким, как в упражнении 6.2.4.30 книги Кнута "Сортировка и поиск".)
5. [И. Маккрейт] При обычной реализации последовательных файлов на диске в каждом блоке есть указатель на блок, следующий за ним. При таком методе затрачивается постоянное время на запись блока

(при первоначальной записи файла), на чтение первого блока файла и на чтение I-го блока, если перед этим прочитан I – 1 блок. Поэтому на чтение I-го блока с начала файла требуется время, пропорциональное I. Покажите, как, добавив только еще один указатель в каждый блок, вы можете сохранить все эти свойства, позволить читать I-й блок за время, пропорциональное  $\log I$ . Объясните, что общего у алгоритма для чтения I-го блока с программой из задачи 9 гл. 4 для возведения числа в I-ю степень за время, пропорциональное  $\log I$ .

6. На многих компьютерах наиболее "дорогая" часть программы двоичного поиска – деление на 2 для нахождения центра текущей области. Объясните, как заменить это деление умножением в предположении, что таблица построена соответствующим образом. Приведите алгоритмы построения таких таблиц и поиска в них.
7. Какова подходящая реализация для очереди с приоритетами, в которой представлены целые числа в диапазоне от 1 до K, если средний размер длины очереди намного больше K.
8. Докажите, что логарифмическая зависимость времени работы процедур Insert (вставка) и ExtractMin (удаление минимального элемента) при реализации очередей с приоритетами посредством пирамид является оптимальной с точностью до коэффициентов.
9. Основная идея пирамиды близка спортивным болельщикам. Предположим, что в полуфинале Брайан победил Эла, Линн победил Питера, а в борьбе за чемпионский титул Линн одержал верх над Брайном. Подобные результаты обычно изображаются так:

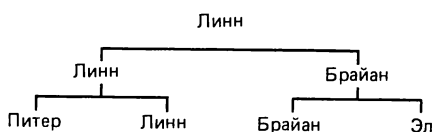


Рис. 12.11

Такое "турнирное дерево" обычно для теннисных турниров и в финальных играх с выбыванием, по окончании сезона, в футболе, бейсболе и баскетболе. В предположении, что результаты матчей закономерные (это предположение часто неверно в спорте), определите, какова вероятность, что второй по силе игрок выйдет в финал чемпионата? Дайте алгоритм распределения игроков по силе в соответствии с их результатами перед матчем.

10. Реализуйте пирамиду на языке, в котором поддерживаются абстрактные типы данных.

## 12.7. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Первая статья Дж. В. Дж. Уильямса (J. W. J. Williams) на тему пирамид появилась в июньском номере журнала Communications of the ACM за 1964 г., но даже сегодня ее еще принято читать. Хотя по сегодняшним стандартам она слишком коротка (менее 1 с.), по стандартам того времени она была длинной (алгоритм Флойда TREESORT в августовском номере CACM за 1962 г. занял менее 1/4 с.).

Более современные идеи на эту тему изложены в книге Тарьяна "Структуры данных и сетевые алгоритмы" (Tarjan. Data Structures and Network Algorithms), опубликованной Society for Industrial and Applied Mathematics в 1983 г. Эта монография является прекрасным введением в область, определенную названием книги; гл. 3 посвящена пирамидам.

## Глава 13. ПРОГРАММА ПРОВЕРКИ ПРАВОПИСАНИЯ

Орфографические ошибки раздражают читающих. А для большинства пишущих проверка правописания – скучная и способствующая ошибкам работа. К счастью, эта задача идеально подходит для компьютеров: монотонная, однообразная работа, которая требует быстрого чтения и хорошей памяти.

В этой главе мы рассмотрим, как построена системная программа проверки правописания spell в операционной системе UNIX. Это прелестная и полезная программа, ее история богата важными уроками по разработке программ.

### 13.1. ПРОСТАЯ ПРОГРАММА

Однажды после обеда (в 1975 г.) С. Джонсон написал первую версию программы spell. Его подход был прост: выделять слова в документе, сортировать их, а потом сравнивать этот упорядоченный список со словарем:

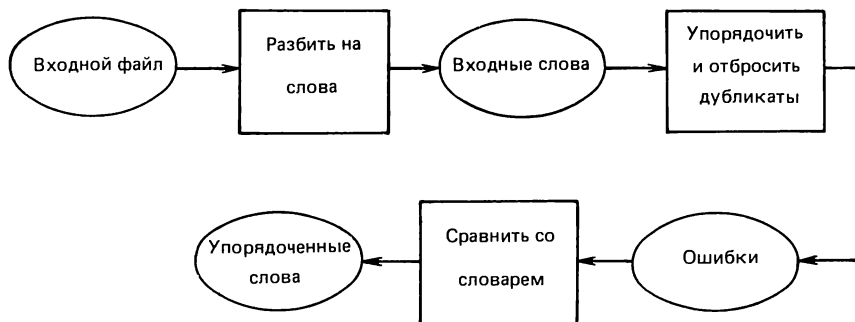


Рис. 13.1

Ее выход представляет собой список всех слов документа, отсутствующих в словаре. Если вы собирались написать слово `stationary`, этой программой будет выявлена такая ошибка, как `stationiry`, но не будет обнаружена ошибка, если написано `stationery`, так как это – другое правильное слово.

Керниган и Плоджер воссоздали программу Джонсона в своей книге "Программные средства на Паскале". Их конвейерную программу можно сформулировать так:

```
prerepare filename :      # Исключить команды форматирования
translit A-Z a-z :      # Заменить прописные буквы строчными
translit ^a-z \n :      # Исключить знаки препинания
sort :                  # Упорядочить слова по алфавиту
unique :               # Исключить дубликаты
common -2 dict         # Выдать слова, которых нет в словаре
```

Вертикальный штрих соединяет выход одной программы с входом следующей программы. Входом первой программы является имя файла `filename`, а выходом последней программы – список (потенциально) ошибочно написанных слов.

Первая программа в этом конвейере (`prerepare`) имеет дело с командами форматирования, присутствующими во многих документах, обрабатываемых с помощью компьютеров. Например, чтобы напечатать слово `boltface` жирным шрифтом, в одном случае может быть введено `@b` (`boltface`), а в другом `/fBboltface/fR`. Программа проверки правописания должна игнорировать указанные команды формирования; бедный пользователь, продирающийся через опечатки, подобные `b` и `fbboltface`, слишком изнурен, чтобы замечать действительные ошибки. Программа `prerepare` копирует входной файл в выходной, удаляя при этом команды форматирования.

Программа `translit` копирует входной файл в выходной, заменяя некоторые символы. При первом обращении к ней она заменяет прописные буквы строчными. При следующем обращении удаляются все не принадлежащие алфавиту символы посредством преобразования их в признак новой строки (`/n`). Результатом является файл, содержащий слова документа в порядке их появления. В каждой строке расположено не более одного слова (многие строки пусты).

Следующая программа (`sort`) сортирует слова в алфавитном порядке, а программа `unique` удаляет дубликаты. Результатом является упорядоченный массив, содержащий все различные слова документа. Програм-

ма соптон с загадочным дополнением (-2) использует стандартный алгоритм слияния, чтобы сообщить обо всех строках своего входного (упорядоченного) файла, которые отсутствуют в (упорядоченном) файле с указанным именем. Ее выход – требуемый список орфографических ошибок.

После обеда Джонсон собрал вместе пять имевшихся программ и словарь с непосредственным доступом, чтобы скомпоновать новое программное средство. Его программа была далека от совершенства, но продемонстрировала, что программу проверки правописания можно создать, и завоевала верных сторонников среди пользователей. Изменения, внесенные в течение нескольких следующих месяцев в программу, представляли собой небольшие модификации ее структуры; полной переделки пришлось ждать несколько лет.

### 13.2. ПРОСТРАНСТВО ВОЗМОЖНЫХ РЕШЕНИЙ

Перед тем как перейти к следующей версии программы проверки правописания spell, давайте рассмотрим имеющиеся у разработчика варианты. Сначала мы исследуем программу, как она представляется снаружи (спецификация задачи, как ее видит пользователь), а затем обратимся к внутренней структуре программы.

При охоте за "поддельными" словами может оказаться полезным знать источник их возникновения: орфографическая ошибка или опечатка. Малограмотные люди часто пишут не often, а oftun, грамотные иногда вместо occasionally пишут occaisionally, и все мы делаем "опечатки". При разработке программы можно учесть ошибки, которые пользователь делает чаще всего.

Есть два типа ошибок, которые может делать программа проверки правописания, и оба заставляют сомневаться в ее полезности. Пропуск слов, содержащих ошибки, это очевидный недостаток. А если программа принимает слишком много правильных слов за ошибочные, пользователь может не захотеть выискивать в грязи крупички золота. Все программы проверки правописания делают ошибки: выдавать слишком мало и слишком много подозрительных слов – трудный выбор при разработке.

Такая программа должна проверять правильность написания слова, но что такое слово? В программе Джонсона важным считалось удалить команды форматирования. В ней игнорировалось различие между строчными и прописными буквами, поэтому она осуществляла поиск правильно (The как the). Есть более тонкие проблемы, связанные со строчными и прописными буквами: DEC – название компании по производству компьютеров, Dec – месяц декабрь, а dec – это ошибка (кроме, конечно, случая, когда это сокращение от слова decimal –

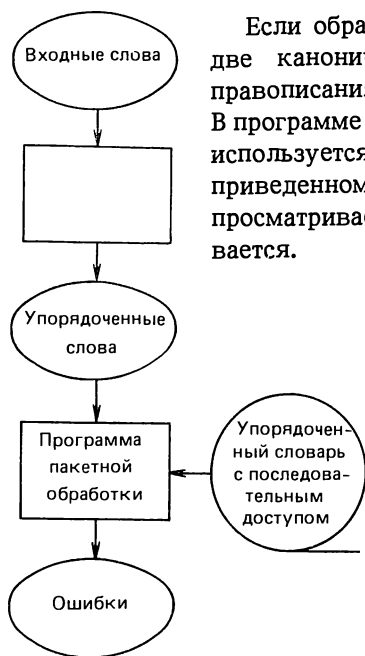


десятичный). К другим тонкостям, связанным со словами, относится обработка цифр, дефисов и апострофов (рассмотрите записи VAX-11/780 или his's). В прототипе программы этими тонкостями можно пренебречь, но готовый программный продукт должен быть разработан более тщательно.

А как быть со словарем вообще? Программа должна распознавать некоторые слова, отсутствующие в обычном словаре, такие как IBM или VLSI (СБИС) (но не vlsi). Однако больше возможностей – не всегда звучит лучше: из словаря может быть известно, что сего (восковина, покрывающая птичий клюв) – это что-то птичье, но в одном из моих файлов это, скорее, неправильное написание слова сге (забота). Аффикс – это приставка, например pre-, или суффикс, такой как -ly; в большинстве словарей анализ аффикса возлагается на пользователя. Хотя программа проверки правописания может с "чистой совестью доложить", что в слове antidisestablishmentarianism допущена ошибка, я бы разозлился, продираясь сквозь длинный список ошибок, подобных cats и gerplay, в то время как в словаре есть слова cat и play. Джонсон модифицировал свою программу, чтобы она обрабатывала обычные окончания -s и -ed, но товарный продукт должен делать более основательную работу по анализу аффиксов.

Вероятно, наиболее спорной проблемой при спецификации программы проверки правописания является то, что она должна делать в случае обнаружения ошибки. Простая программа Джонсона выдавала список неверно написанных слов. Диалоговая программа коррекции ошибок показывает пользователю слово с ошибкой в его контексте, спрашивает, заменить ли его на предлагаемое программой слово в этом месте, заменить ли слово в этом месте и во всех других местах, где оно появится, на предлагаемое программой слово, отредактировать ли это слово и заменить везде, где оно встретится, на отредактированный вариант и т. д.

Некоторые люди говорят, что они жить не могут без высококлассного корректора ошибок – малограмотным его советы кажутся особенно ценными. Как человека довольно грамотного, мое предпочтение отдано простым программам проверки правописания. Теперь при составлении всех документов я постоянно использую программу spell; раньше я редко пользовался высококлассным корректором ошибок, так как всякий раз, когда я пытался работать с ним, мне требовалось несколько минут, чтобы выучить заново его командный язык. Его советы часто бывали скорее раздражающими (или забавными), чем полезными: однажды мне было предложено, чтобы фамилия Tukey писалась, как Turkey (Турция). Кроме того, программу для исправления ошибок обычно гораздо труднее написать и сопровождать, чем программу проверки правописания.



Если обратиться от спецификации к реализации, то две канонические структуры программы проверки правописания – это наши старые друзья из разд. 5.2. В программе Джонсона, работающей в пакетном режиме используется структура, показанная на левом рисунке, приведенном ниже; программа справа оперативно просматривает каждое слово, как только с ним сталкивается.

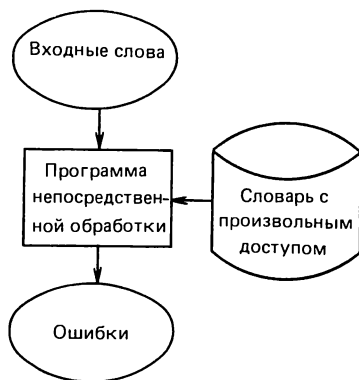


Рис. 13.2

В программе проверки правописания может использоваться любая структура, но диалоговая программа коррекции ошибок обычно ограничена структурой, в которой слова обрабатываются по мере поступления. Аналогично словарь с произвольным доступом могут использовать обе программы, но словарь с последовательным доступом пригоден только для программы, работающей в пакетном режиме.

Есть выбор между сложным анализом аффиксов и увеличением объема словаря. Для простых программ, не выполняющих анализ аффиксов, нужен огромный словарь, содержащий слова *test*, *tests*, *tested*, *tester*, *testing*, *retest*, *pretest* и т. д. При сложном анализе аффиксов объем данных сокращается, так как надо хранить единственную основу *test*, а также информацию о ее аффиксах. Однако такая схема исключает возможность использования словаря с последовательным доступом, потому что слово *predominant* надо искать на букву *d*, а слово *invisible* – на букву *v*.

Для хранения словаря имеется много способов. Если он помещается в ОЗУ, следует рассматривать использование структур, описанных в разд. 11.3. Методом доступа может быть хэш-таблица, двоичное дерево поиска или "цифровое дерево поиска" – структура, которая использует тот факт, что слово – это последовательность символов. Если словарь

должен быть расположен на диске, подходят Б-деревья и хэш-таблица на диске.

Выбор лучшей реализации словаря зависит от ряда факторов. Простые структуры данных легко разработать и сопровождать, но оценочные расчеты показывают, что в такой прикладной задаче "узким местом" может оказаться производительность. Предположим, что при поиске слова нужно выполнить две операции чтения с диска по 50 мс каждая. Такая программа обрабатывает 10 слов/с, так что на обработку документа из 4000 слов (средний размер главы) потребуется шесть долгих минут. Программа, которую мы изучим ниже, делает эту работу за 1/2 мин<sup>1</sup>. Программа коррекции орфографических ошибок, на которую приведена ссылка в разд. 13.6, совершает иногда до 200 обращений к диску, чтобы исправить единственное неправильно написанное слово, что соответствует 10 с раздражающего ожидания для пользователя, работающего в режиме оперативного контроля правописания.

Следующий план содержит наброски наших соображений по выбору из пространства возможных проектных решений; в задаче 4 упомянуты подходы к разработке программ проверки правописания, находящихся вне этого пространства:

**Требования** (точка зрения пользователя).

*Типичный пользователь.* Источник ошибок – орфографические ошибки и опечатки. Реакция на ошибки – грамотным людям требуется только предупреждение об ошибках, малограмотные ценят большую помощь.

*Ресурсы разработки.* Сколько времени можно затратить на программирование?

*Прикладные ресурсы.* Требования по времени и памяти к итоговой программе.

**Спецификация** (точка зрения пользователя).

*Определение слов.* Тонкие моменты состоят в обработке команд форматирования, распознавании строчных и прописных букв, вставленных в текст цифр и знаков препинания.

*Список слов.* Явно заданные слова (запоминаются в виде списка). Неявно заданные слова (образуются в результате анализа аффиксов).

---

<sup>1</sup> Приведу не совсем серьезное доказательство того, что эффективность имеет значение: написав эту главу, я запустил программу проверки правописания spell Макилроя для черновика, содержащего 2000 слов, и спустя 20 с был выдан список ошибок, содержащий слова monograph, textfile и filename. Я расправился с орфографической ошибкой в первом слове и заменил в нескольких местах textfile на filename для единообразия. Это была высокая отдача от вложенных в это дело 20 с; я, вероятно, не стал бы запускать эту программу, если бы ее работа заняла 3 мин.

*Реакция на ошибки.* Программы проверки сообщают об ошибках, программы коррекции их исправляют.

**Реализация** (точка зрения программиста).

*Структура программы.* Пакетные программы сортируют слова для удаления дубликатов. Программы, оперативно взаимодействующие с пользователем, проверяют каждое слово в момент его ввода.

*Реализация списка слов.* Компромисс между анализом аффиксов, объемом словаря и качеством ответов.

*Спецификация словаря.* Доступ к словам в упорядоченном виде или в произвольном порядке?

*Реализация словаря.* ОЗУ (хэширование, деревья поиска или произвольный порядок). Внешняя память (Б-деревья, хэширование). Объединенный метод (часто используемые слова – в ОЗУ, редко используемые – во внешней памяти).

### 13.3. ПРОГРАММА С ТОНКОСТЯМИ

В этом разделе мы рассмотрим программу проверки правописания Макилроя, написанную в 1978 г. Ее взаимодействие с пользователем аналогично программе Джонсона: введя с клавиатуры `spell <имя файла>`, вы получите список неправильно написанных слов в этом файле. К двум преимуществам этой программы перед программой Джонсона относятся лучший список слов и меньшее время работы. Я ей пользуюсь с энтузиазмом: работать с ней легко, она быстро сообщает обо всех моих ошибках в написании слов и выдает очень мало слов, которые на самом деле не являются ошибочными. В моем словаре жемчужина определена как "нечто отборное или драгоценное"; эта программа подходит под такое определение.

Первая проблема, с которой столкнулся Макилрой, состояла в подборе списка слов; чтобы оценить тонкости этой задачи, обратитесь к разд. 13.7. Он начал с поиска совпадающих слов полного словаря (чтобы обосновать выбор) со словарем Brown University Corpus, содержащим 1 млн слов (чтобы выбрать распространенные слова). Это было разумное начало, но оставалось еще много работы.

Подход Макилроя иллюстрируется его поиском соответствующих существительных, которые опущены в большинстве словарей. Сначала фамилии некоторых людей: 1000 наиболее часто встречающихся фамилий из большого телефонного справочника, список имен мальчиков и девочек, знаменитые имена (такие как Дейкстра и Никсон) и мифологические имена из каталога Bulfinch. Обнаружив такие неверные с точки зрения программы слова, как Хегох и Техасо, он добавил названия компаний из списка "500 наиболее удачливых". Так как в библиогра-

фических ссылках упоминаются издательские фирмы, они тоже были включены. Потом настал черед географических названий: страны и их столицы, штаты и их столицы, 100 крупнейших городов Соединенных Штатов и мира, океаны, планеты и звезды.

Он добавил также часто встречающиеся названия животных и растений, термины из химии, анатомии и (для местного употребления) вычислительной техники. Однако следил за тем, чтобы не включить слишком много слов: он не включал правильные слова, которые в обычном понимании похожи на ошибки (такие как геологический термин *swt*), и включал только одно из нескольких возможных правильных написаний (поэтому *traveling*, но не *travelling*).

Трюк Макилроя состоял в проверке выходной информации, выдаваемой программой *spell*, на реальных примерах. Он выявил все проблемы и получил всеобъемлющий вариант решения. Результатом был прекрасный список из 75 000 слов, он включает большинство слов, которыми я пользуюсь в своих документах, кроме того, обнаруживает мои ошибки в написании слов.

Анализ аффиксов в программе *spell* как необходим, так и удобен. Он необходим, поскольку список слов в английском языке не существует; программа проверки правописания для большого количества слов должна либо уметь формировать, например, такие слова, как *misrepresented*, либо сообщать, что они сформированы ошибочно. Анализ аффиксов имеет полезный побочный эффект – уменьшается размер словаря.

Целью анализа аффиксов является сведение слова *misrepresented* к слову *sent* за счет отбрасывания *mis-*, *re-*, *pre-* и *-ed*. В таблицах программы *spell* содержится 40 правил для префиксов и 30 правил для суффиксов. Специальный список из 1300 исключений приостанавливает правильные, но неверные случаи применения этих правил, такие как сведение *entend* (неправильное написание слова *intend*) к *en* плюс *-tend*. Такой анализ сокращает список из 75 000 слов до 30 000 слов.

Программа Макилроя совпадает с программой Джонсона до момента поиска слов в словаре (программа *compon* в предыдущем примере). Новая программа осуществляет цикл по каждому слову, отсекая аффиксы и производя поиск оставшейся части до тех пор, пока не будет найдено совпадение, либо пока аффиксов не останется (и слово будет объявлено ошибочным). Так как в результате обработки аффиксов может быть нарушена упорядоченность поступающих слов, доступ к словарю выполняется в произвольном порядке.

Оценка эффективности показывает важность хранения словаря в ОЗУ. Это было особенно тяжело для Макилроя, который первоначально писал свою программу для компьютера PDP-11 с адресным пространством 64 Кбайт. В реферате его статьи так резюмируется экономия памяти:

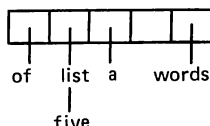
”Отбрасывание префиксов и суффиксов сократило список на 1/3 от его первоначального размера, хэширование на 60 % сократило то, что осталось, а сжатие данных сократило объем еще вдвое”. Таким образом список из 75 000 английских слов (и примерно еще столько же измененных форм) был представлен в виде 26 000 16-разрядных слов компьютера.

Макилрой использовал хэширование, чтобы представить 30 000 английских слов, отведя на каждое 27 битов (позже мы увидим, почему 27). Рассмотрим развитие схем представления на примере списка из пяти слов:

a list of five words

В первом методе хэширования используется хэш-таблица из  $N$  элементов, по длине примерно совпадающая со списком, и хэш-функция, которая представляет строку в виде целого числа в диапазоне от 1 до  $N$ . В таблице  $I$ -й элемент указывает на связанный список, содержащий все строки, отображенные в  $I$ . Если списки, не содержащие элементов, представить пустыми ячейками, а хэш-функцию определить так:  $H(a) = 3$ ,  $H(list) = 2$  и т.д., то таблица из пяти элементов могла бы выглядеть следующим образом:

Рис. 13.3



Чтобы найти слово  $W$ , мы выполняем последовательный поиск в списке, на который указывает ячейка  $H(W)$ .

В следующей схеме используется намного бóльшая таблица. При выборе  $N = 23$  имеется большая вероятность того, что большинство ячеек в хэш-таблице будут содержать только один элемент. В данном примере  $H(a) = 14$ , а  $H(list) = 6$ :

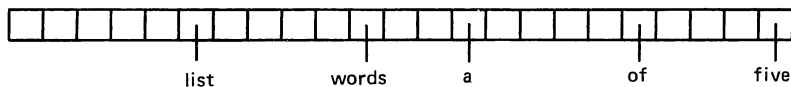


Рис. 13.4

В программе spell используется  $N = 2^{27}$  (примерно 134 млн) ячеек и все, за исключением немногих непустых списков, содержат только один элемент.

Следующий шаг вызывающе смел: вместо связанного списка слов Макилрой решил хранить только единственный бит на каждый элемент

таблицы. Это резко уменьшает память, но порождает ошибки. На рисунке использована та же хэш-функция, что и в предыдущем примере, а нулевые биты представлены пустыми ячейками.

					1				1			1					1				1
--	--	--	--	--	---	--	--	--	---	--	--	---	--	--	--	--	---	--	--	--	---

Рис. 13.5

Чтобы найти слово  $W$ , программа осуществляет доступ к биту с номером  $H(W)$  в этой таблице. Если этот бит установлен в 0, то программа сообщает, что слово  $W$  отсутствует в таблице. Если бит установлен в 1, то программа полагает, что слово есть в таблице. Иногда неверному слову соответствует правильный бит, но вероятность такой ошибки составляет только  $30\,000/2^{27}$  или примерно  $1/4000$ . Поэтому в среднем одно из каждых 4000 неверных слов "притворяется" хорошим. Макилрой отметил, что в типичном черновике редко содержится более 20 ошибок, так что этот дефект создает помеху не более чем в одном прогоне программы из каждой сотни прогонов – вот почему он выбрал число 27.

Для представления хэш-таблицы в виде строки из  $N = 2^{27}$  битов потребовалось бы свыше 16 млн байтов. Поэтому программа использует для представления только по одному биту, т. е. в вышеприведенном примере биты с номерами

6 11 14 19 23

Слово  $W$  считается присутствующим в таблице, если в ней имеется  $H(W)$ . Для очевидных вариантов представления таких величин используется 30 000 27-разрядных слов, но в адресном пространстве компьютера Макилроя имелось только 32 000 16-разрядных слов. Поэтому он упорядочил этот список и использовал код переменной длины для представления разностей между последовательными значениями хэш-функции. В предположении, что фиктивное начальное значение равно 0, вышеприведенный список уплотняется следующим образом:

6 5 3 5 4

Программа spell определяет эти разности, затрачивая в среднем 13.6 бита на каждую из них. При этом дополнительно высвобождается несколько сотен слов, которые используются в качестве указателей на некоторые ячейки, содержащие полезные начальные адреса в уплотненном списке, и поэтому ускоряют последовательный поиск. Результатом является словарь объемом 64 Кбайта с малым временем доступа и почти гарантированный от ошибок.

Мы уже рассмотрели два аспекта эффективности программы spell: она выдает полезную информацию и ей достаточно адресное пространство в

64 Кбайта. К тому же она быстро работает. В решении задачи 2 описано, как эта программа может проверить документ из 5000 слов за 30 с процессорного времени на компьютере VAX-11/750; значит, для проверки этой книги уйдет меньше 10 мин. Для проверки правильности написания одного слова я мог бы, например, написать

`spell`

`necessary`

`^d`

и примерно через 4 с я бы знал, что оно правильное (маленький словарь может быть быстро считан с диска). Статья, на которую приведена ссылка в разд. 13.6, содержит обзор многих программ проверки правописания, но ни одна не поднялась до такого уровня эффективности, как программа Макилроя.

#### 13.4. ОСНОВНЫЕ ПРИНЦИПЫ

Итак, в двух словах эта история такова. Имея хорошую идею, ряд мощных средств и свободную вторую половину дня, С. Джонсон создал полезную программу проверки правописания из шести строк. В уверенности, что эта проблема стоит существенных усилий, несколькими годами позже Д. Макилрой затратил несколько месяцев на разработку важной программы. Из этой истории можно извлечь несколько уроков.

*Прототипы.* Перед разработкой высококвалифицированной программы дайте потенциальным пользователям на многих входных данных поэкспериментировать с ее простым прототипом. Джонсон использовал тривиальный список слов, чтобы создать медленную программу проверки правописания; пользователи хотели иметь лучший список слов и большую скорость, но не было требований сделать программу коррекции ошибок. Прототипы могут помочь оценить параметры окончательного продукта: эксперименты с программой Джонсона дали Макилрою понятие о том, сколько всего слов в типичном документе, сколько различных и неправильно написанных слов.

*Важность разбиения на фрагменты.* Хорошо построенные системы разделяются на независимые компоненты, каждый из которых хорошо выполняет работу одного типа. Пять различных программ в конвейере Джонсона решают пять различных задач; любая из них могла бы быть расширена без вредного воздействия на другие. Анализ аффиксов и представление словаря в программе Макилроя в большей степени независимы друг от друга; им не надо много знать друг о друге, чтобы работать совместно. По моему мнению, важность разбиения опять-таки свидетельствует против программ исправления ошибок: ошибки следу-



ет находить с помощью простой программы проверки, исправлять их – работа редактора текста, а помощь в правильном написании слова должна исходить от ”подсказчика” (см. задачу 5).

*Простота.* Наилучшая конструкция – обычно самая простая. Постановку задачи Джонсона легко специфицировать, а его программа ясно описана в нескольких строках команд. Даже в структуре данных с тонкостями, разработанной Макилроем, простота приносит свои плоды: единая структура его словаря выполняет функцию, которая во многих программах, использующих несколько структур данных, выполняется гораздо медленней.

*Разработка программного обеспечения.* Хотя здесь не потребовалось никакого формального управления разработкой, программа spell – разработка высшего класса. Джонсон и Макилрой использовали проверенные инженерные средства: создание прототипов, простоту, разбиение на фрагменты, тщательную постановку задачи и оценки. Они использовали стандартные компоненты: готовые фильтры для исключения команд форматирования, сортировки и удаления дубликатов. Когда не могли воспользоваться существующими программными средствами, они применяли испытанные методики: в структуре данных для списка слов объединяются хэширование, алгоритмы аппроксимации и сжатие данных. Итоговая программа начинена искусными проектными решениями: Макилрой принял компромиссные решения по времени работы, объему памяти, точности и постановке задачи, чтобы получить в целом эффективное программное средство.

### 13.5. ЗАДАЧИ

1. Соберите по документам и словарям такие данные, как распределение длины слов и частоты всех букв и диграмм (пар букв). Для словарей оцените сжатие, достигаемое с помощью простого анализа аффиксов (какой процент слов охватывается окончаниями -s, -ed и -ly)? Для документов подсчитайте общее число слов, число различных слов и число неверно написанных слов. Какие другие статистические данные полезно знать?
2. Сделайте оценки различных вариантов разработки программы проверки правописания. (Например, следует ли использовать быстрый фильтр для отбора сотни наиболее употребительных английских слов? Желательно ли сортировать слова для того, чтобы устранить дубликаты в пакетной программе, использующей быстрый словарь с оперативным доступом?) Охарактеризуйте время работы программы проверки правописания в своей системе.
3. Изобретите другие структуры данных для словарей с произвольным доступом. Особое внимание уделите структурам, которые не всегда

- дают правильный ответ. Проанализируйте затраты памяти и время работы (как на чтение словаря с диска, так и на доступ к слову).
4. Придумайте программы проверки правописания, которые не используют полный словарь.
  5. Разработайте "подсказчик", который совместно с программой проверки правописания могли бы использовать малограмотные люди. Если введено слово *ossigance*, он должен "подсказать", что вы имели в виду *ossigence*.
  6. Программам для проверки правописания, для игр со словами, для составления кроссвордов и головоломок нужны разные словари. Приведите слова, которые могли бы быть в одном из словарей, но не содержаться в других. Какие другие словари могли бы потребоваться различным программам?
  7. Обсудите, как разработать программу проверки правописания для других языков, например для русского.
  8. Обсудите спецификацию и реализацию других программ, которые могли бы оказаться полезными для людей, составляющих и хранящих свои документы с помощью компьютеров.
  9. Разработайте программу для нахождения *K* наиболее часто встречающихся во входном файле слов. Типичное значение *K* около 100.
  10. Ошибки, присущие аппроксимации словаря Макилроя, приемлемы для его программы проверки правописания – ошибочное слово не распознается только в 1 % ее прогонов. А какое значение это может иметь в других программах, таких как "подсказчик" из задачи 5, или в программах, "играющих" в слова?

### 13.6. ЛИТЕРАТУРА ДЛЯ ДОПОЛНИТЕЛЬНОГО ЧТЕНИЯ

Детали программы Макилроя *spell* приведены в его статье "Разработка списка слов для проверки правописания" (McIlroy. Development of Spelling List в сборнике IEEE Transactions on Communications COM-30, 1 за январь 1982 г., с. 91 – 99). Это очаровательная и восхитительная работа, и каждый, кто серьезно изучает программирование, должен обязательно прочесть ее.

Статья "Компьютерные программы для обнаружения и исправления ошибок" Дж. Л. Питерсона (James L. Peterson. Computer programs for detecting and correcting spelling errors), опубликована в декабрьском номере журнала Communications of the ACM за 1980 г. Первая часть этой статьи содержит обзор задач по проверке правописания и различных реализаций программ проверки и исправления. Потом описывается программа исправления орфографических ошибок, которую Питерсон разработал и реализовал; полностью программа на Паскале опубликована в монографии, выпущенной издательством Springer-Verlag.

## (ДОПОЛНЕНИЕ)

Рецепт изготовления жаркого из слона начинается так: "Сначала поймите слона." Если ваш рецепт создания программы проверки правописания начинается так: "Сначала найдите правильное английское слово", то, может быть, вы найдете, что проще приготовить восхитительное жаркое из слона. Прочитав черновик этой главы, В. Висоцки написал следующее примечание, которое помогло мне оценить эту задачу.

"Проверка правописания – это один из лучших примеров, которые я видел, показывающих необходимость разработки прототипов: сделайте нечто незначительное, поработайте с ним, посмотрите, насколько это полезно на практике, потом модифицируйте и расширяйте. Если вы обратили внимание, априори почти невозможно угадать в деталях, какие именно особенности должна иметь программа проверки правописания, чтобы быть более полезной.

Это связано с тем фактом, что мы имеем дело с английским языком. Французский язык, например, более академичен при определении корней слов и имеет более систематический набор правил. На французском языке намного легче, чем на английском, определить, является ли словом *glotchification* и правильно ли оно написано. Таким образом, намного проще было бы написать программу проверки правописания для французского языка, чем для английского. Но она была бы гораздо менее полезна для французского языка, так как каждый, много пишущий на французском, знает, как писать слова правильно (и как определять границы применимости слов, и как решать, вероятен ли данный неологизм).

Язык имеет бросающее нам вызов свойство изменяться со временем: *fribble* (бездельник) – это слово и оно есть в словаре Уэбстера, но моя дочь была бы удивлена определением, которое дает словарь: "*glotch* является словом и все знают, что оно означает"<sup>1</sup>, но его нет в этом словаре, т. е. *glout* – это, очевидно, не слово вопреки словарю Уэбстера.

Мне кажется, что изменчивость английского языка является глубоко фундаментальной причиной, по которой программа исправления написания слов не будет работать. Моя учительница в пятом классе была таким "корректором написания слов", и в вашем черновике она почти наверняка исправила бы слова *newline* и *online*, а также *filename*, ни одно из которых на самом деле не нуждается в исправлении. Я еще

---

<sup>1</sup> Я, например, не знаю (*glotch* – это большая, беспорядочная совокупность).

помню строгость ее наказания, когда я предложил добавить слово *abaft* к ее каноническому списку всех предлогов английского азыка. К несчастью для всех корректоров, как электронных, так и живых, английский язык (и написание слов) основывается на соглашениях между пользователями, а не на решениях, принимаемых академией или экспертами.

Это именно то, что делает программу проверки правописания такой интересной в реализации и таким хорошим примером в вопросах разработки программ”.

## ЭПИЛОГ

Кажется, что лучшим заключением этой книги было бы интервью с автором, так что начнем.

*Вопрос:* Спасибо, что согласились дать это интервью.

*Ответ:* Ну что Вы, мое время – это ваше время.

*Вопрос:* Почему Вы стремитесь собрать эти главы в отдельную книгу, ведь они уже опубликованы в *Communications of the ACM*?

*Ответ:* На это есть несколько причин: я обнаружил целую дюжину ошибок, внес сотни небольших исправлений и добавил ряд новых разделов. В книге стало в 1,5 раза больше задач, решений к ним и рисунков. Кроме того, удобней, чтобы эти главы были в одной книге, чем в десятке журналов. Однако основная причина в том, что темы, излагаемые в этих главах, легче понимать, когда они собраны вместе (целое больше, чем сумма его отдельных частей).

*Вопрос:* Что это за темы?

*Ответ:* Наиболее важная состоит в том, что напряженные размышления на темы программирования могут быть и полезными, и приятными. Работа программиста – это нечто большее, чем регулярная разработка программ по формальным требованиям. Если эта книга поможет хотя бы одному разочаровавшемуся программисту полюбить опять свою работу, она выполнит свою задачу.

*Вопрос:* Это довольно расплывчатый ответ. Имеется ли какая-нибудь общая идея, объединяющая эти главы?

*Ответ:* Эффективность – это тема ч. II и всех глав. В ряде глав широко используется верификация программ. В приложении приведен каталог алгоритмов этой книги.

*Вопрос:* Такое впечатление, что в большинстве глав основное внимание уделяется процессу разработки. Можете ли Вы подытожить свои советы в этой области?

*Ответ:* Я рад, что Вы спросили об этом. Я случайно подготовил список перед этим интервью. Вот некоторые моменты, которые неоднократно упоминались в разделах "Основные принципы".

Работайте над правильно поставленной задачей.

Исследуйте спектр возможных решений.

Обращайте внимание на данные.

Используйте оценки.

Используйте модульную структуру.

Создавайте прототипы.

Идите на компромиссы, когда это необходимо.

Стремитесь к простоте.

Эти моменты первоначально рассматривались применительно к программированию, но они применимы в любой инженерной деятельности.

*Вопрос:* Тут возникает вопрос, который меня волнует: несложно упростить маленькие программы, приведенные в этой книге, но распространяются ли эти методики на реальное программное обеспечение?

*Ответ:* У меня есть три ответа: "да", "нет" и "может быть". Да, они распространяются; например, в разд. 3.4 описан огромный программный проект, который был упрощен "всего только" до 80 чел.-лет. Настолько же банален ответ "нет": если упрощение выполнено должным образом, Вы избегаете создания громоздких систем и эти методики не требуют распространения на такие системы. И хотя оба эти взгляда заслуживают внимания, истина лежит посередине, и отсюда возникает "может быть". Некоторые программные системы должны быть большими и материал этой книги иногда можно применить к таким системам. Система UNIX является замечательным примером мощного средства, которое построено из простых и элегантных частей.

*Вопрос:* Вы рассказывали и о других системах, разработанных в фирме Bell Labs. Не является ли эта книга несколько ограниченной?

*Ответ:* Может быть, слегка. Я тяготел к материалу, который видел в практической работе, и среда, в которой я работал, оказала влияние на книгу. Большая часть материала книги – это вклад моих коллег, и они заслуживают доверия (или упрека). Я научился многому у исследователей и разработчиков, работающих в фирме Bell Labs. Здесь замечательная атмосфера общего дела, которая способствует взаимодействию между

исследованиями и разработкой. Поэтому многое из того, что Вы называли ограниченностью, всего лишь патриотизм по отношению к моей фирме.

*Вопрос:* Давайте вернемся обратно на землю. Почему Вы используете так много различных языков? В книгу напиханы AWK, Бейсик, Си, Кобол, Фортран и Ваш собственный странный псевдоязык на основе Паскаля. Почему бы не выбрать один наилучший язык и придерживаться его?

*Ответ:* Идеи этой книги не ограничены каким-либо одним языком программирования. Программисты должны учиться думать на удобном псевдоязыке, а затем выражать свои идеи на языке, используемом для реализации. Д. Гриз выразил это лучше всех: "программа на языке, а не в нем".

*Вопрос:* Что не вошло в эту книгу?

*Ответ:* Программа проверки правописания из гл. 13 – самая большая программа, описанная в деталях. Я надеялся включить большую систему, составленную из многих программ, но я не смог описать ни одну интересную систему на десяти страницах, составляющих типичную главу. Читатели, интересующиеся такими системами, должны просмотреть раздел "Case Studies" в журнале Communications of the ACM, впервые появившийся в июльском номере за 1984 г. Вообще говоря, мне бы хотелось в дальнейшем в журнале CACM обратиться к темам "теория вычислительных машин и систем для программистов" (подобно темам верификации программ из гл. 4 и разработки алгоритмов гл. 7) и "инженерные методики в вычислительной технике" (подобно оценкам, изложенным в гл. 6).

*Вопрос:* Если Вы употребляете такие термины, как "теория", "инженерный", то почему в этих главах так мало теорем и таблиц и так много всяких историй?

*Ответ:* Будьте осторожны – люди, берущие интервью сами у себя, не должны критиковать стиль письма.

Приложение.

## КАТАЛОГ АЛГОРИТМОВ

Эта книга содержит большую часть материала по алгоритмам, изучаемого в колледже, но в другом аспекте: основное внимание уделено не математическому анализу, а прикладным задачам и написанию программ. В данном приложении этот материал изложен более привычным образом.

*Постановка задачи.* Выходная последовательность – это упорядоченная перестановка входной последовательности. Если на входе – файл на диске, на входе как правило, другой файл на диске; если на входе – массив, на выходе, как правило, тот же самый массив.

*Приложения.* Приведенный ниже список только дает намек на разнообразие применений сортировки.

*Требования к выходной информации.* Некоторые пользователи желают иметь упорядоченную выходную информацию; см. разд. 1.1 и рассмотрите с этой точки зрения свой телефонный справочник и ежемесячную проверку счетов. Для подпрограмм, аналогичных двоичному поиску, требуется упорядоченная входная информация.

*Сбор элементов с одинаковыми значениями.* Программисты используют сортировку, чтобы собрать вместе равные элементы последовательности: программа проверки правописания Джонсона, описанная в разд. 13.1, осуществляет сбор всех слов документа, а программа для поиска анаграмм из разд. 2.4 и 2.8 группирует слова, относящиеся к одному классу анаграмм. Упорядоченные входные файлы в разд. 5.2 и 13.1 позволяют обрабатывать одинаковые элементы слиянием (см. также задачи 6 гл. 2, 4 гл. 7 и 4 гл. 13).

*Другие прикладные задачи.* В программе поиска анаграмм в разд. 2.4 и 2.8 сортировка используется для канонического упорядочения букв в слове и, следовательно, для построения сигнатуры данного класса анаграмм. В задаче 7 гл. 2 сортировка используется для перепорядочения данных на ленте.

*Подпрограммы общего назначения.* Следующие подпрограммы сортируют произвольную последовательность из  $N$  элементов.

*Сортировка методом вставок.* Время работы программы из пяти строк, приведенной в разд. 10.1, составляет  $O(N^2)$  для худшего случая и для случайных входных данных. В разд. 10.2 она используется для сортировки почти упорядоченного массива за время  $O(N)$ . Из программ, приведенных в этой книге, она является единственной стабильной сортировкой: на выходе элементы с одинаковыми ключами имеют тот же относительный порядок, что и на входе. В решении задачи 10 гл. 8 содержится реализация этой программы на языке Си.

*Программа быстрой сортировки.* Программа быстрой сортировки из 12 строк, приведенная в разд. 10.2, имеет ожидаемое время работы  $O(N \log N)$  для массива из  $N$  различных элементов. Она рекурсивна и

имеет логарифмическую зависимость среднего размера стека от числа элементов. В худшем случае ей требуется время  $O(N^2)$  и стек длиной  $O(N)$ . Время ее работы для массива с одинаковыми элементами  $O(N^2)$ . Для варианта из 15 строк в задаче 3 гл. 10 среднее время работы  $O(N \log N)$  для любого массива (в задаче 11 гл. 10 дается альтернативная схема). В задаче 13 гл. 10 предлагается лучший выбор элемента, относительно которого осуществляется разбиение массива. В таблице, приведенной в разд. 10.3 и решениях задач 10 гл. 8 и 10 гл. 10, представлены эмпирические данные о времени работы программы быстрой сортировки. В решении задачи 10 гл. 8 содержится ее реализация на языке Си.

*Пирамидальная сортировка.* Пирамидальная сортировка из разд. 12.4 выполняется за время  $O(N \log N)$  для любого  $N$ -элементного массива. Она нерекурсивна и использует дополнительную память фиксированного объема. В решениях задач 1 гл. 12 и 2 гл. 12 описана более быстрая программа пирамидальной сортировки всего из 14 строк.

*Сортировка слиянием.* Алгоритм, набросок которого приведен в разд. 1.3, эффективен для сортировки файлов на диске и на ленте. Набросок алгоритма слияния дан в задаче 3 гл. 12.

Время работы программ сортировки методом вставок и быстрой сортировки сравниваются в разд. 10.3; время работы других алгоритмов сортировки обсуждается в решении задачи 3 гл. 1

*Алгоритмы специального назначения.* Эти алгоритмы приводят к коротким и эффективным программам для некоторых входных данных.

*Поразрядная сортировка.* Программа сортировки битовых строк Макилроя в задаче 6. гл. 10 может быть обобщена на случай сортировки строк для алфавитов большого объема (например, байтовых).

*Сортировка битовых строк.* В программе сортировки битовых строк, приведенной в разд. 1.4 и состоящей из семи строк, используется тот факт, что целые числа, которые надо сортировать, имеют небольшой диапазон значений, не имеют дубликатов и с ними не связано никаких дополнительных данных. Подробности реализации и дальнейшее развитие метода описаны в задачах 2 и 5 гл. 1.

*Другие алгоритмы сортировки.* При многопроходной сортировке из разд. 1.3 многократно считываются входные данные, т. е. жертвуется время во имя экономии памяти. В разд. 11.1 описана задача формирования упорядоченного списка случайных целых чисел; программы приведены в разд. 11.2 и 11.3.



*Постановка задачи.* Подпрограмма поиска определяет, является ли заданный ей на входе элемент членом данного множества и, возможно, выдает связанную с ним информацию. В статических прикладных задачах это множество известно до начала выполнения любой операции поиска; в динамических прикладных задачах элементы могут добавляться к множеству и удаляться из него.

*Прикладные задачи.* В телефонном справочнике Леска из задачи 6 гл. 2 поиск осуществляется для преобразования закодированного имени в номер телефона. Программа эндшпиля Томпсона, описанная в разд. 9.7, осуществляет поиск в множестве расстановок фигур, чтобы найти оптимальный ход. Программа проверки правописания Макилроя из разд. 13.3 осуществляет поиск в словаре, чтобы определить, правильно ли написано слово. Дополнительные прикладные задачи описаны вместе с подпрограммами.

*Подпрограммы общего назначения.* Следующие подпрограммы выполняют поиск в произвольном массиве из  $N$  элементов.

*Последовательный поиск.* Программа для поиска в массиве дана в задаче 9 гл. 4; более быстродействующая версия приведена в задаче 5 гл. 8. Этот алгоритм используется при разбиении слов (задача 5 гл. 3), аппроксимации географических данных (разд. 8.2), представлении разреженных матриц (разд. 9.2), генерации случайных последовательностей (разд. 11.3) и для упаковки "карманов" (задача 4 гл. 12). Во введении к гл. 13 и в задаче 1 гл. 3 описаны две "дурацкие" программы, реализующие последовательный поиск.

*Двоичный поиск.* Алгоритм для поиска в упорядоченном массиве, выполняющий  $O(\log N)$  сравнений, описан в разд. 2.2, а программа для него разработана в разд. 4.2. В разд. 8.3 эта программа расширена, чтобы она могла находить первый из нескольких равных элементов, и повышена ее производительность; более эффективная структура данных описана в задаче 6 гл. 12. К приложениям этого метода относятся поиск анаграмм слова, заданного на входе (задача 1 гл. 2), телефонных номеров (задача 6 гл. 2), положения точки на отрезках (задача 7 гл. 4), индекса элемента в разреженном массиве (разд. 11.3). В прикладных задачах, перечисленных в задаче 3 гл. 12, используется двоичный поиск на структуре данных типа "пирамида". В задачах 9 гл. 2 и 7 гл. 8 обсуждаются компромиссы, связанные с двоичным и последовательным поиском.

*Другие методы.* В задаче 9 гл. 1 используется хэширование телефонных номеров, в разд. 11.3 хэшируется множество целых чисел, а в

разд. 13.3 хэшируются слова в словаре (см. также задачу 3 гл. 13). В разд. 11.3 кратко описаны двоичные деревья поиска. Как увеличить скорость выполнения этих программ рассказывается в задаче 6 гл. 8.

*Алгоритмы специального назначения.* Эти алгоритмы приводят к коротким и эффективным программам для некоторых типов входных данных.

*Индексирование по ключу.* Некоторые ключи можно использовать в качестве индексов в массивах, содержащих значения. Ключи в качестве индексов использовались в задачах с номерами избирательных участков (разд. 1.4), с номерами курсов в колледже (задача 7 гл. 1), с символами (разд. 8.2), с аргументами тригонометрических функций (задача 9 гл. 8), с индексами в разреженных массивах (разд. 9.2), со значениями счетчика команд (задача 7 гл. 12), с расстановкой фигур на шахматной доске (разд. 9.7), со случайными целыми числами (разд. 11.3) и со значениями в очередях с приоритетами (задача 7 гл. 12). В задаче 4 гл. 8 уменьшен объем памяти за счет использования индексирования по ключу и численных функций.

*Другие методы.* В разд. 8.1 рассказывается, как было уменьшено время поиска за счет расположения часто встречающихся элементов в хэш-таблице. В разд. 9.1 описано, как упростился поиск в таблице налогов, после того, как был понят контекст задачи.

#### ДРУГИЕ АЛГОРИТМЫ ДЛЯ РАБОТЫ С МНОЖЕСТВАМИ

В этих задачах обрабатываются наборы из  $N$  элементов, в которых, возможно, имеются дубликаты.

*Выбор.* В задаче 8 гл. 2 мы должны выбрать  $K$ -й по величине элемент множества. В решении задачи 9 гл. 10 приведен эффективный алгоритм для этой задачи; более медленные алгоритмы упомянуты в задачах 8 гл. 2, 1 гл. 10 и 3 гл. 12.

*Очереди с приоритетами.* Посредством очередей с приоритетами реализуются множества элементов, на которых определены операции вставки произвольных элементов и удаления минимального элемента. В разд. 12.3 описаны две последовательные структуры для этой задачи. Пирамиды, рассмотренные в этом разделе, особенно эффективны. Прикладные задачи описаны в задачах 3, 4 и 7 гл. 12. В решении задачи 10 гл. 12 приведена программа на языке Си++.

#### АЛГОРИТМЫ ДЛЯ ВЕКТОРОВ И МАТРИЦ

Алгоритмы для взаимной перестановки подвекторов вектора рассматриваются в разд. 2.3 и в задачах 3 и 4 гл. 2; в решении задачи 3 гл. 2

содержится программа для этих алгоритмов. Программа для вычисления максимума вектора описана в задачах 9 гл. 4 и 6 гл. 8. В задаче 7 гл. 2 для транспортировки матрицы, записанной на ленте, используется сортировка. Алгоритмы для векторов и матриц, в которых для хранения различных данных используется одна и та же память, приведены в разд. 9.2 и 12.4. Разреженные вектор и матрицы рассматриваются в разд. 3.1, 9.2 и 13.3; в задаче 8 гл. 1 описана схема для установки начальных значений векторов, которая используется в разд. 11.3. В гл. 7 приведены пять алгоритмов для вычисления подвектора с максимальной суммой элементов, в нескольких задачах гл. 7 фигурируют векторы и матрицы.

### СЛУЧАЙНЫЕ ОБЪЕКТЫ

Подпрограммы для генерирования псевдослучайных целых и действительных чисел определены и использованы в разд. 10.2 и 11.1. В разд. 11.3 описан алгоритм для "перетасовки" элементов массива. В разд. 11.1 – 11.3 приведен ряд алгоритмов для выбора случайных подмножеств (см. также задачу 6 гл. 11); в задаче 4 гл. 1 показано применение такого алгоритма.

### ДРУГИЕ АЛГОРИТМЫ

В разд. 7.9 дан обзор влияния алгоритмов на задачи численного анализа, теории графов и геометрии.

*Численные алгоритмы.* В решении задачи 3 гл. 2 представлен аддитивный алгоритм Евклида для вычисления наибольшего общего делителя двух целых чисел. В задаче 9 гл. 4 дана программа для эффективного алгоритма возведения числа в целую положительную степень. В задаче 9 гл. 8 тригонометрическая функция вычисляется путем просмотра таблицы. В решении задачи 11 гл. 8 приведен метод Хорнера для оценки полинома. Суммирование большого количества чисел с плавающей точкой описано в задачах 1 гл. 10 и 3 гл. 12.

*Базы данных.* В разд. 3.4 дан набросок базы данных с взаимосвязью объектов, использованной в большой программной системе. В задаче 9 гл. 1 описано, как хэширование применяется в базе данных универмага, реализованной с использованием бланков на бумаге.

### ПОДСКАЗКИ ДЛЯ НЕКОТОРЫХ ЗАДАЧ

#### ГЛАВА 1

1. Рассмотрите 2-проходный алгоритм.
4. Прочитайте гл. 11.
- 5, 7, 8. Попытайтесь использовать индексирование по ключу.

9. Рассмотрите хэширование и не ограничивайтесь компьютеризованной системой.
10. Это задача для птиц.

## ГЛАВА 2

1. Подумайте о сортировке, двоичном поиске и сигнатурах.
2. Постарайтесь получить алгоритм, время работы которого линейно.
4. Рассмотрите, какое воздействие оказывает постраничная организация.
5. Воспользуйтесь тождеством  $SBA = (A^R B^R C^R)^R$ .
7. Висоцки использовал системную программу и две простые программы для изменения формата данных на лентах.
8. Затраты на выполнение  $S$  последовательных поисков пропорциональны  $SN$ ; общие затраты на  $S$  двоичных поисков равны затратам на сами поиски плюс время, необходимое для упорядочения таблицы. Прежде чем полностью довериться коэффициентам в различных алгоритмах, рассмотрите задачу 7 гл. 8.
10. Как Архимед определил, что королевская корона не из чистого золота?

## ГЛАВА 3

2. Один массив используйте для представления коэффициентов рекуррентного соотношения, а другой — для представления  $k$  предыдущих значений. Эта программа состоит из цикла внутри другого цикла.
4. Только одна программа должна быть написана с самого начала, две другие могут использовать ее как подпрограмму.
6. Программа для решения этой задачи на АWK состоит только из 12 строк. Она не соответствует наброску, приведенному в разд. 3.2, поскольку в ней используются массивы, входные поля и средства подстановки строк, имеющиеся в АWK.

## ГЛАВА 4

2. Ведите работу из точного инварианта. Рассмотрите добавление к массиву двух фиктивных элементов, что поможет вам установить начальное значение инварианта:  $X[0] = -\infty$  и  $X[N+1] = \infty$ .
5. Если вы решили эту задачу, ждите ближайшего Ученого совета и требуйте докторской степени по математике.
6. Найдите инвариант, сохраняемый этим процессом, и свяжите начальное состояние банки с ее конечным состоянием.
7. Снова прочитайте разд. 2.2.

9. Попробуйте воспользоваться следующими инвариантами цикла, которые истинны непосредственно перед выполнением оператора while: для сложения векторов

$$I \leq N + 1 \text{ и } \forall_{1 \leq j \leq I-1} A[j] = B[j] + C[j]$$

и для последовательного поиска

$$I \leq N + 1 \text{ и } \forall_{1 < j \leq I-1} X[j] \neq T.$$

## ГЛАВА 5

2. Попробуйте заняться игрой Конвея "Жизнь" на следующих стадиях разработки. Постановка задачи: каков размер поля? что является выходом программы? Алгоритмы: можете ли вы найти способ вести учет активной в текущий момент части поля? можете ли вы обнаруживать циклы? Структуры данных: должно ли поле быть представлено разреженным массивом, обычным двумерным массивом или их комбинацией? Оптимизация программы и оборудования: я видел написанную в виде микропрограммы игру "Жизнь", работающую со скоростью урагана.
7. Автомобильные аварии устраняются такими мерами, как подготовка водителей, строгие требования к выполнению ограничения скорости, введение возрастных ограничений на употребление спиртного, строгие наказания за управление автомобилем в состоянии опьянения и хорошая система общественного транспорта. Если авария случается, ущерб для пассажиров можно уменьшить соответствующей конструкцией пассажирского салона, употреблением ремней безопасности (возможно, это должно регламентироваться законом) и наполняемыми воздухом баллонами. Если ущерб причинен, его последствия могут быть уменьшены парамедиками на месте происшествия, быстрой эвакуацией с помощью вертолетов "скорой помощи", в травматологических центрах и восстановительной хирургией.

## ГЛАВА 6

10. Некоторые люди боятся называть цифры ("У меня нет мыслей по поводу того, какова глубина Миссисипи"). Другая крайность, есть люди, которые охотно выдают числа с нереальной точностью (глубина реки 31.415926535 футов").

## ГЛАВА 7

- 4, 5, 6. Используйте кумулятивный массив.
7. Очевидный алгоритм имеет время работы  $O(N^4)$ ; постарайтесь найти

кубический алгоритм.

8. В дополнение к вычислению максимальной суммы в области давайте на выходе информацию о максимальных векторах, оканчивающихся с каждой стороны области.

## ГЛАВА 8

1. Предположите, что каждый байт состоит из восьми битов; как вы можете их использовать?
2. Какова взаимосвязь между подмассивами `CountTable[0...7]` и `CountTable[8...15]`?
4. Объедините и подберите функции и таблицы.
7. Чтобы сделать двоичный поиск при малых  $N$  конкурентоспособным с последовательным, сделайте операцию сравнения очень "дорогой" (см., например, задачу 7 гл. 4).

## ГЛАВА 9

1. Какие коды сгенерирует компилятор для доступа к упакованным полям?
4. Закодировать повторяющиеся элементы.
7. Сократить объем данных, считая некоторые области памяти эквивалентными. Такие области могли бы быть либо блоками постоянной длины (например, по 64 байта), либо совпадать с границами подпрограмм.

## ГЛАВА 10

4. Обеспечьте продвижение индекса цикла  $I$  справа налево так, чтобы он приближался к известному значению  $T$  для  $X[L]$ .
5. Если вам требуется решить две подзадачи, какую вы должны решить тотчас же, а какую отложить, чтобы вернуться к ней позже, — бóльшую или меньшую?
9. Модифицируйте программу быстрой сортировки так, чтобы она осуществляла рекурсию только в подобласти, содержащей  $K$ .

## ГЛАВА 11

3. Ступайте к статистику и воспользуйтесь фразами "задача о сборе купонов" и "задача о дне рождения".
8. В задаче говорится, что вы можете использовать компьютер, но в ней не говорится, что вы должны это сделать.

1. Попробуйте использовать для этой структуры пирамидальную сортировку.



Рис. П.1

2. Обратитесь к задаче 1, а также рассмотрите вынесение команд из цикла.
5. Пирамиды содержат неявные указатели от узла  $I$  к узлу  $2I$ ; попробуйте сделать то же самое в файлах на диске.
6. При двоичном поиске в массиве  $X[1 \dots 7]$  используется неявное дерево, корень которого находится в  $X[4]$ . Каким образом можно вместо этого использовать неявные деревья, описанные в разд. 12.1?
8. Используйте сортировку с наименьшим значением  $O(N \log N)$ . Если бы время работы процедур `Insert` и `ExtractMin` было меньше, чем  $O(\log N)$ , то вы смогли бы выполнить сортировку за время, меньшее чем  $O(N \log N)$ ; покажите, как.

4. К возможным подходам относятся обнаружение "близких" ошибок (таких как слова `programmer` и `programeer` в одном документе) и проверка часто встречающихся случаев удвоения и утроения букв (см. задачу 1).
9. Что общего у этой задачи с задачей проверки правописания?

## РЕШЕНИЯ НЕКОТОРЫХ ЗАДАЧ

### РЕШЕНИЯ К ГЛ.1

1. Двухпроходный алгоритм сначала сортирует целые числа от 1 до 13 500, используя 13 500/16-844 слова в памяти, а при втором проходе сортирует числа с 13 501 по 27 000.  $K$  = проходной алгоритм сортирует максимально  $N$  неповторяющихся целых чисел в диапазоне  $1 \dots N$  за время  $KN$  и использует память объемом  $N/K$ .
3. Системная программа сортировки (хорошо оптимизированная) работает 65 с на компьютере VAX-11/750, чтобы упорядочить файл, содержащий 26 000 различных чисел от 1 до 27 000. Первый вариант реализованной мной программы сортировки с использованием побитового представления потребовал 41.6 с для тех же самых данных: 32 с на ввод-вы-

вод и 9.6 с на вычисления. Оптимизация программы сортировки с помощью принципов, изложенных в гл. 8, сокращает время работы до 14 с: 12.3 на ввод-вывод и 1.7 на вычисления.

4. (См. гл. 11, особенно задачу 7.) В нижеприведенной программе предполагается, что выходом функции RandInt (A, B) является случайное целое число от A до B.

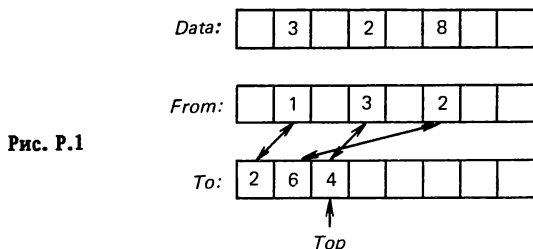
```

for I := 1 to N do
  X[I] := I
for I := 1 to K do
  Swap(X[I], X[RandInt(I,N)])
print X[I]

```

Функция RandInt обсуждается в разд. 10.2.

5. Если каждое целое число появляется максимально 10 раз, мы можем отвести под счетчик количества его появлений половину байта, т. е. 4 бита. Применяя это решение к задаче 1, мы можем упорядочить весь файл за один проход, использовав  $27\,000/2$  байтов, или за K проходов, использовав  $27\,000/2K$  байтов.
7. Информация о курсе может быть представлена 27 000 байтами в ОЗУ, если для каждого из всех возможных номеров курсов хранить число имеющихся мест в виде двух байтов. Если число из четырех цифр не является правильным номером курса, то в таблице из 10 000 элементов в его позиции запоминается специальное значение (например, -1). Такое изменение сокращает время работы предложенной программы с нескольких часов до нескольких минут.
8. Эффект от установки начальных значений вектора Data [1 . . N] может быть достигнут с помощью сигнатуры, содержащейся в двух дополнительных векторах From и To из N элементов и в целой переменной Top. Если задано начальное значение элемента Data [I], то  $\text{From}[I] \leq \text{Top}$  и  $\text{To}[\text{From}[I]] = I$ . Таким образом, From — простая сигнатура, а с помощью векторов To и Top гарантируется, что в ячейку From не попадет случайная сигнатура, обусловленная произвольным содержимым памяти. Пустые элементы в векторе Data теряют свои начальные значения, как показано на рисунке



Переменная Top в начальный момент равна 0; элемент I массива — первый, к которому осуществляется доступ с помощью следующей программы:



```

Top      := Top + 1
From[I]  := Top
To[Top]  := I
Data[I]  := 0

```

Такой метод (по Ахо, Хопкрофту и Ульману) искусно объединяет индексирование по ключу и хитроумную сигнатурную схему. Он может быть использован как для матриц, так и для векторов.

9. В отделе заказов бумажные бланки заказов размещаются в виде матрицы "карманов"  $10 \times 10$ , где две последние цифры номера телефона клиента используются в качестве хэш-индекса. Когда клиент делает по телефону заказ, этот заказ помещается в соответствующий "карман". Когда клиент прибывает за товарами, продавец последовательно ищет его заказ среди других в нужном "кармане" — это классическое "открытое хэширование с разрешением коллизий с помощью последовательного поиска". Две последние цифры телефонного номера достаточно близки к случайному числу и поэтому дают случайную хэш-функцию, в то время, как две первые цифры дали бы никуда не годную хэш-функцию — почему? В некоторых муниципалитетах близкая схема используется для записи документов в комплект регистрационных книг.
10. Компьютеры этих двух подразделений были связаны микроволновой линией, но для вывода чертежей на испытательной базе потребовался дорогой принтер. Поэтому обслуживающий персонал выдавал чертежи на основном предприятии, фотографировал их и посылал 35-мм пленку на испытательную станцию с почтовым голубем, а там фотографии увеличивались и печатались. 45-минутный полет голубя составлял только половину времени от доставки на автомобиле и стоил всего несколько долларов в день. В течение 16 месяцев, пока осуществлялся проект, голуби передали несколько сотен катушек с пленкой и только две из них потерялись (здесь водились ястребы; секретные данные не передавались).

## РЕШЕНИЯ К ГЛ. 2

- А. Полезно взглянуть на эту программу двоичного поиска с точки зрения 20 битов, которыми представлено каждое целое число. При первом проходе алгоритма мы читаем (максимально) 1 млн целых чисел на входе и записываем на одну ленту те из них, которые содержат в старшем разряде 0, а на другую — содержащие 1 в старшем разряде:

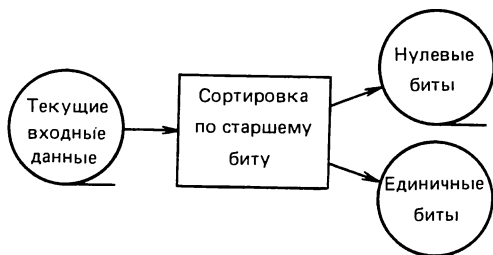


Рис. Р.2

На одной из этих двух лент может содержаться до 500 000 целых чисел. Мы используем ее в качестве очередных входных данных и повторяем процесс отбора, но теперь со вторым битом. Если на исходной входной ленте содержится  $N$  элементов, то при первом проходе мы читаем  $N$  чисел, при втором проходе максимально  $N/2$ , при третьем —  $N/4$  и т. д., так что общее время работы пропорционально  $N$ . Отсутствующее число может быть найдено с помощью упорядочения ленты и последующего ее сканирования, но на это потребовалось бы время, пропорциональное  $N \log N$ . Эта задача поставлена и решена Э. Рейнгоулдом из университета шт. Иллинойс.

Б. См. разд. 2.3.

В. См. разд. 2.4.

1. Чтобы найти все анаграммы для заданного слова, мы сначала вычисляем его сигнатуру. Если предварительная обработка не допускается, то мы должны читать весь словарь последовательно, вычислять сигнатуру каждого слова и сравнивать эти две сигнатуры. Вероятно, хорошей схемой для сигнатур является массив счетчиков. При использовании предварительной обработки мы выполняем двоичный поиск в заранее обработанном файле, содержащем пары (сигнатура, слово), упорядоченные в соответствии с сигнатурами.
2. Программа двоичного поиска обнаруживает элемент, который присутствует по крайней мере дважды, путем рекурсивного поиска на подынтервале, содержащем более половины этих чисел. Мое исходное решение не гарантировало, что количество чисел после каждой итерации уменьшается в 2 раза, так что в худшем случае время работы при  $\log_2 N$  проходах было пропорционально  $N \log N$ . Дж. Сакс из Университета Карнеги-Меллон уменьшил это время до линейной зависимости от  $N$ , отметив, что такой поиск может устранить запоминание чересчур большого количества дубликатов. Если в его программе поиска известно, что дубликат должен быть в текущей области из  $M$  чисел, она запоминает на текущей рабочей ленте только  $M + 1$  число; если на ленту стремится попасть большее количество чисел, программа просто отбрасывает их. Хотя при таком методе часто игнорируются входные переменные, эта стратегия достаточно осторожна, чтобы гарантировать, что она найдет по крайней мере один дубликат.
3. Следующая "жульническая" программа циклически сдвигает массив  $X[1 \dots N]$  влево на  $\text{RotDist}$  позиций:

```
for I := 1 to GCD(RotDist,N) do
    /* Переместить I-е элементы блоков */
    This := I
    Temp := X[This]
    Next := This + RotDist
    if Next > N then Next := Next-N
    while Next ≠ I do
        X[This] := X[Next]
        This := Next
        Next := Next + RotDist
```

```

if Next > N then Next := Next-N
X[This] := Temp

```

Наибольшим общим делителем чисел I и N является число циклов, которое надо переставить (в терминах современной алгебры это число классов смежности в группе перестановок, образованной вращением). Следующая программа взята из разд. 18.1 книги Гриза "Наука программирования". В ней предполагается, что подпрограмма Swap (A, B, L) меняет местами подмассивы X [A...A + L - 1] и X [B...B + L - 1]:

```

P := RotDist+1
I := RotDist; J := N-RotDist
while I ≠ J do
  /* Инвариант:
    X[1 ..P-I-1] - на окончательном месте
    X[P-I..P-1 ] = A - надо поменять местами с B
    X[P ..P+J-1] = B - надо поменять местами с A
    X[P+J..N    ] - на окончательном месте
  */
  if I > J then
    Swap(P-I,P,I);      I := I-J
  else
    Swap(P-I,P+J-I,I);  J := J-I
Swap(P-I,P,I)

```

Инварианты цикла описаны в гл. 4. Эта программа изоморфна аддитивному алгоритму Евклида (медленному, но правильному) для вычисления наибольшего общего делителя I и J:

```

function GCD(I,J)
  while I ≠ J
    if I > J then
      I := I-J
    else
      J := J-I
  return I

```

Гриз и Миллз рассмотрели все три алгоритма вращения в разделе Swapping Sections технического отчета по информатике Корнеллского университета (Cornell University Computer Science Technical Report 81-451).

4. Несмотря на то, что в программе с ухищрениями выполняется только половина таких же критических по времени операций, накладные расходы на ее реализацию в 2 раза больше, чем для более простого алгоритма реверсирования. Начиная с некоторого значения  $N$ , компьютеру приходится работать вне оперативной памяти, и он должен обращаться к страницам памяти на диске. Алгоритм реверсирования хорошо подходит для страничной организации памяти: из-за локального характера операции реверсирования он будет читать с диска каждую страницу дважды. С другой стороны, алгоритм с ухищрениями в этом смысле дает пессимистическую оценку: он читает страницу с диска, обрабатывает один элемент этой страницы и возвращается к ней после просмотра всех остальных страниц.
6. Сигнатура имени — это его кодирование с помощью кнопок номеронабирателя телефона, таким образом, сигнатура  $LESK^*M^*$  соответствует номеру 5375\*6\*. Чтобы найти ложные совпадения в справочнике, мы строим сигнатуры для всех имен в соответствии с их кодированием кнопками, упорядочиваем их по сигнатурам (и по именам для одинаковых сигнатур) и потом последовательно читаем упорядоченный файл, чтобы сообщить о совпадающих сигнатурах для разных имен. Чтобы получить имя, заданное нажатием кнопок, мы используем промежуточный файл, содержащий сигнатуры и другие данные. Хотя мы могли бы упорядочить этот файл и осуществлять двоичный поиск для имени, закодированного нажатием кнопок, в реальной системе мы, вероятно, использовали бы хэширование или пакет для работы с файлами на диске.
7. Для транспонирования матрицы Висоцки пронумеровал колонки и столбцы в каждой записи, вызвал системную программу сортировки на ленте, чтобы отсортировать элементы сначала по столбцам, затем по строкам, после чего с помощью еще одной программы узнал номера строк и столбцов.
8. *Aga! Озарение* для этой задачи заключается в том, что некое подмножество из  $K$  элементов имеет сумму, не превышающую  $T$ , тогда и только тогда, когда это выполняется для подмножества, состоящего из  $K$  минимальных элементов. Такое подмножество может быть найдено за время, пропорциональное  $N \log N$ , сортировкой исходного множества или за время, пропорциональное  $N$ , с помощью алгоритма отбора (см. решение задачи 9 гл. 10). Когда Ульман дал эту задачу в качестве упражнения, студенты разработали алгоритмы как с двумя вышеупомянутыми значениями времени работы, так и с временем работы  $O(N \log K)$ ,  $O(NK)$ ,  $O(N^2)$  и  $O(N^K)$ . Можете ли вы найти естественные алгоритмы с такими значениями времени работы?
10. Эдисон наполнил колбу водой и вылил ее в мензурку. (Если вы помните, в подсказке упоминался Архимед, который тоже использовал воду для вычисления объемов; в его время *aga! Озарение* отмечалось возгласом "Эврика!".) Вы можете вычислить центр тяжести двумерного объекта (попытайтесь сделать это для книги в жесткой обложке), используя большой, указательный и средний пальцы в качестве трех точек опоры. Если вы станете сближать пальцы, то, так как на наиболее удаленный от центра палец приходится наименьший вес, он имеет наименьшее трение и, скорее всего, будет двигаться относительно книги, поэтому точки опоры будут приближаться к центру тяжести. К. Дьюдни рассмотрел приспособление для аналоговых вычислений в колонке "Computer Recreations" в июньском за 1984 г. и в июльском за 1985 г. номерах журнала Scientific American.

1. Каждый элемент в таблице налогов содержит три значения: нижняя граница в данной группе, основной налог и процент дохода, превышающего нижнюю границу, взимаемый в виде налога. Добавление к концу таблицы специального элемента с "бесконечной" нижней границей упростит написание программы последовательного поиска и заодно сделает его более быстрым (см. задачу 5 гл. 8); можно использовать также двоичный поиск. Эти методы применимы к любым кусочно-линейным функциям.
3. Букву I (состоящую из символов X)

```

XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX
   XXX
   XXX
   XXX
   XXX
   XXX
   XXX
XXXXXXXXXX
XXXXXXXXXX
XXXXXXXXXX

```

можно закодировать так:

3 строки 9 X

6 строки 3 пробел 3 X 3 пробел

3 строки 9 X

или более компактно:

3 9 X

6 3 п 3 X 3 п

3 9 X

4. Чтобы найти число дней между двумя датами, вычислите для каждого дня его номер с начала соответствующего года, отнимите более ранний от более позднего (возможно заняв дни у года), а потом добавьте 365, умноженное на разность годов, плюс по одному дню на каждый високосный год. Чтобы найти день недели для заданной даты, вычислите число дней между этой датой и днем, про который известно, что он — воскресенье, а потом используйте арифметику по модулю 7, чтобы преобразовать это число в день недели. Для подготовки календаря на месяц заданного года мы должны знать, сколько дней в месяце (будьте внимательны, чтобы правильно вычислить их для февраля) и день недели, который приходится на 1-е число.
5. Так как сравнения выполняются в слове справа налево, то слова выгодно хранить в перевернутом (справа налево) виде. К возможным вариантам представления последовательности суффиксов относятся двумерный массив символов (что обычно расточитель-

но), одномерный массив символов, в котором суффиксы отделены друг от друга специальным символом, и такой же массив символов, к которому добавлен массив указателей, по одному на каждое слово.

8. Колонка "Жемчужины программирования" в июньском номере журнала САСМ за 1985 г. (с. 570 — 576) иллюстрируют ассоциативные массивы в АWK. Большинство структур данных, рассмотренных в разд. 11.3, пригодны для реализации ассоциативных массивов в различных контекстах, но, как правило, стоит выбрать хэширование.

#### РЕШЕНИЯ К ГЛ. 4

1. Чтобы показать, что переполнения не возникает, добавим к инварианту условия  $1 \leq L \leq N + 1$  и  $0 \leq U \leq N$ ; после этого можно ограничить сумму  $L + U$ . Эти же условия можно использовать, чтобы показать, что к элементам вне границ массива доступ никогда не происходит. Мы можем определить формально функцию MustBe ( $L, U$ ) как  $X[L - 1] < T$  и  $X[U + 1] > T$ , если мы определим фиктивные граничные элементы  $X[0]$  и  $X[N + 1]$ , как это сделано в разд. 8.3.
2. См. разд. 8.3.
5. В качестве введения к этой прославленной нерешенной математической задаче см. статью Б. Хайеса (B. Hayes. On the ups and downs of hailstone numbers") в разделе "Компьютерные развлечения" в январском номере журнала Scientific American за 1984 г. Технические подробности приведены в статье Дж. Лагариаса "Задача  $3x + 1$  и ее обобщения" (J. C. Lagarias. The  $3x + 1$  problem and its generalizations) в январском номере журнала American Mathematical Monthly за 1985 г.
6. Этот процесс завершается, так как на каждом шаге число бобов в банке уменьшается на 1. Так как всякий раз из кофейной банки удаляется 0 или 2 белых боба, сохраняется инвариант четности (четное или нечетное количество) количества белых бобов. Поэтому последним оставшимся бобом будет белый тогда и только тогда, когда в начальный момент количество белых бобов было нечетным.
7. Ввиду того, что отрезки прямых, образующие ступеньки лестницы, расположены в порядке возрастания  $Y$ , два таких отрезка, между которыми лежит заданная точка, можно найти с помощью двоичного поиска. Основное сравнение в этом поиске устанавливает, лежит точка ниже, на или выше заданного отрезка; как бы вы написали программу для такой процедуры?
8. См. разд. 8.3.
10. В колонке "Жемчужины программирования" в июльском номере журнала САСМ за 1985 г. описано, как я испытывал и отлаживал программу, а также алгоритмы двоичного поиска и выбора.

#### РЕШЕНИЯ К ГЛ. 5

1. Последовательная программа читает 10 000 блоков со скоростью 200 блоков/с, так что она всегда работает 50 с. Программа непосредственной обработки читает  $R$  записей за  $R/20$  с, таким образом, при  $R = 100$  она работает 5 с; при  $R = 10\,000$  она работает около 8 мин. Программа непосредственной обработки работает быстрее при  $R < 1000$ .
3. При использовании языка Си на компьютере VAX-11/750 для вычисления внутреннего произведения двух векторов с двойной точностью требуется на  $1/3$  больше времени, чем при вычислениях с одинарной точностью. При использовании Бейсика на персональном компьютере эти значения времени не различаются. В обоих случаях в системах большая

часть вычислений выполнялась с двойной точностью, даже если результаты усекались потом до одинарной точности.

4. В лабораторной системе изображения обрабатывались с большой точностью за счет следующих факторов: тщательной лабораторной проработки для повышения культуры проектирования; сканирующего устройства с разрешающей способностью 100 микрон, чтобы выдать компьютеру точное изображение; хороших, но "трудоемких" алгоритмов обработки изображений, эффективно реализованных на матричном процессоре.
6. "Прежде чем заставить "что-либо" работать быстро, заставьте "это" работать вообще — обычно хороший совет. Однако Б. Вулфу из фирмы Tartan Laboratories потребовалось только несколько минут, чтобы убедить меня, что эта старая банальность верна не настолько, как я некогда думал. Он воспользовался примером с системой подготовки документов, которую мы оба использовали. Хотя она работала быстрее своей предшественницы, иногда она казалось мучительно медлительной: ей требовалось несколько часов, чтобы сформировать книгу (хотя большие документы можно сформировать по частям). Решающим доводом Вулфа был примерно такой: "В этой программе, как и в любой другой большой системе, имеется десяток известных, но незначительных ошибок. В следующем месяце в ней будет 10 других известных ошибок. Если бы вы могли выбирать между устранением 10 имеющихся сейчас ошибок и ускорением работы программы в 10 раз, что бы вы выбрали?"

## РЕШЕНИЯ К ГЛ. 6

В эти решения включены предположительные значения некоторых констант, которые могут отличаться в 2 раза от их истинных значений к моменту, когда книга будет подписана в печать, но не в более отдаленные времена.

1. Если плотность записи на магнитную ленту составляет 6250 байтов на дюйм, умноженные на 2400 футов (длина ленты на бобине) и на 0.5 (потери при образовании блоков), то получим 90 Мбайт на бобине с лентой. Скорость передачи по линии связи, равная 7000 байтов/с, соответствует 25 Мбайт/ч. Поэтому велосипедист имеет около трех часов для передачи данных, что обеспечивает его превосходство в радиусе (примерно) 20 миль. Велосипедист имеет в 50 раз больший запас времени, т. е. почти неделю по сравнению с линией, скорость передачи которой составляет 1200 бод.
2. Большой диск имеет емкость 200 Мбайт, а 5.25-дюймовый гибкий диск — 200 Кбайт. Работая изо всех сил, я ввожу 50 слов (или 300 байтов)/мин. Поэтому для заполнения гибкого диска потребуется 700 мин (или примерно 12 ч), в то время как для диска большой емкости потребуется в 100 раз больше времени, т. е. несколько лет.
3. Терминал стоит 2000 дол. Программист (включая, к сожалению, многие затраты, кроме оклада) "стоит" около 100 000 дол в год, 2000 дол в неделю, 400 дол в день или 50 дол в час. Поэтому, если программист использовал терминал в течение 40 ч вне работы, затраты окупились и наниматель получает бесплатную работу. (На с. 630 раздела "Жемчужины программирования" в июльском номере журнала CACM за 1984 г. содержатся два интересных письма, обсуждающих эту проблему. Программист описывает различные способы использования терминала дома и заключает, что всего несколько часов работы дома могли бы сэкономить 40 ч работы "на работе". Руководитель разработки программного обеспечения перечисляет ряд проблем, связанных с домашними терминалами, таких как поддержание "норм по безопасности и охране здоровья в связи с терминалами в соответствии с актом Уолша-Хеалея".)

4. На выполнение микросекундной команды потребуется 1 с. На один оборот диска, выполняемый за 16 мс (при 3600 оборотах/мин), уйдет 5 ч, а на 30-миллисекундный поиск — 10 ч, 2 с, требующиеся для ввода моего имени, превратятся примерно в месяц.
5. За 1 с суперкомпьютер способен выполнить 100 млн операций с плавающей точкой над 64-разрядными числами; мидикомпьютер может выполнить 1 млн сложений 16-разрядных целых чисел; микрокомпьютер 0.5 млн 8-разрядных команд, а Бейсик на персональном компьютере может выполнить 100 операторов. Значения времени, указанные в задаче, дают примерно одинаковую производительность для первых трех компьютеров, но "бедный Бейсик" остается далеко позади.
6. Если не учитывать снижение быстродействия из-за очередей, то 30-миллисекундная дисковая операция обеспечивает время выполнения запроса, равное 3 с или 1200 запросов/ч.
7. Стоимость этого изменения составляет 100 дол времени компьютера плюс 400 дол времени программиста. При экономии 10 мин в день или 16 дол в день потребуется месяц, чтобы окупить затраты. Если скорость возросла вдвое, экономия, равная 80 дол в день, окупилась бы это ускорение за неделю.
9. Даже я могу вводить цифры со скоростью 1 цифра/с, что дает 3 записи/мин или 200 записей/ч. Если служащий вводил бы данные повторно, используя знакомые ему средства, это заняло бы меньше 2 ч и стоило бы менее 50 дол. Автоматизированное решение требовало бы программного обеспечения существенного объема (я бы тщательно искал подходящие пакеты, прежде чем писать программу самому), а также покупки модемов. Хотя решение на высоком техническом уровне, очевидно, предпочтительней для данных большого объема, простое решение лучше для рассматриваемой задачи.

#### РЕШЕНИЯ К ГЛ. 7

2. В алгоритме 1 используется примерно  $N^3/6$  вызовов процедуры `max`, в алгоритме 2 примерно  $N^2/2$  вызовов, а в алгоритме 4 примерно  $2N$  вызовов. В алгоритме 2b дополнительный объем памяти для кумулятивного массива растет линейно, а в алгоритме 3 дополнительный объем памяти для стека имеет логарифмическую зависимость, в других алгоритмах используется фиксированный объем дополнительной памяти. Алгоритм 4 пригоден для интерактивного взаимодействия: он находит ответ за один проход по входным данным, что особенно полезно при обработке файлов на диске.
3. Замените оператор присваивания `MaxSoFar := 0` на `MaxSoFar := -∞`. Если вас беспокоит использование `-∞`, `MaxSoFar := X[1]` тоже можно использовать; почему?
4. Задайте начальные значения в массиве `Sum` так, чтобы `Sum[I] = X[1] + ... + X[I]`. Сумма элементов подвектора `X[L...U]` равна 0, если `Sum[L-1] = Sum[U]`. Поэтому подвектор с суммой, наиболее близкой к 0, находится с помощью определения местоположения двух наиболее близких элементов в массиве `Sum`, что может быть сделано за время  $O(N \log N)$  посредством сортировки массива. Это время работы с точностью до коэффициентов оптимально, так как любой алгоритм для решения этой задачи может быть также использован, чтобы решить задачу "уникальности элементов", в которой определяется, содержит ли массив дубликаты (Добкин и Липтон показали, что для решения этой задачи требуется как раз столько времени в наихудшем случае на модели вычислений в виде дерева принятия решений).
5. Общая стоимость путешествия между пунктами I и J при линейной зависимости стоимости проезда от длины пути равна `Sum[J] - Sum[I-1]`, где `Sum` — кумулятивный массив, как и выше.



6. В этом решении используется другой кумулятивный массив.

```
for I := L to U do  
  X[I] := X[I]+V
```

Цикл моделируется следующими операторами присваивания:

```
Cum[U] := Cum[U]+V  
Cum[L-1] := Cum[L-1]-V
```

в которых символически добавляется элемент  $V$  к подвектору  $X[1 \dots U]$ , а затем вычитается из подвектора  $X[1 \dots L-1]$ . После того, как все такие суммы определены, мы вычисляем сумму массива  $X$  с помощью следующего фрагмента:

```
for I := N-1 downto 1 do  
  /* Цикл выполняется от N-1 до 1 "вниз" */  
  X[I] := X[I+1]+Cum[I]
```

Это сокращает время в наихудшем случае для  $N$  сумм с  $O(N^2)$  до  $O(N)$ . Такая задача возникла при сборе статистики в задаче  $N$  тел Аппела, описанной в разд. 5.1. Использование этого решения сократило время работы статистической подпрограммы с 4 ч до 20 мин; такое ускорение было бы незаметным, если бы программа работала год, но оно важно, когда она выполняется только за день.

7. Подмассив с максимальной суммой в массиве  $M \times N$  может быть найден за время  $O(M^2N)$  при использовании методики из алгоритма 2, если размерность равна  $M$ , и при использовании методики из алгоритма 4, если размерность равна  $N$ . Поэтому задача размерностью  $N \times N$  может быть решена за время, пропорциональное  $N^3$ . Это решение было независимо обнаружено более чем дюжиной читателей; имена большинства из них приведены в разделе "Жемчужины программирования" в ноябрьском номере журнала CACM за 1984 г. Как я знаю, нижняя граница для этой задачи пропорциональна  $N^2$ . Когда книга подписывалась в печать, эта задача оставалась открытой.

## РЕШЕНИЯ К ГЛ. 8

1. В компьютере с 8-битовыми байтами с помощью таблицы длиной 256 байтов можно было бы отобразить символы в 8 (возможно, перекрывающихся) классах символов. Бит  $I$  в байте  $J$  указывал бы, принадлежит ли символ  $J$  классу  $I$ . Проверка принадлежности включала бы доступ к элементу массива, выполнение операции AND (логическое И) с битовой маской, содержащей один бит, установленный в 1, и сравнение результата с 0.
2. Задано  $N$  — степень числа 2, мы должны установить начальные значения элементов  $C[0 \dots N-1]$  так, чтобы элемент  $C[I]$  содержал число битов, установленных в 1, в двоичном представлении индекса  $I$ . Мы используем тождество: при  $J < 2^K$  имеет место  $C[J + 2^K] = C[J] + 1$ ; что означает установку  $K$ -го бита в 1, т. е. добавление 1 к счетчику битов. По этой причине каждый элемент в правом столбце точно на 1 больше соответствующего элемента в левом столбце:

C[0] = 0	C[8] = 1
C[1] = 1	C[9] = 2
C[2] = 1	C[10] = 2
C[3] = 2	C[11] = 3
C[4] = 1	C[12] = 2
C[5] = 2	C[13] = 3
C[6] = 2	C[14] = 3
C[7] = 3	C[15] = 4

Поэтому программа начинает с таблицы, состоящей из одного элемента, и в цикле удваивает ее размер копированием и добавлением 1; она работает за время  $O(N)$ :

```

C[0] := 0
P := 1
while M < N do
/* Инвариант: значения элементов массива C[0..M-1]
вычислены */
  for J := 0 to P-1 do
    C[M+J] := C[J]+1
  P := P+P

```

После того, как эта задача появилась в журнале CACM, Дж. Де'Бозер из Скарбоуджа, Онтарио и Р. Боннет из Батлингтона, Вермонт нашли аналогичное решение с одним циклом. Ключом к программе Бозера является тот факт, что "целое число, полученное удалением из J старшего бита, установленного в 1, будет  $J - P$ ".

```

C[0] := 0
P := 1; PP := P+P
for J := 1 to N-1
  /* Инвариант: P - степень числа 2;
  PP = P+P; P <= J <= PP */
  if J = PP then
    P := PP; PP := P+P

```

/\* P -наибольшая степень числа 2 <= J \*/

C[J] := C[J-P]+1

Эта программа работает за время  $O(N)$  и пригодна для всех целых положительных  $N$ .

3. Если алгоритм двоичного поиска сообщил, что он обнаружил искомый элемент  $T$ , то установлен факт, что он есть в таблице. Однако, будучи примененными к неупорядоченным таблицам, такие алгоритмы могут иногда сообщать, что элемента  $T$  нет в таблице, когда в действительности он есть. В таких случаях эти алгоритмы обнаруживают пару соседних элементов, на основании которых, если бы таблица была упорядочена, можно было бы установить, что элемента  $T$  в ней нет.
4. Для этой таблицы Брукс объединил два представления. Функция обеспечивала верный ответ с точностью до нескольких единиц, а хранившаяся в массиве десятичная цифра соответствовала разности.
6. В разделе "Жемчужины программирования" в сентябрьском номере журнала CACM за 1984 г. (с. 870 — 871) дан набросок построения программы Р. Г. Дромея для вычисления максимального элемента в массиве, в которой используются сигнальные метки. Итоговая программа

I := 1

while I <= N do

    Max := X[I]; X[N+1] := Max; I := I+1

while X[I] < Max do I := I+1

Во всех упоминавшихся структурах данных можно применять дополнительный элемент — сигнальную метку, чтобы исключить проверку из внутреннего цикла поиска. Перед началом поиска в сигнальную метку помещается искомое значение, это гарантирует, что оно будет найдено. Когда поиск успешно завершится, одним сравнением можно установить, найдено ли "действительное" значение или значение, помещенное в сигнальную метку. Это исключает из внутреннего цикла проверку, исчерпаны ли уже входные данные.

Связанные списки имеют сигнальную метку в самом конце; необходимо также хранить указатель на такой узел (это особенно удобно в циклически связанных списках с дополнительным фиктивным узлом). В закрытых хэш-таблицах сигнальная метка устанавливается в конце массива. Нулевые указатели в стандартных деревьях двоичного поиска заменяются указателями на единственный узел — сигнальную метку в модифицированном дереве.

9. Замена вычисления функции на несколько 72-элементных таблиц сокращает время работы программы на компьютере IBM 7090 с  $1/2$  ч до 1 мин. Так как для расчета лопастей винта вертолета требуется около 300 прогонов этой программы, несколько сотен дополнительных слов памяти сокращают процессорное время с недели до нескольких часов.
10. Операционная система UNIX обеспечивает получение двух профилей. В первом содержатся данные о том, сколько раз вызывалась каждая процедура, а также временные характеристики. Когда работает программа, приведенная на следующей странице, выдается такая информация:

ВРЕМЯ, %	СУММАРНОЕ ВРЕМЯ	КОЛ-ВО ВЫЗОВОВ	МС/ВЫЗОВ	ИМЯ
35.2	0.26	223	1.11	_quicksort
31.8	0.49	7491	0.03	swap
13.6	0.59	1	100.04	_insert
8.0	0.65			mcount
6.8	0.70	1	50.02	_main
4.5	0.73			_rand

Эти данные квалифицируют процедуру swap как подходящего кандидата на то, чтобы выписать ее в явном виде. Программа получения более подробного профиля выводит весь текст программы и числа в левом столбце, которые являются счетчиками частоты выполнения каждого оператора. Будьте осторожны, чтобы не допустить ошибок при закрытии скобок — это особенность данной конкретной программы профилирования.

```

1      #include <stdio.h>
1      int n, x[1001];
1
1      main()
1      {   int i;
1          n = 1000;
1          for (i = 1; i <= n; i++) x[i] = rand();
1          quicksort(1, n); insert();
1          for (i = 1; i < n; i++)
999              if (x[i] > x[i+1])
0                  printf("Ошибка в quicksort");
7491      )
7491
7491      swap(px, py)
7491      int *px, *py;
7491      {   int t;
1          t = *px; *px = *py; *py = t;

```

```

1      }
1
1      insert()
1      {   int i, j;
1          for (i = 2; i <= n; i++) {
999          j = i;
3115          while (j > 1 && x[j] < x[j-1]) {
2116              swap(&x[j], &x[j-1]);
2116              j--;
999          }
233      }
233  }
233
233  quicksort(l, u)
233  int l, u;
233  {   int i, m;
233      if (u-l > 15) {
116          swap(&x[l], &x[l+(u-l+1)*rand()/32768]);
116          m = l;
116          for (i = l+1; i <= u; i++)
9288              if (x[i] < x[l])
5143                  swap(&x[++m], &x[i]);
9288          swap(&x[l], &x[m]);
116          quicksort(l, m-1);
116          quicksort(m, u);
116      }

```

Эта программа рассматривается в разд. 10.2. Обратите внимание, что в цикле сортировки методом вставок (insert) каждый элемент сдвигается вниз в среднем на 2.1 позиции. Какие еще счетчики сообщают интересную информацию?

11. С помощью метода Хорнера этот полином вычисляется так:

$Y := A[N]$

for  $I := N-1$  downto  $0$  do

$Y := X * Y + A[I]$

Здесь используется  $N$  умножений, и такая программа работает в 2 раза быстрее, чем предыдущая.

## РЕШЕНИЯ К ГЛ. 9

1. Каждый оператор на языке высокого уровня, осуществляющий доступ к одному из упакованных полей, компилируется в большое число машинных команд; для доступа к неупакованному полю требуется меньше команд. Распаковав записи, Фельдман несколько увеличил объем памяти под данные, но сильно сократил память для команд и время работы программы.
2. Некоторые читатели предложили запоминать тройки  $(x, y, \text{PointIdentifier})$ , упорядоченные по  $y$  для каждого  $x$ ; после этого для поиска заданной пары  $(x, y)$  может быть использован двоичный поиск. Структуру данных, описанную в основном тексте книги, легче формировать, если входные данные упорядочены по значениям  $x$  (а при постоянном  $x$  — по  $y$ , как и ранее). Для структуры, описанной в тексте, поиск можно было бы выполнить быстрее, осуществляя двоичный поиск в массиве  $\text{Row}$  от значения  $\text{FirstInCol}[I]$  до значения  $\text{FirstInCol}[I+1] - 1$ . Обратите внимание, что значения  $y$  появляются в порядке возрастания и что программа двоичного поиска должна корректно работать в случае пустого подмассива. Объем памяти, необходимый для реализации данной структуры, может быть уменьшен на 2000 16-разрядных слов при использовании индексирования по ключу: элемент  $I$  массива содержит два 1-байтовых поля, которые дают позиции  $x$  и  $y$  для  $I$ .
3. В справочнике таблицы расстояний между городами запоминаются в виде треугольных массивов, что сокращает их объем в 2 раза. В математических таблицах иногда хранятся только последние значащие цифры функций, а старшие значащие цифры приводятся только 1 раз, скажем, для каждой строки значений. При составлении телевизионных программ объем сокращается за счет указания только моментов начала программы (в отличие от перечисления всех программ, идущих в эфир на любом заданном 30-мин интервале). В рекламных объявлениях, помещаемых в газетах, место экономится, например, за счет сокращенного обозначения домов и автомобилей.
4. Эта последовательность может быть укорочена с помощью кодирования идущих подряд одинаковых символов. Мы будем использовать три числа 0,  $n$ ,  $x$ , чтобы обозначить  $n$  повторений числа  $x$ ; одиночный нуль кодируется как 0, 1, 0. Тогда последовательность в задаче сжимается таким образом:

0, 38, 128, 152, 166, 172, 153, 164, 0, 19, 128,

и расходуется вместо 62 байтов 11.

5. Для исходного файла требуется 300 Кбайтов памяти на диске. Уплотнение двух цифр в один байт уменьшает этот объем до 150 Кбайтов, но увеличивает время на чтение файла. Замена длительных операций  $\text{div}$  и  $\text{mod}$  просмотром таблицы обходится в 200 байтов в ОЗУ, но сокращает время чтения почти до его первоначального значения. Таким образом, за 200 байтов ОЗУ покупается 150 Кбайтов на диске. Другие варианты кодирования, например, представление  $A$  и  $B$  в виде  $16 \times A + B$ , могли бы дать сравнимую скорость без использования таблицы, а также обеспечить другие преимущества (см. сноску в разд. 9.2).

1. Выполнять сортировку, чтобы найти минимум или максимум среди  $N$  действительных чисел, все равно, что стрелять из пушки по воробьям. В решении задачи 9 показано, как медиану можно найти быстрее и без сортировки, но в некоторых системах может оказаться проще осуществить сортировку. Сортировка обычно вполне пригодна для нахождения моды. Хотя для вычисления среднего с помощью очевидной программы требуется время, пропорциональное  $N$ , при подходе, сначала предполагающем сортировку, можно выполнить эту работу с большей точностью (см. задачу 3.Б гл. 12).
2. Данная программа приводит к ошибке, если операционная система обеспечивает проверку границ массивов и проверяется выполнение обоих условий в логическом операторе, даже когда первое имеет значение "ложь". Аналогичная ситуация возникает в программе сортировки методом вставок из разд. 10.1, если  $J = 1$ . Ошибки могут быть устранены несколькими способами, такими как использование условного оператора `and`, задание элемента  $X[0]$  (и, возможно, использование его в качестве ячейки для сигнальной метки), засылка минимального значения в элемент  $X[1]$  перед вызовом основной сортировки, или использованием булевой переменной.
3. Если все элементы равны, программа быстрой сортировки, приведенная в этой главе, удаляет только один элемент при каждом из  $N$  рекурсивных обращений. Программа Седжуика имеет следующий инвариант:

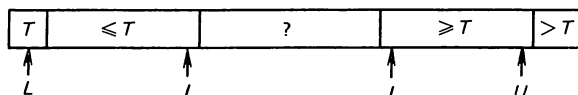


Рис. Р.3

При "лишних" взаимных перестановках одинаковых элементов она сокращает ровно в 2 раза подмассив дубликатов ключей.

4. Седжуик отметил, что метод Ломута может быть модифицирован для работы справа налево при использовании следующего инварианта:

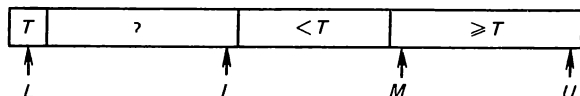


Рис. Р.4

Тогда программа разбиения выглядит так:

```

M := U+1
for I := U downto L do
  if X[I] >= T then
    M := M-1
    Swap(X[M], X[I])
  
```

При завершении программы мы знаем, что  $X[M] = T$ , так что можно организовать рекурсию с параметрами  $(L, M - 1)$  и  $(M + 1, U)$ ; дополнительной процедуры `Swap` не требуется. Седжуик использовал также элемент  $X[L]$  в качестве сигнальной метки, чтобы исключить из внутреннего цикла одну проверку:

```

M := U+1
I := U+1
repeat
    repeat I := I-1 until X[I] >= T
    M := M-1
    Swap(X[M],X[I])
until I = L

```

5. См. статью Седжуика, на которую дана ссылка в разд. 10.5.
7. Для работы программы Макилроя требуется время, пропорциональное объему данных, которые нужно отсортировать, что оптимально для худшего случая. В ней предполагается, что каждой записи в массиве  $X[1 \dots N]$  соответствует целое число  $Lengh$  и указатель на массив  $Bit[1 \dots Lengh]$ .

```

procedure Sort(L, U, Depth)
    if L < U then
        for I := L to U do
            if X[I].Lengh < Depth then
                Swap(X[I],X[L])
                L := L+1
        M := L
        for I := L to U do
            if X[I].Bit[Depth] = 0 then
                Swap(X[I],X[M])
                M := M+1
        Sort(L, M-1, Depth+1)
        Sort(M, U, Depth+1)

```

Эта процедура вызывается первоначально с параметрами  $Sort(1, N, 1)$ . В программе присваиваются значения параметрам и переменным, определяющим циклы `for`. Хотя это изящно выглядит во многих языках, но зато в некоторых других запрещено (например, в Паскале). Линейная зависимость времени работы строго определяется тем фактом, что операция `Swap` перемещает указатели на битовые строки, а не сами эти строки.

8. Ответ на этот вопрос целиком зависит от конкретной операционной системы. Чтобы получить основательный, но в чем-то устаревший ответ, обратитесь к статье В. А. Мартина "Сортировка" (W. A. Martin. Sorting) в декабрьском номере журнала *Computing Surveys* 3, 4, за 1971 г.



9. Этот алгоритм существует благодаря С. А. Р. Хор; он обсуждается в разделе "Жемчужины программирования" в ноябрьском номере журнала CACM за 1985 г.

procedure Select(L, U, K)

pre L <= K <= U /\* На входе \*/

post X[L..K-1] <= X[K] <= X[K+1..U] /\* На выходе \*/

if L < U then

Swap(X[L], X[RandInt(L,U)])

T := X[L]

M := L

for I := L+1 to U do

/\* Инвариант: X[L+1...M] < T

и X[M+1...I-1] >= T \*/

if X[I] < T then

M := M+1

Swap(X[M], X[I])

Swap(X[L], X[M])

/\* X[L...M-1] <= X[M] <= X[M+1...U] \*/

if M < K then Select(M+1, U, K)

else if M > K then Select(L, M-1, K)

Эту программу можно улучшить многими способами. Программа разбиения, приведенная в задаче 3 гл. 10, исключает наихудший дорогостоящий случай. Так как рекурсия — последнее действие в процедуре, она может быть преобразована в цикл while:

L := 1; U := N

while L < U do

/\* Инвариант: X[1...L-1] <= X[L...U] <= X[U+1...N] \*/

Procedure Select(K)

post X[1...K-1] <= X[K] <= X[K+1...N] /\* На выходе \*/

... Здесь тело цикла ...

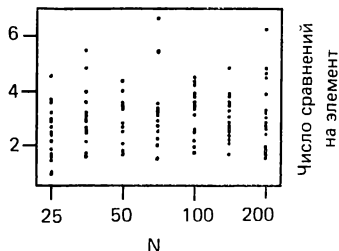
/\* Инвариант: и X[L...M-1] <= X[M] <= X[M+1...U] \*/

```
if M <= K then L := M+1
```

```
if M >= K then U := M-1
```

В задачах 5.2.22 — 32 в книге "Сортировка и поиск" Кнут показывает, что для поиска медианы  $N$  элементов эта программа выполняет в среднем  $3.4N$  сравнений; вероятностное доказательство "близко по духу" доказательству для наихудшего случая в решении задачи 2.А. Для подтверждения интуитивных представлений об эффективности я осуществил сбор этих данных с помощью программы на языке AWK, состоящей из 16 строк, которая генерировала случайные данные, запускала алгоритм и собирала статистику:

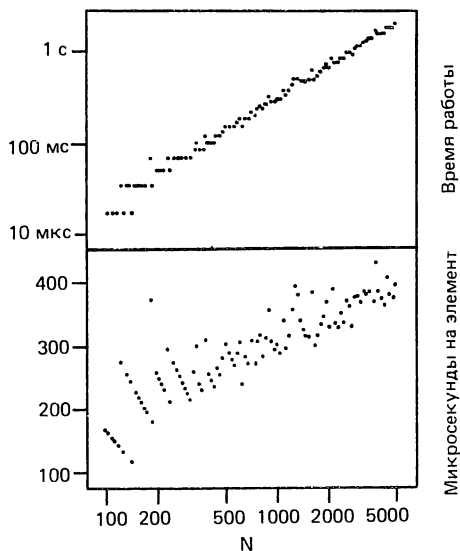
Рис. Р.5



Только изредка программе требовалось более  $4N$  сравнений, чтобы найти медиану для массива из  $N$  элементов. (Сравните эти данные с возрастающей зависимостью на втором графике в следующем решении).

10. На верхнем графике изображено время работы программы быстрой сортировки 2 как функция от размера входного массива —  $N$ ; 100 значений  $N$  равномерно разместились вдоль логарифмической оси. Для малых времен график дискретен, так как в моей системе время измерялось в единицах, равных  $1/60$  с. Этот график показывает, что время работы строго коррелировано с размерностью на входе, но не позволяет сделать более глубокий вывод.

Рис. Р.6



Ось на нижнем графике соответствует времени работы в микросекундах, приходящемуся на один элемент массива (т. е. общее время, деленное на  $N$ ). Этот график демонстрирует широкую вариацию времени работы из-за рандомизации процедуры Swar. Линейная зависимость данных (по крайней мере, при  $N > 500$ ) показывает, что время работы, приходящееся на один элемент, растет логарифмически, это означает, что общее время работы программы быстрой сортировки  $O(N \log N)$ . В большинстве учебников по алгоритмам дается математическое доказательство этого факта.

12. Отдел распространения журнала Scientific American получает большой объем почты. (Задача на оценочные расчеты: сколько писем журнал получает ежедневно?) Большая часть корреспонденции относится к нескольким основным категориям: оплата счетов, возобновление подписки, ответы на письма и вложенные в журнал почтовые открытки и т. д. Прежде чем почтой займутся сотрудники, осуществляющие ввод данных, она должна быть отсортирована по таким группам. Сортировка почты вручную дорога, поэтому журнал заставил делать эту работу для себя почтовое ведомство Соединенных Штатов: используются различные номера почтовых индексов для каждой из основных категорий (каждый индекс стоит около 100 дол в год).

## РЕШЕНИЯ К ГЛ. 11

1. Чтобы выбрать  $M$  целых чисел в диапазоне  $1, \dots, N$ , возьмите случайное число  $I$  из этого диапазона и выдайте числа  $I, I + 1, \dots, I + M - 1, \dots$ . При этом методе каждое целое число выбирается с вероятностью  $M/N$ , но эти числа обязательно смещены к некоторому подмножеству.
2. Если ранее было выбрано менее  $N/2$  целых чисел, вероятность того, что очередное случайно взятое число не было выбрано ранее, больше  $1/2$ . То, что среднее количество попыток для получения не выбранного ранее числа меньше двух, логически следует из того, что надо бросать монету в среднем дважды, чтобы выпал "орел".
3. Давайте рассматривать множество  $S$  в задаче 2 как набор  $N$  пустых в начальный момент урн. При каждом обращении к функции RandInt выбирается урна, в которую мы кладем шар; если она была ранее занята, проверка, выполняемая процедурой Member, дает значение "истина". Количество шаров, требуемое, чтобы гарантировать, что в каждой урне содержится по крайней мере один шар, известно статистикам как "задача сбора купонов" (сколько бейсбольных билетов должен я собрать, чтобы гарантировать, что у меня есть все  $N$ ?); ответ — округленно  $N \ln N$ . В этом алгоритме выполняется  $M$  проверок, если все шары попадают в разные урны; определения вероятности попадания двух шаров в одну урну — это "парадокс дней рождения" (в любой группе из 23 и более людей с большей вероятностью у двоих совпадают дни рождения). Вообще, два шара с большой вероятностью попадут в одну урну из  $N$ , если имеется  $O(\sqrt{N})$  шаров.
6. Чтобы напечатать эти величины в возрастающем порядке, можно поместить оператор print после рекурсивного вызова или печатать не  $I$ , а  $N + 1 - I$ .
7. Чтобы напечатать различные целые числа в случайном порядке, печатайте каждое при его первом появлении; см. также решение задачи 4 гл. 1. Чтобы напечатать числа с дубликатами в упорядоченном виде, исключите проверку того, присутствует ли уже число в наборе. Чтобы напечатать числа с дубликатами в случайном порядке, воспользуйтесь следующей тривиальной программой:

```
for I := 1 to M do
    print RandInt(1,N)
```

8. Я дал эту задачу точно в таком виде, как она сформулирована, в качестве домашнего задания по курсу "Разработка прикладных алгоритмов". Студенты, которые описали методы, требующие для получения ответа нескольких минут процессорного времени, получили нулевые баллы. Ответ "Я должен поговорить с преподавателем статистики" заслуживал частичную похвалу, а отличный ответ выглядел примерно так:

Числа от 4 до 10 не влияют на ход игры, так что их можно проигнорировать. Карта выигрывает, если перед числом 3 выбраны числа 1 и 2 (в любом порядке). Это случается, если число 3 выбирается последним, что происходит один раз из трех. Поэтому вероятность того, что случайная последовательность выигрывает, равна в точности  $1/3$ .

Пусть вас не вводит в заблуждение постановка задачи; вы не должны использовать время процессора именно потому, что процессор есть под рукой!

## РЕШЕНИЯ К ГЛ. 12

1. Модифицированная процедура SiftDown выглядит так:

```
procedure SiftDown(L,U)
```

```
    pre  MaxHeap(L+1,U)    /* На входе */
```

```
    post MaxHeap(L,U)      /* На выходе */
```

```
    I := L
```

```
    loop
```

```
    /* Инвариант: функция MaxHeap(L,U) - истинна, за исключением,
       возможно, участка от I до его (0,1,2)
       "сыновей" */
```

```
    C := 2*I
```

```
    if C > U then break
```

```
    if C+1 <= U and X[C+1] > X[C] then C := C+1
```

```
    /* C - наибольший "сын" I */
```

```
    if X[I] >= X[C] then break
```

```
    Swap(X[C], X[I])
```

```
    I := C
```

Оператор and и второй оператор if должны быть условными. Время работы этой процедуры  $O(\log U - \log L)$ . Эта программа строит пирамиду за время  $O(N)$ :

```
for I := N-1 downto 1 do
```

```
  /* Инвариант: функция MaxHeap(I+1,N) - истинна */
```

```
  SiftDown(I,N)
```

```
  /* MaxHeap(I,N) */
```

Так как функция  $\text{MaxHeap}(L, N)$  имеет значение "истина" для всех целых чисел  $L > N/2$ , граница  $N - 1$  в цикле `for` может быть заменена на `int(N/2)`.

2. При использовании программы при решении задачи 1 пирамидальная сортировка выглядит так:

```
for I := int(N/2) downto 1 do
```

```
  SiftDown(I,N)
```

```
for I := N downto 2 do
```

```
  Swap(X[I], X[1])
```

```
  SiftDown(1, I-1)
```

Ее время работы по-прежнему выражается функцией  $O(N \log N)$ , но с меньшим коэффициентом, чем в исходной пирамидальной сортировке. Быстродействие процедуры `SiftDown` можно повысить, если вынести из ее цикла операторы пересылки значения во временную переменную  $T$  и обратной пересылки из  $T$ . Быстродействие процедуры `SiftUp` может быть повышено за счет исключения команд из цикла и размещения сигнальной метки в ячейку  $X[0]$ , чтобы исключить проверку `if I = 1`.

3. Использование пирамид заменяет во всех этих задачах количество шагов с  $O(N)$  на  $O(\log N)$ .

А. При построении кода Хоффмана на каждой итерации выбираются два наименьших узла в множестве и объединяются в один новый узел; это реализуется с помощью двух процедур `ExtractMins`, за которыми следует процедура `Insert`. Если частоты на входе даны в упорядоченном виде, то время вычисления кода Хоффмана будет линейно зависеть от  $N$ ; детали оставлены в качестве дополнительного упражнения.

Б. В простом алгоритме для суммирования чисел с плавающей точкой при сложении очень малых чисел с очень большими может потеряться точность. Наилучший алгоритм всегда складывает два наименьших в множестве числа и изоморфен по отношению к алгоритму для кода Хоффмана, который упоминался выше.

В. Пирамида из 1000 элементов (с минимальным в вершине) содержит 1000 наибольших из уже просмотренных чисел.

Г. Пирамиду можно использовать для слияния упорядоченных файлов путем представления следующего элемента в каждом файле; на каждой итерации из пирамиды выбирается наименьший элемент, а в пирамиду вставляется следующий за ним. Следующий элемент, который должен быть выбран из  $N$  файлов, может быть найден за время  $O(\log N)$ . (В статье Линдермана, на которую дана ссылка в разд. 10.5, описано, как этот алгоритм увеличивает время работы реальной программы системной сортировки; Линдерман сократил это время, применив двоичный поиск.)

4. Над последовательностью "карманов" надстраивается пирамидальная структура; в каждом узле пирамиды содержится информация об объеме свободного места в наименее заполненном относительно своих сыновей "кармане". При принятии решения о том, куда поместить новый вес, поиск идет влево, если вес можно поместить в "карман" (т. е. в наименее заполненном левом "кармане" имеется достаточно места для данного веса), а если нет, то вправо, к "карманам", куда вес обязан поместиться. На это требуется время, пропорциональное глубине пирамиды, т. е.  $O(\log N)$ . После того, как вес помещен, пирамида перестраивается, чтобы отобразить новую заполненность "карманов". Д. Джонсон, Т. Лейттон, К. Макгеох и я использовали этот алгоритм для проведения экспериментов по упаковке "карманов". Мы были способны решить задачу размерностью  $N = 100\,000$  за 1 мин, в то время как при использовании последовательного поиска столько времени требуется для  $N = 1000$ .
5. При обычной реализации последовательного файла на диске в блоке  $I$  содержится указатель на блок  $I + 1$ . Маккрейт отметил, что если в узле  $I$  содержится также указатель на узел  $2I$ , то произвольный узел  $N$  может быть найден максимально за  $1 + \log_2 N$  обращений. Следующая рекурсивная программа выводит такой путь доступа на печать:

```
function Path(N)

    pre   целое число N >= 0      /* На входе */
    post  печатается путь до N    /* На выходе */

    if N = 0 then
        print "Начать с 0"
    else if Even(N) then
        path(N/2)
        print "Удвоить ", N
    else
        path(N-1)
        print "Увеличить на 1 ", N
```

Обратите внимание, что эта программа изоморфна по отношению к программе вычисления  $X^N$  за  $O(\log N)$  шагов, которая приведена в задаче 9 гл. 4.

6. Модифицированный двоичный поиск начинается при  $I = 1$  и на каждой итерации устанавливает  $I$  равным либо  $2I$ , либо  $2I + 1$ . В ячейке  $X[1]$  содержится медиана, в  $X[2]$  — первая квартиль,  $X[3]$  — третья квартиль и т. д. С. Р. Махани из фирмы Bell Labs и Дж. И. Мурро из университета Ватерлоо получили программу для расстановки  $N$  элементов упорядоченного массива в порядке "пирамидального поиска" за время  $O(N)$  и с использованием дополнительной памяти объёмом  $O(1)$ . В качестве предшественника их метода рассмотрим копирование упорядоченного массива  $A$  размером  $2^K - 1$  в массив "пирамидального поиска"  $B$ ; нечетные элементы из массива  $A$  переходят по порядку во вторую половину массива  $B$ ; элементы конгруэнтные 2 по модулю 4, переходят во вторую четверть  $B$  и т. д.

10. Я реализовал очередь с приоритетами на базе пирамиды с использованием языка Си++ (см. книгу Строустрапа "Язык программирования Си++", Stroustrup. C++ Programming Language, выпущенную издательством Addison-Wesley в 1986 г.). Для членов, принадлежащих классу очередей с приоритетами (priqueue), определяются следующие атрибуты:

```
class priqueue {  
    int n, maxsize;  
    float *x;  
    void swap(int,int);  
    int heap(int,int);  
public:  
    priqueue(int);  
    void insert(float);  
    float extractmin();  
};
```

Определения под заголовком public доступны всем пользователям класса, тогда как определения, обозначенные private (переменные и процедуры), доступны только конкретному разработчику. В данном случае пользователи видят только процедуры insert и extractmin и (неявно) следующую процедуру, которая формирует элемент этого класса (имя процедуры совпадает с именем класса):

```
priqueue::priqueue(int m)  
{  
    if (m < 1) error ("Ошибка в размере");  
    maxsize = m;  
    x = new float[maxsize+1];  
    n = 0;  
}
```

Эта процедура проверяет максимальный размер приоритетной очереди, запоминает его в ячейке maxsize, выделяет новый массив и устанавливает начальное значение для n. При объявлении массива указывается размер maxsize + 1, так как массивы в языке Си++ считаются с нулевого элемента, а в пирамидах используется индексация с 1. В моей реализации использовалось макроопределение ASSERT для проверки инварианта структуры данных:

```
#define ASSERT(e) if (!(e)) error("Утверждение не выполнено")
```

Макроопределения в языке Си++ такие же, как и в языке Си. Класс приоритетных очередей имеет функцию типа private, которая имеет значение "истина" (возвращает на выход

де 1), если массив обладает признаком пирамидальности, и значение "ложь" в противном случае.

```
int priqueue::heap(int l, int u)
{   for (int i = 2*l; i <= u; i++)
        if (x[i/2] > x[i]) return 0;
    return 1;
}
```

Я использовал ее для проверки программы, но эта процедура настолько трудоемкая, что в промышленных версиях программы макроопределение ASSERT должно быть определено как нуль, чтобы исключить вычисление функции heap.

Эта процедура помещает t в очередь с приоритетами. Она проверяет инвариант структуры данных на входе и на выходе.

```
void priqueue::insert(float t)
{   ASSERT(heap(1,n));
    if (++n > maxsize) error("Слишком много элементов");
    x[n] = t;
    int i = n;
    for (;;) { /* Инвариант: функция heap(1,n) - истинна, за
                  исключением, возможно, участка от i до
                  его "отца" */
        if (i == 1) break;
        int p = i/2;
        if (x[p] < x[i]) break;
        swap(p,i);
        i = p;
    }
    ASSERT(heap(1,n));
}
```

Цикл начинается оператором for, а заканчивается оператором break, swap — процедура типа private, которая переставляет указанные элементы массива x. Процедуры swap и extractmin определяются аналогично. Получившийся в результате класс я использовал в



этой пробной программе, которая заполняет массив А целыми числами, а затем сортирует их.

```
main()  
( /* Объявление и задание начальных значений массива */  
  int n = 10; float a[n];  
  for (int i = 0; i < n; i++) a[i] = n-i;  
  
  /* Сортировка с использованием приоритетной очереди */  
  priqueue q(n);  
  for (i = 0; i < n; i++) q.insert(a[i]);  
  for (i = 0; i < n; i++) a[i] = q.extractmin();  
)
```

В строке, следующей за вторым комментарием, объявляется, что q — приоритетная очередь с максимальным размером n, что аналогично объявлению массива float a[n] в третьей строке. В двух следующих строках эти операции применяются к объекту q. Данный класс может быть использован также в объявлениях, подобных следующим:

```
priqueue events(30);  
priqueue times(maxtimes);
```

С помощью этого языка можно выделить память для различных приоритетных очередей и связать переменные n и x с соответствующими объектами. Пользователь может теперь считать приоритетные очереди новым примитивом, таким, например, как целые числа или массивы.

### РЕШЕНИЯ К ГЛ. 13

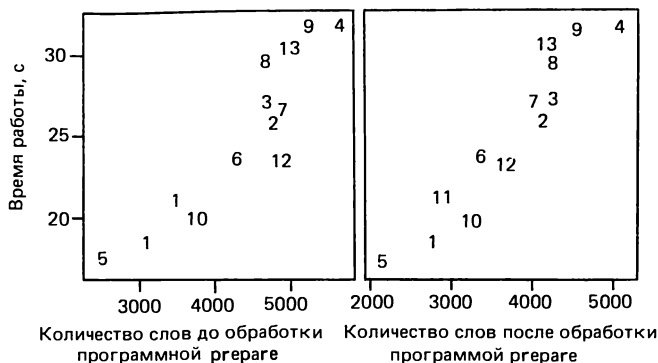
В журнале Bell System Technical Journal 57, 6, ч. 2 (июль — август 1978 г., с. 2137 — 2154) опубликована статья Макмагона, Черри и Морриса "Статистическая обработка текста" (McMahon, Cherry, Morris. Statistical text processing). В ней описываются средства для сбора статистики и результат использования этих средств.

На графике слева показано время работы программы spell на компьютере VAX-11/750 для тринадцати глав этой книги как функция от общего числа слов; числа на графике — номера глав. Я собрал эти данные, используя системную программу подсчета количества слов wc в операционной системе UNIX.

Когда этот график оказался не таким прямолинейным, как я ожидал, я сразу понял в чем дело: гл. 12, например, обрабатывается быстро, так как в ней много рисунков, которые не обрабатываются программой spell. Поэтому на правом графике подсчитаны слова, которые остались после программы rereage (в данном случае в этой роли выступала прог-

рамма операционной системы UNIX deroff), удаляющей команды форматирования; он дает более точное представление о времени работы.

Рис. Р.7



- См. статьи "Опыт использования эффективного по памяти способа хранения словаря" Р. Никса (R. Nix. Experience with a space efficient way to store a dictionary) в майском номере журнала CACM за 1981 г. и "Сокращение размера словаря при использовании методов хэширования" Д. Дж. Доддса (D. J. Dodds. Reducing dictionary size by using a hashing technique) в июньском номере журнала CACM за 1982 г.
- В одном из способов слова упорядочиваются, а затем сканируются для обнаружения близких несовпадений (например, слова `programer` и `programmer`); может быть, имеет смысл выполнить это в прямом и обратном направлении (чтобы выловить слово `pregrammer`). В статье, на которую дана ссылка в решении задачи 1, описывается программа, которая читает документ, подсчитывает частоты упоминания всех пар и троек букв, а затем выводит слова с подозрительными сочетаниями (такими как двойное `x` в слове `REXX`).
- Программа, предлагающая пользователю вариант правильного написания, могла бы выполнять некоторые операции над входными словами (такие как перестановка двух смежных букв, добавление или удаление одной буквы) и сообщать обо всех перемещениях в словаре. Об интерактивном подходе, пригодном для небольших словарей см. в статье Дурхама, Лэмба и Сакса "Исправление написания слов при взаимодействии с пользователем" (Durham, Lamb, Saxe. Spelling correction in user interfaces в октябрьском номере журнала CACM за 1983 г.). В качестве другого подхода можно использовать метод Soundex, упомянутый в разд. 2.5.
- В статье Макилроя, на которую дана ссылка в разд. 13.6, описаны "британские" дополнения, с помощью которых обрабатываются слова, подобные `centre`, `favour`, `realise` и `speciality`.
- Статья Л. Черри "Средства для письма" (Cherry. Writing tools) появилась в журнале IEEE Transactions on Communications COM-30, 1 (январь 1982 г., с. 100 — 104). В ней описано множество программ для нахождения таких ошибок, как многословные фразы, плохой стиль, ошибки пунктуации и инфинитив с отделенной частицей.

90 к.



• РАДИО И СВЯЗЬ •

