БИБЛИОТЕКА ПРОГРАММИСТА

Джон Бентли

Жемчужины программирования

2-е издание

- постановка задачи
- оптимизация кода
- минимизация памяти
- верификация программ

и многое другое...

Jon Bentley

Programming Pearls

Second Edition

Addison-Wesley
An imprint of Addison Wesley Longman, Inc.
Reading, Masachusetts • Harlow, England • Menlo Park, California
Berkley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Джон Бентли

БИБЛИОТЕКА ПРОГРАММИСТА

Жемчужины программирования

2-е издание

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростоа-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2002

Джон Бентли

Жемчужины программирования 2-е издание

Перевел с английского Д. Солнышков

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Научный редактор	*А. Пасечник*
Художник	*Н. Биржаков*
Иллюстрации	*Л. Панич*
Корректор	*Н. Тюрина*
Верстка	*Л. Панич*

ББК 32.973.2-018
УДК 681.3.06

Бентли Дж.

Б46 Жемчужины программирования. 2-е издание. — СПб.: Питер, 2002. — 272 с.: ил.

ISBN 5-318-00715-5

Эта книга написана для программистов. Хороший программист должен знать все, что написано до него, только тогда он будет писать хорошие программы. Главы этой книги посвящены наиболее привлекательному аспекту профессии программиста: жемчужинам программирования, рождающимся за пределами работы, в области фантазии и творчества. В них рассматриваются: постановка задач, теория алгоритмов, структуры данных, вопросы повышения эффективности кода, а также верификация и тестирование программ.

ISBN 5-318-00715-5
ISBN 0-201-65788-0 (англ.)

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.
Лицензия ИД № 05784 от 07.09.01.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 953005 — литература учебная.
Подписано в печать 26.09.02. Формат 70×100/16. Усл. п. л. 21,93. Доп. тираж 4000 экз. Заказ № 1411.
Отпечатано с фотоформ в ФГУП «Печатный двор» им. А. М. Горького
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Содержание

Предисловие

Компьютерное программирование многолико. Фред Брукс в книге «Мифический человеко-месяц» (Fred Brooks, The Mithical Man Month) дает общий план. В его эссе недооценивается жизненно важная роль управления в больших компьютерных проектах. Более высокий уровень детализации дает Стив Макконнелл в книге «Программа написана» (Steve McConnell, Code Complete). Он учит хорошему стилю программирования. Хороший программист должен знать все, что написано в этих книгах, и тогда он будет писать хорошие программы. К сожалению, искусное применение отлично разработанных принципов проектирования оказывается не всегда интересным — пока программа не будет написана вовремя и не будет вести себя предсказуемо.

О книге

Главы этой книги посвящены более привлекательному аспекту профессии программиста: жемчужинам программирования, рождающимся за пределами работы, в области фантазии и творчества. Как настоящие жемчужины растут из песчинок, которые попадают внутрь раковины и раздражают моллюска, так жемчужины творчества программистов произрастают из реальных задач, раздражающих реальных программистов. Получающиеся в результате программы оказываются интересными. Они учат нас важным принципам программирования и разработки.

Большая часть этих глав первоначально была опубликована в разделе «Жемчужины творчества программистов» (Programming Pearls) журнала *Communications of the Association for Computing Machinery*. Они были собраны, отредактированы и опубликованы в виде первого издания этой книги в 1986 году. Двенадцать из тринадцати глав первого издания были включены во второе, к ним добавились три новых.

Я предполагаю наличие у читателя опыта программирования на каком-либо языке высокого уровня. Сложные методы (вроде шаблонов в C++) тоже применяются в некоторых программах книги, но не знакомый с ними читатель всегда сможет без проблем перейти к следующему разделу.

Хотя любую главу можно читать отдельно от всех остальных, их можно логически разделить на несколько групп. Главы с первой по пятую образуют первую часть книги. В них рассматриваются основы программирования: постановка задачи, алгоритмы, структуры данных, а также верификация и тестирование программ. Вторая часть посвящена эффективности, которая важна не только сама по себе, но и в качестве отправной точки для дальнейших интересных исследований. В третьей части изученные методы применяются к нескольким важным задачам, относящимся к сортировке, поиску и строкам.

Один совет: не читайте слишком быстро. Читайте главы медленно, по одной за раз. Пробуйте решать задачи, когда вам предлагается это сделать. Некоторые из них кажутся простыми до тех пор, пока не поломаешь над ними голову несколько часов. Старайтесь решить все задачи, собранные в конце каждой из глав. Больше всего пользы от книги вы получите, придумывая собственные решения. Старайтесь обсуждать идеи и решения со своими коллегами, прежде чем подглядывать в подсказки и решения задач. Дополнительная литература в конце каждой главы — это не просто библиография. Я рекомендую читателям несколько хороших книг, которые составляют важную часть моей собственной библиотеки.

Эта книга написана для программистов. Я надеюсь, что задачи, подсказки, решения и список дополнительной литературы помогут изучать ее самостоятельно. Книга использовалась в курсах лекций по теории алгоритмов, верификации программ и программированию. Каталог алгоритмов в приложении 1 окажется полезным программистам-практикам и поможет использовать книгу при изучении алгоритмов и структур данных.

Программы

Я реализовал все программы, тексты которых были приведены в первом издании на псевдокоде, но получившиеся исходные коды видел только я один. Для этого издания я заново переписал все старые программы и добавил еще столько же новых. Тексты всех программ можно скачать по адресу:

http://netlib.bell-labs.com/cm/cs/pearls/

Среди прочего там находятся тестовые программы, предназначенные для проверки, отладки и измерения времени работы функций. На том же сайте можно найти еще много полезного материала. Поскольку сейчас можно свободно скачивать большое количество разнообразного программного обеспечения, это издание включает новую тему: оценка и использование компонент программного обеспечения.

Программы написаны в кратком стиле: короткие имена переменных, мало пустых строк, не проверяются ошибки. Это недопустимо в больших программных проектах, но весьма полезно для донесения основных идей алгоритмов. В решении 5.1 я пишу о своем стиле программирования более подробно.

В книге есть несколько настоящих программ на C и C++, но большая часть функций записана на псевдокоде, который занимает меньше места и исключает некрасивые элементы синтаксиса реальных языков. Запись `for i = [0, n)` означает перебор всех значений `i` от `0` до `n-1`. В подобных циклах круглые скобки обозначают, что граничное значение не включается в диапазон, а квадратные, напротив, показывают, что значение включено в диапазон. Запись `function(i, j)` все так же подразумева-

ет вызов функции с аргументами i и j, а запись array[i, j] — обращение к элементу массива с индексами i, j.

В этом издании для большинства программ приведены времена выполнения на «моем компьютере» — Pentium II 400 МГц со 128 Мбайт оперативной памяти под управлением Windows NT 4.0. Я измерял время выполнения программы и на некоторых других компьютерах, и все существенные отличия отражены в книге. Во всех экспериментах при компиляции включался максимально возможный уровень оптимизации. Я рекомендую вам измерить быстродействие программ на своем компьютере. Готов поспорить, что соотношение времен окажется приблизительно тем же.

Читателям первого издания

Надеюсь, что, пролистав это издание, вы скажете: «*Что-то уж больно похоже на то, что я уже читал раньше!*» А через несколько минут констатируете: «*Нет, эту книгу я никогда раньше не читал!*»

У второго издания те же цели, что и у первого, но контекст задач шире. Информационные технологии далеко продвинулись в таких важных областях, как базы данных, сети и пользовательские интерфейсы. Большинство программистов должны быть хорошо знакомы с этими технологиями. В центре каждой из областей находится ядро проблем программирования. Программы все так же остаются главной темой моей книги. Новое издание — это немного подросшая рыба в сильно расширившемся пруду.

Один из разделов старой главы 4, посвященный реализации двоичного поиска, разросся в главу 5, посвященную тестированию, отладке и измерению производительности. Старая глава 11 разрослась и разделилась на новую главу 12, посвященную той же задаче, что и старая глава 11, и главу 13, посвященную реализации наборов. В старой главе 13 описывалась программа проверки орфографии, работавшая в 64 Кбайт адресного пространства. От нее я отказался, но история осталась в разделе 13.8. Новая глава 15 посвящена задачам со строками. В старых главах прибавилось новых разделов, и одновременно некоторые другие разделы исчезли. С новыми задачами, новыми решениями и четырьмя новыми приложениями это издание книги выросло на 25%.

Многие старые ситуации перешли в это издание без изменений в силу их исторической ценности. Некоторые из старых историй были переписаны в современных терминах и величинах.

Благодарности к первому изданию

При написании и выпуске книги мне оказали поддержку многие, за что я им очень благодарен. Идея создания раздела в журнале Communications of the ACM первоначально возникла у Питера Деннинга (Peter Denning) и Стюарда Линна (Stuart Lynn). Питер усердно трудился в ACM над тем, чтобы воплотить свои мечты в реальность, и привлек меня к этой теме. Сотрудники «штаб-квартиры» ACM, в особенности Роз Стайер (Roz Steiter) и Нэнси Эйдриенс (Nancy Adriance), 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 нынешнем виде, а также многим читателям, которые своими комментариями к ним сделали возможным и необходимым их издание в развернутом виде.

Эл Ахо (Al Aho), Петер Деннинг (Peter Denning), Майк Гари (Mike Garey), Девид Джонсон (David Johnson), Брайан Керниган (Brian Kernighan), Джон Линдерман (John Linderman), Дуг Макилрой (Doug McIlroy) и Дон Станат (Don Stanat), несмотря на сильную занятость, прочитали все главы с великим вниманием. За особенно ценные комментарии я благодарю также Генри Бэрда (Henry Baird), Билла Кливленда (Bill Cleveland), Дэвида Гриза (David Gries), Эрика Гросса (Eric Grosse), Линн Джелински (Lynn Jelinski), Стива Джонсона (Steve Johnson), Боба Мелвилла (Bob Melville), Боба Мартина (Bob Martin), Арно Пензиаса (Arno Penzias), Криса Ван Вайка (Chris Van Wyk), Вика Высоцкого (Vic Vissotsky) и Памелу Зейв (Pamela Zave). Эл Ахо (Al Aho), Эндрю Хьюм (Andrew Hume), Брайан Керниган (Brian Kernighan), Рави Сети (Ravi Sethi), Лаура Скинтер (Laurs Skinger) и Бьярн Страуструп (Bjarne Stroustrup) оказали неоценимую помощь в подготовке книги, а курсанты Вэст-Пойнтского кадетского корпуса провели «полевые испытания» предпоследнего варианта рукописи этой книги. Спасибо всем.

Благодарности ко второму изданию

Дан Бентли (Dan Bentley), Русс Кокс (Russ Cox), Брайан Керниган (Brian Kernighan), Марк Керниган (Mark Kernighan), Джон Линдерман (John Linferman), Стив Маккоппелл (Steve McConnell), Дуг Макилрой (Doug McIlroy), Роб Пайк (Rob Pike), Говард Трики (Howard Trickey) и Крис Ван Вайк (Chris Van Wyk) прочитали новое издание очень внимательно. За особо ценные комментарии я благодарю Пола Абрахамса (Paul Abrahams), Гленду Чайлдресс (Glenda Childress), Эрика Гросса (Eric Grosse), Энн Мартин (Ann Martin), Петера Макилроя (Peter McIlroy), Петера Мемишиана (Peter Memishian), Сундара Нарасимхана (Sundar Narasimhan), Лизу Рикер (Lisa Ricker), Денниса Ритчи (Dennis Ritchie), Рави Сети (Ravi Sethi), Кэрол Смит (Carol Smith), Тома Сжимански (Tom Szymanski) и Кентаро Тояму (Kentaro Toyama). Спасибо Петеру Гордону (Peter Gordon) и его коллегам из Addison Wesley за помощь в подготовке этого издания.

Мюррей Хилл, Нью Джерси
Декабрь 1985
Август 1999
Д. Б.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу: http://www.piter.com/download.

На web-сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Часть

Предварительные сведения

В первых пяти главах содержится обзор основ программирования.

Глава 1 посвящена истории единственной задачи. В ней показывается, как сочетание правильной формулировки задачи с простыми приемами программирования может привести к элегантному решению. Глава иллюстрирует главную мысль всей книги: глубокие раздумья над обыденными жизненными ситуациями могут оказаться простой забавой, но могут принести и реальные практические выгоды.

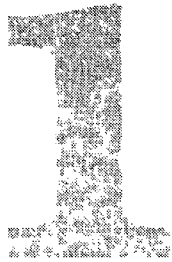
В главе 2 исследуются три задачи, показывающие, как размышления над алгоритмами могут привести к созданию простого и эффективного кода.

В главе 3 показывается, насколько важную роль в разработке программного обеспечения играет правильное структурирование данных.

В главе 4 мы рассмотрим верификацию программ как инструмент написания правильного кода. Верификация интенсивно используется в главах 9, 10 и 11 при создании и проверке небольших быстрых функций.

И наконец, в главе 5 мы доберемся до реализации наших абстрактных алгоритмов на языке C: проверим их на работоспособность, подвергнем испытаниям различными тестами и измерим их производительность.

Как расколоть орешек



Один программист задал мне простой вопрос: «Как отсортировать файл на диске?» Прежде чем я расскажу о своей первой ошибке, попробуйте ответить на этот вопрос лучше, чем в свое время это сделал я. Итак, что бы вы сказали?

1.1. Дружеский разговор

Моя ошибка состояла в том, что я ответил на вопрос, вкратце объяснив, как надо выполнять сортировку записей в дисковом файле *слиянием* (merge sort). Предложение углубиться в тонкости теории алгоритмов было встречено без энтузиазма — для него было важнее решить задачу, чем повысить уровень собственного образования. Тогда я рассказал ему о программе сортировки дисковых файлов из одной известной книги по программированию. В этой программе было около 200 строк кода, разбитого на 12 функций; по моим прикидкам, на написание и тестирование такой программы у этого программиста ушло бы не больше недели.

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

— Зачем вам понадобилось писать собственную программу сортировки? Почему для этой цели не воспользоваться системными средствами?

— Мне необходимо встроить функцию сортировки в большую программу, и по некоторым причинам я не могу воспользоваться системной программой для сортировки файла на диске.

— Что именно вы сортируете? Сколько записей в файле и каков их формат?

— Файл содержит не более десяти миллионов записей, каждая из которых представляет собой семизначное целое число.

— Подождите минутку. Если файл настолько мал, зачем вообще связываться с диском? Почему бы не отсортировать его в оперативной памяти?

— Хотя используемый компьютер и оснащен достаточно большим количеством оперативной памяти, эта функция сортировки является лишь малой частью большой программы. К моменту ее выполнения свободно будет лишь около мегабайта оперативной памяти.

— *Что-нибудь еще можно сказать об этих записях?*

— Каждая из них представляет собой положительное семизначное целое число без каких-либо связанных с ним дополнительных данных, и каждое число встречается в файле только один раз.

Теперь задача проясняется. В Соединенных Штатах номера телефонов состоят из трехзначного кода города (area code), за которым следуют семь цифр местного номера. Звонки по номерам с кодом 800 (единственным в то время) были бесплатными (toll-free). Существующие базы бесплатных номеров включали достаточно большое количество информации: бесплатный номер телефона, реальный номер, на который перенаправляется вызов (иногда несколько номеров с установленными правилами отбора), имя и адрес абонента и так далее.

Программист занимался созданием небольшого блока системы обработки такой базы данных, а сортировать он должен был именно эти телефонные номера. Входные данные представляли собой файл со списком телефонных номеров (вся прочая информация из файла удалялась), причем повторение какого-либо номера было бы ошибкой. На выходе нужно было получить отсортированный по возрастанию файл с номерами телефонов. Контекст задачи определяет также требования к производительности программы. Во время работы с системой пользователь должен был обращаться к этому файлу приблизительно раз в час, и до завершения операции сортировки он не мог бы продолжить работу. Следовательно, время, отведенное для сортировки, должно было быть минимальным, оптимально — не более десяти секунд.

1.2. Точная постановка задачи

Для программиста эти требования свелись к вопросу: «Как отсортировать дисковый файл?» Прежде чем мы займемся этой проблемой всерьез, попробуем представить исходные данные в более удобной форме:

- **входные данные:** файл, содержащий не более n положительных целых чисел, каждое из которых не превышает n, где $n=10^7$. Каждое из этих чисел должно встречаться не более одного раза. Связи этих чисел с другими данными не существует;
- **результат:** упорядоченный по возрастанию список поступивших на вход целых чисел;
- **ограничения:** наличие приблизительно одного мегабайта оперативной памяти; место на диске не ограничено. Время выполнения операции не должно превышать нескольких минут, если же оно будет сокращено до десяти секунд, дальнейшая оптимизация программы не требуется.

Подумайте над задачей в такой постановке. Что бы вы теперь посоветовали программисту?

1.3. Разработка программы

Очевидный вариант программы использует в качестве отправной точки сортировку слиянием общего вида, которая может быть затем оптимизирована с учетом того факта, что сортируются целые числа. Это сократит программу с двухсот до нескольких десятков строк и несколько ускорит ее выполнение. Однако на кодирование и отладку такой программы по-прежнему потребуется несколько дней.

Второй вариант решения учитывает детали конкретной задачи. Если мы отведем под каждое число семь байт, в доступном мегабайте оперативной памяти поместится 143 000 номеров. Если использовать 32-битное представление целых чисел, в тот же мегабайт поместится уже 250 000 номеров. Поэтому можно написать программу, которая будет считывать файл 40 раз. На первом проходе в память считываются все записи с порядковыми номерами от 0 до 249 999, которые затем сортируются и записываются в выходной файл. За второй проход сортируются записи с номерами от 250 000 до 499 999, а за 40-й — от 9 750 000 до 9 999 999. Для сортировки можно использовать алгоритм быстрой сортировки (quicksort), который занимает всего 20 строк кода (см. главу 11). Вся программа, таким образом, займет всего лишь один или два экрана. Кроме того, нам больше не придется беспокоиться о временных файлах; к сожалению, цена за эти улучшения — сорок операций чтения/записи файла.

Сортировка слиянием считывает входной файл один раз, сортирует его с использованием временных файлов, которые считываются и записываются много раз, а затем записывает результат за один проход (рис. 1.1).

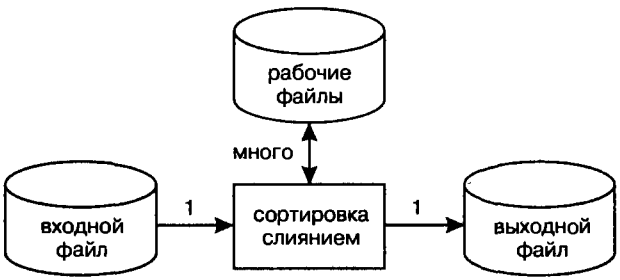


Рис. 1.1. Однопроходный алгоритм

Алгоритм с сорока проходами осуществляет считывание входного файла 40 раз, записывая результат за один раз, не используя никаких временных файлов (рис. 1.2).

Рис. 1.2. Сорокапроходный алгоритм

Оптимальной для нас была бы программа, работающая по следующей схеме, соединяющей преимущества предыдущих: входной файл считывается только один раз, и временные файлы не используются (рис. 1.3).

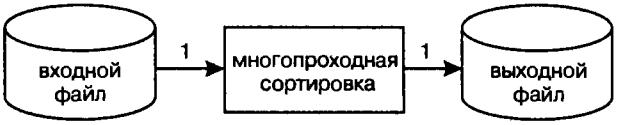


Рис. 1.3. Оптимальный алгоритм

Реализовать это можно, лишь отобразив все числа из входного файла в доступный мегабайт оперативной памяти. Таким образом, задача сводится к отображению не более чем десяти миллионов отличающихся друг от друга целых чисел в *приблизительно* восемь миллионов доступных бит. Попробуйте придумать подходящее отображение.

1.4. Набросок решения

В свете вышеизложенного решение с использованием битового массива напрашивается само собой. Набор неотрицательных целых чисел, не превышающих, к примеру, 20, может быть представлен строкой из 20 битов. Возьмем, допустим, набор {1, 2, 3, 5, 8, 13}. Представление его в виде битового массива будет выглядеть так:

```
01110100100001000000
```

Биты, соответствующие имеющимся числам, установлены, а все прочие — сброшены.

В реальной задаче семь десятичных знаков задают число, не превышающее 10 миллионов. Представим файл как строку из десяти миллионов битов[1], в которой i-й бит равен единице тогда и только тогда, когда в файле присутствует число i. В этом представлении используются три особенности данной задачи, нехарактерные для сортировки в общем виде: диапазон входных данных не слишком широк, одинаковых чисел быть не может и, кроме них, в записях больше ничего нет.

При использовании битового массива для представления набора целых чисел программа естественным образом разбивается на три части.

1. Массив инициализируется нулями.
2. Считываются целые числа из файла, и соответствующие биты в массиве устанавливаются в 1.
3. Формируется упорядоченный выходной файл (путем последовательной проверки значения каждого бита и записи в файл целых чисел, соответствующих ненулевым битам).

Если n — количество битов в массиве (в данном случае — 10 000 000), программа на псевдокоде будет выглядеть следующим образом:

```
/* Часть 1: инициализация массива нулями */
    for i = [0, n)
        bit[i] = 0
```

[1] Мой знакомый программист смог найти недостающие два миллиона битов. В задаче 5 в конце главы исследуется случай, в котором мегабайт доступной памяти — жесткое ограничение.

```
/* Часть 2: помещение элементов в набор */
    для каждого i во входном файле
        bit[i] = 1
/* Часть 3: формирование упорядоченного выходного файла */
    for i = [0, n)
        if bit[i] == 1
            записать i в выходной файл
```

Помните, что, согласно предисловию, обозначение for i=[0, n) соответствует перебору всех целых чисел от 0 до n–1.

Этого наброска оказалось достаточно, чтобы программист смог решить задачу. Некоторые проблемы, с которыми он столкнулся при реализации, описаны в задачах 2, 5 и 7 в конце главы.

1.5. Основные принципы

Программист рассказал мне о задаче по телефону. Нам понадобилось около пятнадцати минут, чтобы дойти до правильной постановки задачи и ее решения (с битовым массивом). Для реализации программы ему потребовалось несколько часов; заняла она несколько десятков строк кода, что было гораздо лучше, чем неделя программирования и сотни строк кода — а ведь в начале беседы он рассчитывал именно на это. Кроме того, программа была быстрой, как молния: сортировка слиянием выполняла бы эту задачу за десятки минут, тогда как в этой программе время тратилось в основном на считывание входного файла и запись результата — всего порядка десяти секунд. В решении 3 приведены данные о времени выполнения нескольких вариантов решения этой задачи.

Приведенные выше факты убедительно доказывают: внимательное изучение небольшой задачи может дать огромную практическую пользу. В данном примере несколько минут раздумий позволили уменьшить время кодирования, длину программы и время ее выполнения на порядок. Генерал Ч. Йегер (первый человек, превысивший скорость звука) охарактеризовал двигатель своего самолета следующими словами: «Простой, мало деталей, легко обслуживать, очень прочный»; практически теми же словами можно охарактеризовать и получившуюся программу. Однако эта программа предназначена для решения конкретной задачи, и изменить ее в соответствии с изменяющимися требованиями будет непросто. Этот пример является рекламой искусного программирования, прекрасно иллюстрирующей следующие общие принципы.

- *Правильно ставьте задачу.* Постановка задачи в этом случае определяла решение на 90% — я рад, что программист не удовлетворился первым предложенным вариантом. Задачи 10, 11 и 12 (см. раздел 1.6) сами подскажут вам красивое решение, как только вы сможете правильно поставить вопрос. Попробуйте сами найти верное и красивое решение прежде, чем читать подсказки.
- *Используйте битовые массивы в качестве структур данных.* Эти структуры используются для представления плотного набора элементов конечного множества, в который каждый элемент может входить не более одного раза и где с элементами не связано никаких дополнительных данных. Даже если данные условия в какой-либо другой задаче не выполняются (могут быть повторы

элементов или имеются дополнительные данные), для индексирования таблицы со сложными полями все равно часто может быть использован ключ из конечного набора (см. задачи 6 и 8).

- *Применяйте многопроходные алгоритмы.* Такие алгоритмы обрабатывают входные данные за несколько проходов, выполняя за каждый проход часть работы. В разделе 1.3 мы встретились с примером 40-проходного алгоритма; задача 5 потребует написания двухпроходного.
- *Ищите возможности избежать компромисса «время-память».* Фольклор программистов и теория программирования содержат множество примеров компромисса «время-память». Например, программа может требовать меньшего объема памяти, но работать большее время. Двухпроходный алгоритм в задаче 5 увеличивает время работы вдвое, сокращая в то же число раз необходимую память. Однако, по моему опыту, зачастую сокращение требований к памяти сокращает и время работы программы[1]. Эффективная с точки зрения экономии памяти структура битового массива привела к значительному сокращению времени сортировки. Это произошло по двум причинам: во-первых, меньшее количество данных требует меньшего времени для обработки, а во-вторых, хранение данных в памяти, а не на диске сокращает затраты на обращение к ним. Разумеется, столь значительные улучшения оказались возможными лишь потому, что начальная версия программы была далека от оптимальной.
- *Стремитесь к простоте.* 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». Многим программистам следует оценивать свою работу по этому критерию. Простые программы обычно более надежны и эффективны, чем их более сложные варианты, их проще создавать и сопровождать.
- *Разбейте разработку программы на этапы.* Данный пример иллюстрирует процесс разработки, подробно описанный в разделе 12.4.

1.6. Задачи

Подсказки к задачам и их решения собраны в нескольких разделах в конце книги.

1. Если бы памяти было достаточно, как бы вы реализовали сортировку на языке, в котором доступны библиотеки для представления и сортировки[2] наборов?
2. Как бы вы реализовали битовый массив с использованием побитовых логических операций (таких, как И, ИЛИ, сдвиг)?

---

[1] С компромиссами подобного рода можно столкнуться в любой инженерной деятельности. Например, коиструкторы автомобилей могут увеличить пробег за счет быстроты разгона, добавив более тяжелые (и надежные) детали. Однако лучше стремиться к одновременному улучшению всех характеристик. В описании машины, на которой я некоторое время ездил, было сказано: «Уменьшение веса корпуса ведет к облегчению ходовой части и даже к возможности удаления некоторых ее деталей, таких как гидроусилитель руля».

[2] Например, стандартные библиотеки языка Си. — *Примеч. пер.*

3. Быстродействие программы было одним из главных требований при постановке задачи. Получившаяся версия оказалась достаточно эффективной. Реализуйте сортировку битового массива в своей системе и сравните скорость ее работы с библиотечной функцией сортировки, а заодно с сортировками из задачи 1. Пусть n=10 000 000, а входной файл содержит 1 000 000 целых чисел.

4. Если вы серьезно подошли к решению задачи 3, вам придется столкнуться с задачей, заключающейся в генерировании k неповторяющихся натуральных чисел, меньших n. Простейший вариант — взять первые k натуральных чисел и записать их в файл. Такой подход не изменит скорости работы сортировки с битовым массивом, но может исказить результаты измерения скорости работы библиотечной функции сортировки. Каким образом вы сформируете файл с k неповторяющимися случайными целыми числами из диапазона 0..n-1? Попытайтесь написать короткую и эффективную программу.

5. Программист сказал, что под эту программу можно занять около мегабайта памяти, но предложенный нами вариант использовал около 1,25 мегабайта. Программисту удалось выделить необходимый объем оперативной памяти. Если бы мегабайт был жестким ограничением в использовании оперативной памяти, какое решение вы бы тогда предложили? За какое количество проходов выполнялся бы предложенный вами алгоритм?

6. Что бы вы посоветовали программисту, если бы каждое семизначное число его задачи могло повторяться, скажем, не более десяти раз? Как в этом случае меняется предлагаемое вами решение в зависимости от количества доступной памяти?

7. [R. Weil] Предложенная программа имеет несколько недостатков. Во-первых, предполагается отсутствие повторов в массиве входных данных. Что произойдет, если какое-либо число встретится дважды? Как можно изменить программу, чтобы обработать такую ситуацию? Что произойдет, если на входе окажется отрицательное целое число или положительное, но большее либо равное n? Что, если на входе будет вообще не число? Как следует обрабатывать такие ситуации? Какие еще проверки могла бы включать программа? Опишите небольшие наборы данных, которые могут быть использованы для проверки работоспособности программы, включая корректную обработку описанных выше (и других возможных типов) некорректных данных.

8. Когда эта задача была поставлена перед программистом, все бесплатные номера в США начинались с кода 800. Сейчас эти номера могут иметь код 800, 877 и 888, причем список таких кодов растет. Как бы вы отсортировали все эти номера, используя лишь один мегабайт оперативной памяти? Как бы вы хранили набор бесплатных номеров, если требуется обеспечить возможность быстрой проверки наличия какого-либо номера в списке?

9. Одна из проблем, с которыми приходится сталкиваться при использовании большего объема памяти для сокращения времени работы программы, заключается в том, что на инициализацию памяти также может уйти достаточно много времени. Попробуйте обойти эту проблему, разработав метод, позволяющий инициализировать вектор нулями при первом к нему обращении.

Ваша программа должна требовать постоянного времени на инициализацию и доступ к массиву и использовать добавочную память, пропорциональную размеру массива. Поскольку этот метод ускоряет инициализацию, используя еще больше памяти, он может быть применен только в тех случаях, когда ресурсы оперативной памяти не ограничены, а время дорого и массив является разреженным.

10. До того как пришли времена дешевой доставки товаров, некий магазин предлагал покупателям следующую услугу: они могли заказать товары по телефону и заехать за ними через несколько дней. База данных в этом магазине использовала телефонный номер покупателя в качестве главного ключа доступа для получения товара (получатели знают свои телефонные номера, и ключи оказываются практически уникальными). Как бы вы организовали базу данных этого магазина, предоставив возможность быстрого создания и удаления заказов?
11. В начале 80-х инженерам компании Локхид (Lockheed) приходилось каждый день передавать около десятка чертежей из чертежного отдела в Саннивейле (Sunnyvale) в Калифорнии на испытательную базу в Санта-Крузе. Хотя между отделами было всего около 25 миль, доставка курьером на автомобиле требовала около часа (с учетом горной местности и загруженности трассы), а стоила она сотню долларов в день. Предложите альтернативные способы передачи чертежей и оцените их стоимость.
12. В эпоху космических исследований возникла проблема использования письменных принадлежностей в невесомости. Согласно известной легенде, американское Национальное агентство по аэронавтике и космическим исследованиям, NASA, потратило на решение проблемы миллион долларов, разработав специальную авторучку. Что, по той же легенде, было сделано в СССР?

1.7. Дополнительная литература

В этой главе мы едва коснулись увлекательной темы постановки задач для программистов. Чтобы глубже разобраться в этом вопросе, обратитесь к книге Майкла Джексона (Michael Jackson, Software Requirements & Specifications, Addison-Wesley, 1995). Сложные темы в этой книге раскрыты в форме небольших эссе, независимых, но связанных общей идеей.

Проблема программиста, описанная в этой главе, была не столько технической, сколько психологической: он не мог продвинуться в решении, поскольку изначально поставил перед собой неверно сформулированную задачу. Мы нашли решение, сняв концептуальный психологический блок, и пришли к более простой постановке задачи. Книга Джеймса Адамса (James L. Adams, Conceptual Blockbusting, Perseus, 1986) посвящена изучению подобных прорывов и помогает перейти к более созидательному стилю мышления. Хотя изначально эта книга не была предназначена для программистов, многое в ней может быть применено к решению наших задач. Адамс определяет концептуальные блоки как «ментальные стены, не дающие решающему правильно воспринимать задачу и прийти к ее решению». Приведенные ранее задачи 10, 11 и 12 дадут вам возможность попробовать сломать несколько таких «стен».

Ага! Алгоритмы

Изучение теории алгоритмов может оказаться очень полезным программисту-практику. Курс лекций по этому предмету снабжает студентов функциями для решения важных задач и учит их находить подходы к решению новых задач. В следующих главах мы увидим, каким образом более совершенные алгоритмы могут влиять на программное обеспечение, сокращая время на разработку и повышая эффективность программ.

Алгоритмы играют существенную роль в решении обычных задач программирования. В своей книге «Ага! Озарение» (Martin Gardner, Aha! Insight), откуда я бессовестно украл заглавие, Мартин Гарднер описывает то, что я имею в виду: «Кажущаяся сложной задача может иметь простое неожиданное решение». В отличие от более совершенных методов, «ага!-алгоритмы» не требуют высшего образования; они могут прийти в голову любому программисту, готовому серьезно задуматься, до написания программы, в процессе ее кодирования или после этого.

2.1. Три задачи

Довольно общих слов! Эта глава посвящена трем небольшим задачам. Попробуйте решить их самостоятельно, прежде чем продолжить ознакомление с этой главой.

1. Пусть есть некий файл, содержащий не более четырех миллиардов 32-битных целых чисел в случайном порядке. Нужно найти число, которое отсутствует в файле (а хотя бы одно число будет отсутствовать наверняка — почему?). Как бы вы решили эту задачу при наличии неограниченной оперативной памяти? Как вы решите ее, если оперативная память ограничена парой сотен килобайт, но разрешается использование нескольких временных файлов?
2. Осуществите циклический сдвиг одномерного массива из `n` элементов на `i` позиций влево. Например, если `n=8`, а `i=3`, вектор `abcdefgh` превращается

в defghabc. Простейшая программа использует n-элементный вспомогательный массив и выполняет задачу за n шагов. Можете ли вы сдвинуть массив за время, пропорциональное n, используя лишь несколько десятков байт под дополнительные переменные?

3. Дан словарь английского языка. Требуется найти все наборы анаграмм. Например, слова pots, stop и tops являются анаграммами друг для друга, поскольку каждое из этих слов может быть получено перестановкой букв любого другого.

2.2. Вездесущий двоичный поиск

Я задумываю целое число от 1 до 100, а вы его угадываете. 50? Мало. 75? Много. И так игра продолжается до тех пор, пока вы не угадаете задуманное мною число. Если число взято из диапазона 1..n, то оно может быть наверняка угадано за $\log_2 n$ попыток. Если n=1000, достаточно будет 10 попыток, а если n=1 000 000, потребуется не более 20 попыток.

Этот пример иллюстрирует метод, позволяющий решить множество задач программирования, — *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 текущей позиции, выше или ниже ее. При двоичном поиске местоположение объекта обнаруживается с помощью повторяющегося выбора элемента из середины текущей области. Если выбранный элемент отличается от искомого, текущая область делится пополам, после чего процесс выбора и сравнения повторяется. Поиск закончен, если обнаруживается искомый элемент или пустая область.

Самое обычное применение двоичного поиска — поиск элемента в отсортированном массиве. При поиске числа 50 будут проверены следующие элементы (рис. 2.1).

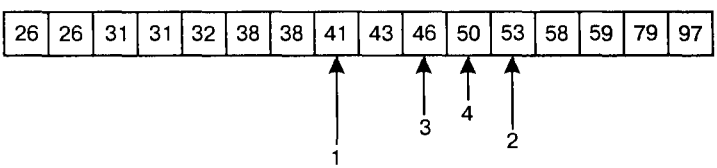


Рис. 2.1. Двоичный поиск в отсортированном массиве

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

Программа последовательного поиска выполняет в среднем около n/2 сравнений при поиске в наборе из n элементов, тогда как двоичный поиск всегда делает не более $\log_2 n$. Этот факт может очень сильно повлиять на производительность системы. Вот пример, относящийся к системе бронирования билетов авиакомпании TWA:

«Одна из программ осуществляла последовательный поиск в огромном объеме памяти примерно 100 раз в секунду. По мере роста сети среднее время обработки сообщения возросло до 0,3 мс, что оказалось для нас слишком много. Мы установили,

что проблема заключается в использовании процедуры последовательного поиска, заменили ее процедурой двоичного поиска, и проблема исчезла».

Я часто сталкивался с этим и в других системах. Программисты начинают с простой структуры данных и последовательного поиска, который поначалу оказывается достаточно быстрым. Когда производительности подобной программы начинает не хватать, можно отсортировать таблицу и воспользоваться двоичным поиском, что устраняет проблему.

Однако поле применения двоичного поиска не ограничивается быстрым поиском в отсортированных массивах. Рой Вейл (Roy Weil) применил этот метод для нахождения некорректной строки во входном файле, содержащем более 1000 строк. К сожалению, строку эту нельзя было обнаружить по ее внешнему виду; узнать о ее наличии можно было, лишь обработав файл (начальную часть до произвольного места) некой программой и получив некорректный ответ, на что требовалось несколько минут. Его предшественники, пытавшиеся найти ошибку, пробовали обрабатывать несколько строк за один проход, благодаря чему продвигались к цели со скоростью улитки. Попробуйте догадаться, как удалось Вейлу найти ошибку за десять попыток?

Теперь, когда мы слегка размялись, можно обратиться к задаче 1. На вход поступает последовательный файл (записанный на каком-либо носителе или на диске — хотя к диску и можно обращаться произвольно, последовательное чтение обычно осуществляется с гораздо большей скоростью). Этот файл содержит не более четырех миллиардов 32-битных целых чисел в произвольном порядке, а нам необходимо найти число, отсутствующее в этом файле. Такое число обязательно есть, поскольку 32-битных целых чисел всего 2^{32}, или 4 294 967 296. При наличии неограниченного объема памяти мы могли бы воспользоваться битовым массивом, описанным в главе 1, и выделить 536 870 912 байт (по 8 бит каждый) под все возможные целые числа. Однако нам предлагается найти недостающее число, используя лишь несколько сот байт оперативной памяти и несколько вспомогательных файлов последовательного доступа. Чтобы реализовать двоичный поиск в этой ситуации, прежде всего нужно определить диапазон, представление элементов диапазона и метод проверки, позволяющий определить, в какой половине этого диапазона не хватает числа.

Диапазоном у нас будет набор целых чисел, в котором заведомо недостает по крайней мере одного элемента, а представлять мы его будем в виде файла, содержащего все числа этого диапазона. И вот: озарение! Мы можем проверять диапазон, подсчитывая количество элементов, меньших и больших значения середины диапазона. Либо в верхней, либо в нижней половине диапазона будет не более половины общего количества чисел в диапазоне. Поскольку хотя бы один элемент во всем диапазоне обязательно отсутствует, в одной из половин диапазона также будет отсутствовать по крайней мере один элемент, и в этой половине будет меньшее число элементов. Вот основные составляющие алгоритма двоичного поиска. Попробуйте соединить их вместе самостоятельно, прежде чем подсмотреть ответ Эда Рейнгольда (Ed Reingold).

В этих двух примерах мы лишь слегка коснулись необъятной области применения двоичного поиска в программировании. При поиске корней уравнения

с одной переменной можно использовать метод *бисекции* — последовательного деления интервала, содержащего корень, на равные части. Алгоритм в решении задачи 9 из главы 11 выбирает случайный элемент, а затем рекурсивно вызывается для области по одну сторону от выбранного элемента. Такой алгоритм называется *случайным* (иногда его называют «*рандомизованным*») *двоичным поиском*. Другие приложения двоичного поиска включают деревья (структуры данных) и отладку программ. (Если программа не выводит сообщения об ошибке при внезапном завершении работы, как вы будете вести поиск ошибки?) Во всех подобных случаях программисту может помочь понимание того факта, что все подобные программы являются всего лишь версиями базового алгоритма двоичного поиска.

2.3. Мощь элементарного

Мы нашли решение: двоичный поиск. Теперь попробуем поискать задачи для этого решения. Рассмотрим задачу, которая может иметь несколько решений. Задача 2 заключается в циклическом сдвиге массива x из n элементов влево на i позиций за время, пропорциональное n, причем доступно лишь несколько десятков байт оперативной памяти. В некоторых языках программирования операция циклического сдвига является элементарной (то есть выполняется одним оператором). Для нас важно, что циклический сдвиг соответствует обмену соседних блоков памяти разного размера: при перемещении фрагмента текста с помощью мыши из одного места файла в другое осуществляется именно эта операция. Ограничения по времени и объему памяти существенны для многих подобных приложений.

Можно попытаться решить задачу, копируя первые i элементов массива x во временный массив, сдвигая оставшиеся n-i элементов влево на i позиций, а затем копируя данные из временного массива обратно в основной массив на последние i позиций. Однако данная схема использует i дополнительных переменных, что требует дополнительной памяти. Другой подход заключается в том, чтобы определить функцию, сдвигающую массив влево на один элемент (за время, пропорциональное n), а потом вызывать ее i раз, но такой алгоритм будет отнимать слишком много времени.

Решение проблемы с указанными ограничениями на использование ресурсов потребует написания более сложной программы. Одним из вариантов решения будет введение дополнительной переменной. Элемент x[0] помещается во временную переменную t, затем x[i] помещается в x[0], x[2*i] — в x[i] и так далее (перебираются все элементы массива x с индексом по модулю n), пока мы не возвращаемся к элементу x[0], вместо которого записывается содержимое переменной t, после чего процесс завершается. Если i = 3, а n = 12, этот этап проходит следующим образом (рис. 2.2).

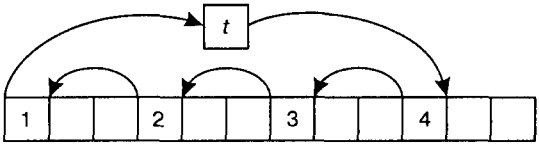


Рис. 2.2. Вариант решения задачи с циклическим сдвигом

Если при этом не были переставлены все имеющиеся элементы, процедура повторяется, начиная с x[1] и так далее, до достижения конечного результата. В задании 3 вам предлагается воплотить это решение в виде кода. Будьте аккуратны.

Можно предложить и другой алгоритм, который возникает из рассмотрения задачи с другой точки зрения. Циклический сдвиг массива x сводится фактически к замене ab на ba, где a — первые i элементов x, а b — оставшиеся элементы. Предположим, что a короче b. Разобьем b на b_l и b_r, где b_r содержит i элементов (столько же, сколько и a). Поменяем местами a и b_r, получим $b_r b_l a$. При этом a окажется в конце массива — там, где и полагается. Поэтому можно сосредоточиться на перестановке b_r и b_l. Эта задача сводится к начальной, поэтому алгоритм можно вызывать рекурсивно. Программа, реализующая этот алгоритм, будет достаточно красивой (в решении к заданию 3 описывается итерационное решение Гриса (Gries) и Миллса (Mills)), но она требует аккуратного написания кода, а оценить ее эффективность непросто.

Задача кажется сложной, пока вас не осенит озарение («ага!»): итак, нужно преобразовать массив ab в ba. Предположим, что у нас есть функция reverse, переставляющая элементы некоторой части массива в противоположном порядке. В исходном состоянии массив имеет вид ab. Вызвав эту функцию для первой части, получим a'b. Затем вызовем ее для второй части: получим a'b'. Затем вызовем функцию для всего массива, что даст (a'b')', а это в точности соответствует ba. Посмотрим, как будет такая функция действовать на массив abcdefgh, который нужно сдвинуть влево на три элемента:

```
reverse(0, i-1)  /* cbadefgh */
reverse(i, n-1)  /* cbahgfed */
reverse(0, n-1)  /* defghabc */
```

Дуг Макилрой (Doug McIlroy) предложил наглядную иллюстрацию циклического сдвига массива из десяти элементов вверх на пять позиций (рис. 2.3); начальное положение: обе руки ладонями к себе, левая над правой.

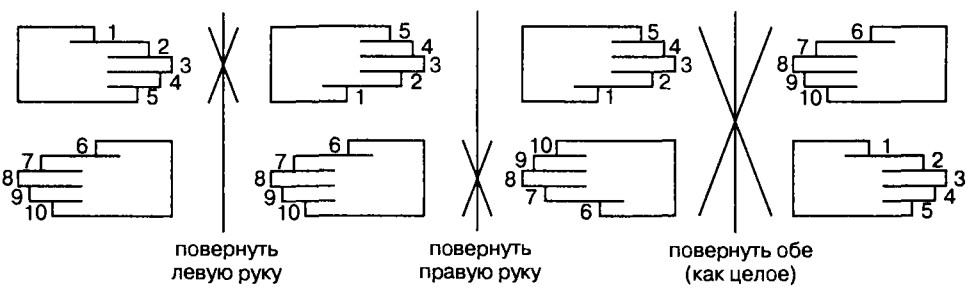


Рис. 2.3. Иллюстрация циклического сдвига

Код, использующий функцию переворота, оказывается эффективным и малотребовательным к памяти, и настолько короток и прост, что при его реализации сложно ошибиться. Б. Керниган и П. Дж. Плоджер пользовались именно этим методом для перемещения строк в текстовом редакторе в своей книге (B. Kernighan, P. J. Plauger, Software Tools in Pascal, 1981). Керниган пишет, что эта функция за-

работала правильно с первого же запуска, тогда как их предыдущая версия, использовавшая связный список, содержала несколько ошибок. Этот же код используется в некоторых текстовых редакторах, включая тот, в котором я впервые набрал настоящую главу. Кен Томпсон (Ken Thompson) написал этот редактор с функцией reverse в 1971 году, и он утверждает, что она уже тогда была легендарной.

2.4. Соберем все вместе: сортировка

Вернемся к задаче 3. Дается словарь английского языка (по одному слову в каждой строке входного файла), нужно найти все возможные анаграммы. Существует несколько причин, по которым стоит заняться этой задачей. Во-первых, решение будет красивой комбинацией правильного подхода и правильных методов. Важность второй причины в некоторых ситуациях неоспорима: вам же не хочется оказаться единственным человеком на вечеринке, который не знает, что deposit, dopiest, posited и topside — анаграммы? А если этих причин недостаточно, в задаче 6 будет описана аналогичная проблема, возникающая в реальной жизни.

Множество предложенных подходов к решению задачи анаграмм оказываются на практике неэффективными и сложными. Любой метод, рассматривающий все перестановки букв каждого слова, обречен. Слово cholecystoduodenostomy (являющееся анаграммой duodenocholecystostomy из моего словаря) дает 22 (!) перестановки, что приблизительно равно $1{,}124\times10^{21}$. Даже если предположить, что проверки будут выполняться со скоростью самого быстродействующего компьютера, выполняющего 10^{12} перестановок в секунду, — обработка одного слова займет $1{,}1\times10^{9}$ секунд. Одно из правил большого пальца[1] (см. раздел 7.1) гласит, что «π секунд равно 1 нановеку», то есть $1{,}1\times10^{9}$ секунд — это почти тридцать пять лет! Любой метод, сравнивающий все пары слов, потребует по меньшей мере целую ночь работы на моем компьютере. В словаре, который я использовал, содержится 230 000 слов, а даже простейшее сравнение анаграмм требует не меньше 1 мкс, поэтому полное время работы составит приблизительно:

```
230 000 слов * 230 000 сравнений/слово * 1 мкс/сравнение = 52 900*106
микросекунд = 52 900 секунд = 14.7 часа
```

Можно ли найти способ обойти эти вычислительные ловушки?

Озарение «ага!»: придумать способ вычисления сигнатуры слова таким образом, чтобы все слова, принадлежащие к одному классу анаграмм, имели одинаковую сигнатуру, затем собрать эти слова вместе. При этом исходная задача разделяется на две подзадачи: выбор сигнатуры и объединение всех слов с одинаковой сигнатурой. Подумайте над этим самостоятельно, прежде чем читать дальше.

Для решения первой подзадачи воспользуемся сортировкой: упорядочим буквы слова в алфавитном порядке[2]. Сигнатурой слова deposit будет deiopst, причем

---

[1] Эмпирические правила приближенных вычислений — *Примеч. перев.*

[2] Этот алгоритм нахождения анаграмм был независимо открыт несколькими программистами приблизительно в середине 60-х годов.

это же сочетание букв является также сигнатурой слова dopiest и любого другого слова из этого класса анаграмм. Для решения второй подзадачи упорядочим слова так, чтобы их сигнатуры располагались в алфавитном (лексикографическом) порядке. Лучшая иллюстрация была предложена Томом Каргиллом (Tom Cargill): сортировать нужно сначала в этом порядке (взмах рукой слева направо), затем в этом (сверху вниз). В разделе 2.8 описана реализация этого алгоритма.

2.5. Принципы

Сортировка

Наиболее очевидная цель применения сортировки — получение отсортированного результата, ценного либо уже самого по себе, либо в качестве промежуточного массива для дальнейшей обработки (например, двоичного поиска). В задаче об анаграммах важным было не столько упорядочение при сортировке, сколько группировка одинаковых элементов (сигнатур). Получение сигнатур является еще одной из возможных целей сортировки: упорядочение букв в слове дает его каноническую форму. Добавление ключей к записям и последующая сортировка по этим ключам дают возможность использовать сортировку для упорядочения данных в дисковых файлах. В разделе 3 мы еще несколько раз вернемся к сортировке.

Двоичный поиск

Алгоритм поиска элемента в упорядоченном наборе оказывается на редкость эффективным и может применяться к данным, находящимся в памяти или на диске. Его единственным недостатком является то, что доступной и заранее отсортированной должна быть вся таблица. Идея, на которой основывается этот простой алгоритм, используется во многих приложениях.

Сигнатуры

Когда отношение эквивалентности используется для определения классов, оказывается полезным определить сигнатуру, отличающую эти классы друг от друга. Сортировка букв слова дает одну из возможных сигнатур для анаграмм, другие варианты могут быть получены, к примеру, путем сортировки и последующей замены повторяющихся букв цифрами, указывающими их количество в слове. Сигнатурой слова mississippi в этом случае будет i4m1p2s4 или i4mp2s4 (если единицу по умолчанию не указывать). Можно также хранить массив из 26 целых чисел (для каждого слова), который будет содержать информацию о количестве повторяющихся букв. Другие области применения сигнатур включают используемый ФБР метод индексации отпечатков пальцев и эвристические алгоритмы Soundex для объединения имен, одинаково звучащих, но по-разному пишущихся (табл. 2.1).

Кнут описывает метод Soundex в главе 6 третьего тома книги «Искусство программирования для ЭВМ, сортировка и поиск».

Таблица 2.1. Сигнатуры одинаково звучащих имен

Имя	Сигнатура Soundex
Smith	s530
Smythe	s530
Schultz	s243
Shultz	s432

Постановка задачи

В главе 1 было наглядно показано, что доскональное выяснение требований заказчика является основополагающей частью процесса написания программы. Тема этой главы — следующая стадия процесса написания программы, а именно поиск ответа на вопрос: «Какими базовыми средствами может быть реализована программа?» Во всех примерах суть озарения состояла в использовании новой базовой операции, заметно упрощающей задачу.

Перспективы для программиста

Лень — двигатель прогресса. Хорошие программисты всегда немножко ленивы. Они сидят и ждут озарения, вместо того, чтобы воплощать в жизнь первую пришедшую им в голову идею. Лень, однако, должна компенсироваться вовремя возникающим желанием писать программу. Самое сложное — это определить, когда же наступает это должное время. Помочь тут может только опыт решения задач и анализа решений.

2.6. Задачи

1. Решите задачу нахождения всех анаграмм одного слова. Как бы вы ее решили, если бы вам дали только это слово и словарь? Что, если бы у вас была возможность обработать словарь, прежде чем искать анаграммы?
2. Дан последовательный файл, содержащий 4 300 000 000 32-битных целых чисел. Как найти целое число, которое встречается дважды?
3. Мы кратко упомянули о двух алгоритмах циклического сдвига массива, требовавших аккуратного кодирования. Реализуйте их. Как в этих программах используется наибольший общий делитель i и n?
4. Несколько читателей обратили внимание на то, что, хотя все три предложенных алгоритма работают за время, пропорциональное n, алгоритм обмена (см. раздел 2.3) оказывается примерно вдвое быстрее алгоритма реверсирования: он сохраняет во временный файл и возвращает обратно каждый элемент лишь один раз, в то время как алгоритм реверсирования делает это дважды. Поэкспериментируйте с функциями, чтобы проверить скорость их работы на реальных машинах; при этом обязательно нужно учитывать, к какой области памяти происходит обращение.

5. Функции сдвига массива меняют ab на ba. Как бы вы преобразовали вектор abc к виду cba? Эта задача моделирует обмен областей памяти, не прилегающих друг к другу.
6. В конце 70-х в лабораториях Белл была разработана программа «Справочной службы, управляемой пользователем», которая позволяла предпринимателям находить номера в телефонном справочнике, используя стандартный телефон с кнопочным набором (рис. 2.4).

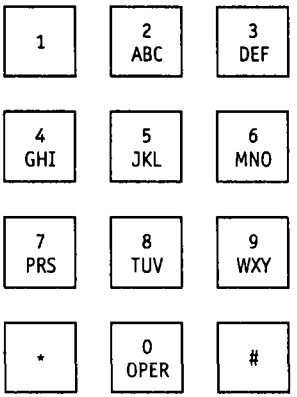


Рис. 2.4. Клавиатура кнопочного телефона

Чтобы найти номер разработчика системы М. Леска (Mike Lesk), нужно было набрать номер LESK*M* (то есть 5375*6*), и система озвучивала его номер. Сейчас подобные службы распространены повсеместно. Проблема может возникнуть, если несколько имен имеют один и тот же код; когда такое случалось в системе Леска, его программа запрашивала у пользователя дополнительную информацию. Если дан большой список имен наподобие обычного телефонного справочника, как бы вы выделили подобные совпадения? Леск провел подобный эксперимент и выяснил, что количество совпадений составляло всего лишь 0,2%. Как бы вы реализовали функцию, возвращающую список имен, задаваемых кодом, подаваемым на ее вход?

7. В начале 60-х Вик Высоцкий (Vic Vissotski) работал с программистом, которому нужно было транспонировать матрицу 4000 на 4000 элементов, хранившуюся на магнитной ленте. Все записи имели один и тот же формат и были длиной порядка нескольких десятков байт. Первому варианту программы, который был предложен этим программистом, потребовалось бы около 50 часов на выполнение задачи. Как удалось Высоцкому уменьшить это время до получаса?
8. [J. Ullmam] Дан набор из n вещественных чисел, вещественное число t, целое k. Как можно быстро определить существование k-элементного поднабора, сумма элементов которого не превышает t?

9. Последовательный поиск выполняется медленно, но не требует предварительной обработки, тогда как двоичный поиск работает очень быстро, но требует наличия отсортированного набора. Сколько нужно выполнить двоичных поисков на n-элементном массиве, чтобы окупить время, затраченное на его предварительную сортировку?
10. Когда к Томасу Эдисону обратился очередной претендент на должность помощника, Эдисон предложил ему вычислить объем пустой колбы от лампочки. Через несколько часов вычислений с кронциркулем в руках претендент вернулся с результатом — 150 $см^3$. После нескольких секунд вычислений Эдисон ответил «ближе к 155». Как он это сделал?

2.7. Дополнительная литература

В разделе 8.8 главы 8 приведены ссылки на несколько хороших книг по теории алгоритмов.

2.8. Реализация поиска анаграмм

Для решения этой задачи я написал три небольшие программы, соединив их в канал обработки. При этом выводимый одной из программ текст поступал на вход второй программы. Первая программа определяет сигнатуры слов, вторая — сортирует, а третья — объединяет слова одного класса в одной строке. Вот пример обработки словаря из шести слов (рис. 2.5).

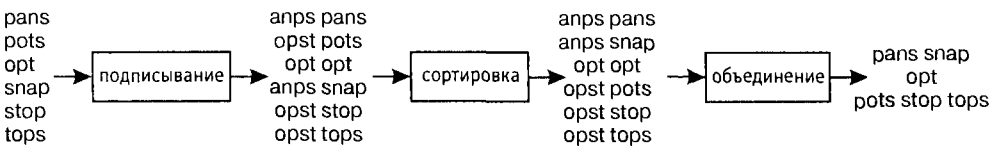


Рис. 2.5. Пример обработки словаря из 6 слов

В результате получаем три класса анаграмм.

В приведенной ниже программе на C (см. листинг 2.1) предполагается, что ни одно слово не состоит более чем из 100 букв и входной файл содержит только строчные буквы и символы перевода строки. (Поэтому перед ее запуском пришлось пропустить словарь через программу, приводившую все символы к нижнему регистру).

Листинг 2.1. Программа подписывания анаграмм

```
int charcomp(char *x, char *y) { return *x - *y; }
#define WORDMAX 100
int main(void)
```

```
{
  char word[WORDMAX], sig[WORDMAX];
  while (scanf("%s", word) != EOF) {
    strcpy(sig, word);
    qsort(sig, strlen(sig), sizeof(char), charcomp);
    printf("%s %s\n", sig, word);
  }
  return 0;
}
```

Цикл while считывает по одной строке в переменную word, пока не доберется до конца файла. Функция strcpy копирует строку в переменную sig, символы которой затем сортируются с помощью стандартной библиотечной функции qsort (аргументы: сортируемый массив, его длина, длина элемента, указатель на функцию сравнения двух элементов). Наконец, printf выводит сигнатуру и слово в файл, завершая строку символом перевода строки.

Для сортировки используется системная программа сортировки, а написанная мной программа squash объединяет анаграммы одного класса в одной строке.

Листинг 2.2. Объединение анаграмм одного класса в одной строке

```
int main(void)
{
  char word[WORDMAX], sig[WORDMAX], oldsig[WORDMAX];
  int linenum=0;
  strcpy(oldsig, "");
  while (scanf("%s %s", sig, word) != EOF) {
    if (strcmp(oldsig, sig)!=0 && linenum > 0)
      printf("\n");
    strcpy(oldsig, sig);
    linenum++;
    printf("%s", word);
  }
  printf("\n");
  return 0;
}
```

Основная работа выполняется вторым оператором printf — он выводит второе поле каждой строки и разделитель (пробел). Оператор if определяет момент смены сигнатуры: если sig отличается от oldsig (предыдущего значения сигнатуры), в выходной файл записывается символ перевода строки (если эта запись не первая в файле). Последний оператор printf записывает в конечный файл символ перевода строки.

После проверки этих небольших программ на тестовых файлах я получил список анаграмм из большого словаря, набрав:

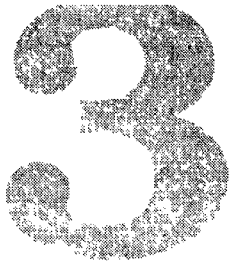
```
sign <dictionary | sort | squash >gramlist
```

Эта команда скармливает файл dictionary программе sign, направляет ее вывод на вход программы sort, вывод последней направляется на вход squash, а результаты ее работы сохраняются в файле gramlist. Программа выполнялась всего 18 секунд: 4 на sign, 11 на sort и 3 на squash.

Я применил эту программу к словарю, содержавшему 230 000 слов. Он, однако не включал многие окончания типа -s и -ed. А вот самые интересные классы анаграмм:

```
subessential suitableness
canter creant cretan nectar recant tanrec trance
caret carte cater crate creat creta react recta trace
destain instead sainted satined
adroitly dilatory idolatry
least setal slate stale steal stela tales
reins resin rinse risen serin siren
constitutionalism misconstitutional
```

Программы и структуры данных



Большинство программистов уже знакомы с ними. Любой хороший программист понимает, что он в своей жизни написал по крайней мере одну подобную программу. Это огромные, запутанные, уродливые создания, которые должны были бы быть короткими, ясными и красивыми. Я видел несколько программ, которые сводились бы к коду наподобие такого:

```
if (k == 1) c001++;
if (k == 2) c002++;
...
if (k == 500) c500++;
```

Хотя программы эти решали несколько более сложные задачи, не будет ошибкой предположить, что их целью было вычислить, сколько раз в некотором файле встречается каждое целое число от 1 до 500. Каждая такая программа содержала более 1000 строк кода. Большинство современных программистов сразу же понимают, что ту же задачу можно было бы решить с помощью программы во много раз меньшего размера, используя другую структуру данных — массив из 500 элементов вместо 500 отдельных переменных.

Итак, правильное представление данных способствует написанию структурированных программ. В этой главе описывается множество программ, которые были существенно сокращены и улучшены изменением внутреннего представления данных.

3.1. Программа обработки результатов опроса

Рассмотрим программу анализа 20 000 анкет, заполненных студентами некоторого колледжа. Часть выводимых результатов представлена в виде табл. 3.1.

Для каждой этнической группы сумма количества мужчин и женщин оказывается чуть меньше целого, поскольку некоторые респонденты ответили не на все вопросы. На самом деле выводимый текст был более сложным. Я привел все семь

строк, которые были в оригинале, но некоторые столбцы оригинальной таблицы исключил. На самом деле их было 25, а таблиц таких было три (по одной на каждый из двух студенческих городков и одна общая). Кроме того, нужно было вывести еще несколько похожих таблиц, например, количество студентов, не ответивших на каждый конкретный вопрос. Каждая анкета представлялась в виде записи, в которой поле 0 содержало этническую группу, представленную целым от 0 до 7 (7 возможных категорий + «отказался отвечать»), поле 1 содержало номер студенческого городка (целое между 0 и 2), поле 2 содержало информацию о гражданстве и так далее до поля 8.

Таблица 3.1. Анализ результатов опроса

	Всего	Граждане США	Постоянная виза	Временная виза	М	Ж
Афро-амер.	1289	1239	17	2	684	593
Мекс.-амер.	675	577	80	11	448	219
Амер. индейцы	198	172	5	3	132	64
Испан.	411	223	152	20	224	179
Азиатск.	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

Программист подошел к написанию программы как системный аналитик высокого уровня. Проработав с ней два месяца и выдав на-гора тысячу строк кода, он решил, что работа наполовину сделана. Я понял, чем вызваны его затруднения, взглянув на программу. Она содержала 350 отдельных переменных: 25 столбцов × 7 строк × 2 таблицы. За объявлением переменных следовала запутанная логическая система, в которой определялось, какую из переменных следует увеличивать в соответствии со значением обрабатываемого поля. Подумайте над тем, как бы вы стали писать подобную программу.

Первое предположение заключается в том, что числа следует хранить в массиве. Следующий выбор сделать сложнее: должна ли структура массива соответствовать выводимым данным (три измерения: городок, этническая группа, 25 столбцов) или входным (четыре измерения: городок, этническая группа, категория и значение в этой категории)? Если не учитывать деление на студенческие городки, графически эти варианты можно изобразить следующим образом (рис. 3.1).

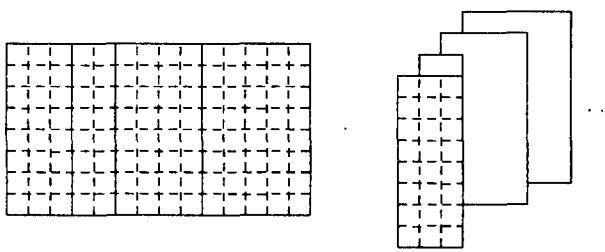


Рис. 3.1. Два варианта представления данных

Оба подхода в данном случае работают. Трехмерный вариант (левая часть рисунка) в моей программе требовал чуть больше работы при считывании данных

и чуть меньше при выводе. Программа состояла всего из 150 строк кода: 80 на построение описанных таблиц, 30 на формирование выводимого текста и 40 для обработки прочих таблиц.

Описанные выше программы были неоправданно велики; в них использовались отдельные переменные, которые следовало представить в виде массива. Уменьшение объема кода на порядок привело к упрощению написания программ: количество ошибок и время написания и отладки кода уменьшилось. Обе программы стали более эффективными и менее требовательными к памяти, хотя в данном случае это и было некритично.

Почему программисты пишут большие программы, когда можно обойтись и маленькими? Одна из причин — недостаток лени, о которой говорится в разделе 2.5 (глава 2); они сразу бросаются воплощать в жизнь первую же пришедшую в голову идею. Но в обоих описанных случаях существовала и еще одна проблема: программисты мыслили на языках, в которых массивы обычно используются как таблицы фиксированных значений, инициализируемые в начале и не меняющиеся в процессе выполнения программы. Джеймс Адамс, автор книги, упоминаемой в первой главе, сказал бы, что у программистов был концептуальный блок, не позволявший им использовать динамические массивы счетчиков.

Есть много других причин, по которым программисты совершают подобные ошибки. Когда я готовился к написанию этой главы, то нашел аналогичный пример в своей собственной программе анализа результатов опроса. В основном цикле содержалось сорок строк кода в восьми блоках из пяти операторов, первые два из которых имели вид:

```
ethnicgroup = entry[0]
campus = entry[1]
if entry[2] == refused /* отказался отвечать */
    declined[ethnicgroup, 2]++
else
    j = 1 + entry[2]
    count[campus, ethnicgroup, j]++
if entry[3] == refused
    declined[ethnicgroup, 3]++
else
    j = 4 + entry[3]
    count[campus, ethnicgroup  j]++
```

Я мог бы заменить эти сорок строк шестью, инициализировав массив offset (сдвиг) значениями 0, 0, 1, 4, 6, ...

```
for i = [2, 8]
    if entry[i] == refused
        declined[ethnicgroup, i]++
    else
        j = offset[i] + entry[i]
        count[campus, ethnicgroup, j]++
```

Я был так рад, сократив объем кода в 10 раз, что пропустил еще одну возможность сократить его, которая на самом деле бросалась в глаза.

3.2. Обработка шаблонных писем

Представьте, что вы только что ввели свое имя и пароль для входа на web-сайт своего любимого магазина. На следующей странице, которую вы увидите, может быть, например, такой текст:

```
Добро пожаловать, Джейн!
Мы надеемся, что никто из членов семьи Паблик не забывает напоминать
своим соседям по улице Мэпл Стрит покупать товары у нас!
Как всегда, мы направим Ваш заказ по адресу
    Мисс Джейн К. Паблик
    600 Мэпл Стрит
    Касл Рок, штат Айова 12345
```

Как программист, вы должны понимать, что за всем этим стоит компьютер, который использует ваше имя (имя пользователя) для обращения к базе данных и получает оттуда сведения в форме:

```
Паблик|Джейн|К|Мисс|600|Мэпл Стрит|Касл Рок|Айова|12345
```

Но каким же именно образом программа формирует страницу специально для вас? Не очень ленивый программист мог бы написать программу таким образом:

```
read lastname, firstname, init, title, streetnum, streetname, town,
state, zip
print "Добро пожаловать, ", firstname, "!"
print "Мы надеемся, что никто из "
print "членов семьи ", lastname," не забывает"
print "напоминать своим соседям "
print "по улице ", streetname, " покупать товары у нас."
print "Как всегда, мы направим Ваш заказ по адресу"
print "     ", title, firstname, init, "  ", lastname
print "     ", streetnum, streetname
print "     ", town, " ,", state, zip
```

Такая программа напрашивается сама собой, но ее написание несколько утомляет.

Более элегантный подход состоит в использовании генератора писем по шаблонам, который обрабатывает шаблоны наподобие следующего:

```
Добро пожаловать, $1!
Мы надеемся, что никто из членов семьи $0 не забывает напоминать своим
соседям по улице $5 покупать товары у нас!
Как всегда, мы направим Ваш заказ по адресу
    $3 $1 $2  $0
    $4 $5
    $6, $7 $8
```

Запись $i обозначает i-е поле записи. При этом $0 соответствует фамилии покупателя и т. д. Форма обрабатывается приведенным ниже псевдокодом, который предполагает, что для вывода символа $ нужно ввести: $$.

```
read fields from database
/* считать значения полей из базы данных */
loop from start to end of schema
/* цикл от начала до конца шаблона */
    c = next character of schema /* следующий символ шаблона */
    if c != '$'
        printchar c /* вывод символа c */
    else
        c = next character of schema /* следующий символ шаблона */
        case c of
            '$'  printchar '$'
            '0' - '9'  printstring field[c]
            default: error("bad schema") /* ошибка  некорректный шаблон
*/
```

Таким образом, шаблон рассматривается как длинный массив символов, в котором строки текста завершаются символами перевода строки. Perl и другие языки сценариев сильно упрощают это, поскольку в них можно использовать переменные наподобие $lastname.

Написание генератора и шаблона, скорее всего, окажется проще, чем написание первого пришедшего в голову варианта программы. Разделение данных и процедур их обработки впоследствии окупится: для изменения текста письма можно исправить его шаблон в текстовом редакторе, что существенно проще изменения текста программы.

Использование концепции шаблона отчета могло бы сильно упростить программу из 5300 строк на языке Cobol, с которой мне однажды пришлось иметь дело. На вход программы поступала информация о материальном статусе семьи, а на выходе должен был получаться буклет, в котором эта информация подытоживалась бы, а семье давались бы некоторые рекомендации. Вот данные о задаче: 120 входных полей, 400 выводимых строк на 18 страницах, 300 строк кода для выделения входных данных, 800 для вычислений и 4200 для вывода. Я оценил, что 4200 строк, требовавшихся для вывода данных, можно было бы заменить интерпретатором шаблона из нескольких десятков строк и собственно шаблоном из 400 строк; при этом необходимости изменять подпрограммы вычислений не было. Если бы программа была сразу написана в этой форме, она бы сократилась примерно в три раза и ее было бы гораздо легче корректировать.

3.3. Примеры

Меню

Я хотел, чтобы пользователь, работавший с моей программой на языке Visual Basic, мог выбрать один из нескольких элементов щелчком мыши на одном из пунктов меню. Я просмотрел отличную коллекцию программ-примеров и нашел тот, в котором пользователь мог выбирать один из восьми пунктов меню. Когда я посмотрел, какой код обеспечивал выбор одного из элементов меню, я ужаснулся:

```
sub menuitem0_click()
    menuitem0.checked = 1
    menuitem1.checked = 0
    menuitem2.checked = 0
    menuitem3.checked = 0
    menuitem4.checked = 0
    menuitem5.checked = 0
    menuitem6.checked = 0
    menuitem7.checked = 0
```

Для второго элемента было написано то же самое, с небольшими изменениями:

```
sub menuitem1_click()
    menuitem0.checked = 0
    menuitem1.checked = 1
```

И так далее для элементов 3–8. А всего на выбор одного из пунктов меню ушло около 100 строк кода.

Я и сам однажды написал подобного монстра, начав с двух пунктов меню, для которых подобная структура выглядела достаточно удобной. Потом я добавил тре-

ний, четвертый и последующие элементы. Я был слишком увлечен тем, что делала моя программа, чтобы остановиться и исправить свою ошибку.

Если подумать немного, можно написать функцию, которая будет обнулять все поля. Первая функция при этом примет вид:

```
sub menuitem0_click()
    uncheckall /* сброс всех полей в 0 */
    menuitem0 checked = 1
```

Но нам все равно придется написать семь аналогичных функций.

К счастью, Visual Basic поддерживает использование массива пунктов меню. Поэтому восемь похожих друг на друга функций можно заменить одной:

```
sub menuitem_click(int choice)
    for i = [0, numchoices)
        menuitem[i] checked = 0
    menuitem[choice] checked = 1
```

Объединение повторяющихся команд в одну функцию сокращает 100 строк кода до 25, а использование массивов сокращает количество строк до четырех. При этом добавлять новые пункты становится проще, а код становится абсолютно ясным и понятным — в нем сложно наделать ошибок. Такой подход дал мне возможность решить задачу, набрав несколько строк кода.

Сообщения об ошибках

В плохих системах можно найти сотни команд вывода сообщений об ошибках, разбросанных по всему коду, вперемежку с другими операторами. В хороших системах вывод всех сообщений об ошибках осуществляется с помощью одной функции. Сравните сложность следующих задач в «хорошей» и «плохой» системах:

- вывод списка всех возможных сообщений об ошибках;
- добавление звукового оповещения к сообщениям о самых опасных ошибках;
- перевод сообщений об ошибках на французский или другие языки.

Функции для работы с датами

Дан год и день с начала года. Нужно определить соответствующий месяц и число. Например, 61-й день в 2004 году будет первым днем третьего месяца. В книге «Элементы стиля программирования» (Kernighan, Plauger, Elements of Programming Style) Керниган и Плоджер привели пример подпрограммы из пятидесяти строк, решавшей эту задачу «в лоб». Подпрограмма эта была взята из чьей-то реальной программы. Затем они привели пример подпрограммы из пяти строк, решавшей ту же задачу. В ней использовался массив из 26 целых чисел. В задаче 4 данной главы обсуждаются вопросы, связанные с функциями для обработки дат.

Анализ слов

При анализе слов английского языка возникает множество вычислительных проблем. В разделе 13.8 (см. главу 13) мы разберем, каким образом программы проверки правописания используют процедуру удаления суффиксов для сокращения словарей: словарь при этом содержит единственное слово «laugh» без всех возможных производных (-ing, -s, -ed и другие). Лингвисты разработали целый набор

правил для выполнения этой процедуры. Дуг Макилрой знал, что эти правила нельзя непосредственно воплощать в виде кода, поэтому в 1973 году, когда он разрабатывал свой первый синтезатор речи, работавший в режиме реального времени, он использовал в нем таблицу из 400 строк и 1000 строк кода. Когда кто-то стал изменять его программу, не используя при этом таблиц, ему пришлось написать 2500 лишних строк кода, что увеличило функциональность программы всего лишь на 20%. Макилрой утверждает, что он мог бы уложиться меньше чем в 1000 добавочных строк, используя таблицы. Попробуйте себя в реализации подобного набора правил в задаче 5 данной главы.

3.4. Структурирование данных

Что же такое «хорошо структурированные данные»? Когда-то давно хорошая структура означала правильный выбор имен переменных. Там, где раньше программисты пользовались параллельными массивами и сдвигами, поздние языки ввели понятия записей, структур и указателей на них. Мы научились заменять код для работы с данными на функции с именами типа «вставка» и «поиск» (insert, search), это помогало менять представление данных отдельно от кода программы. Дэвид Парнас (David Parnas) расширил этот подход: обрабатываемые данные подсказывают следующий шаг на пути к модульной структуре программы.

Затем появилось «объектно-ориентированное программирование». Программисты научились выделять основные объекты в своих программах, в мире появились абстракции и основные операции для них; стало возможным и полезным скрытие деталей реализации. Языки наподобие Smalltalk и C++ позволяют инкапсулировать объекты в классы; в главе 13 мы изучим этот подход во всех подробностях при рассмотрении абстракции наборов и ее реализации.

3.5. Обработка специальных данных

В прежние времена программисты писали все свои программы «с нуля». Современные средства разработки сводят труд программиста к минимуму. Список имеющихся средств, приведенный ниже в этой главе, далеко не полон. Каждое из средств использует один из возможных видов данных для решения конкретной, но достаточно общей проблемы. Языки вроде Visual Basic и Tcl, а также многие интерпретаторы команд позволяют связывать подобные объекты в блоки.

Гипертекст

В начале 90-х, когда число web-сайтов не превышало десятка тысяч, я был ошеломлен количеством сведений, собранных в различных справочниках, выходивших в то время на компакт-дисках. Можно было купить энциклопедии, словари, альманахи, телефонные справочники, классическую литературу, учебники, документацию по различным системам и многое другое. Однако интерфейсы, предоставляемые производителями этих сборников, также ошеломляли: у каждой программы были свои, подчас очень специфические, особенности. Сейчас у меня тоже есть доступ ко всем этим данным на компакт-дисках и в сети Интернет, но интерфейс к ним — это браузер, который я могу выбирать сам. Это упрощает жизнь пользователю и производителю.

Пары «имя-значение»

Библиография может содержать различные вхождения, например такие:

```
%title    The C++ Programming Language, Third Edition
%author    Bjarne Stroustrup
%publisher    Addison-Wesley
%city    Reading, Massachusetts
%year    1997
```

В Visual Basic аналогичный подход используется для описания интерфейса. Текстовое поле в левом верхнем углу формы обладает некоторыми свойствами (атрибутами), которые имеют имена и значения, приведенные в табл. 3.2.

Таблица 3.2. Описание свойств объекта

Имя	Значение
Height	495
Left	0
Multiline	False
Name	txtSample
Top	0
Visible	True
Width	215

Всего этих свойств около 30. Например, чтобы расширить текстовое поле, я могу перетащить его правый край мышью, либо ввести другое значение в соответствующее поле, либо просто выполнить оператор присваивания во время выполнения программы:

```
txtSample Width = 400
```

Программист может выбрать подходящий способ работы с этой простой структурой данных.

Таблицы (spreadsheets)

Вести бюджет организации неспециалисту, скажем мне, достаточно сложно. За отсутствием соответствующего опыта, я бы написал большую программу с неудобным пользовательским интерфейсом. Другой программист взглянул бы на задачу шире и реализовал бы подобное приложение как электронную таблицу с помощью небольшого количества функций на Visual Basic. Интерфейс ее оказался бы удобным и естественным для тех, кому нужно было ей пользоваться. Если бы мне сейчас пришлось писать программу анализа результатов опроса, необходимость работать с массивами чисел заставила бы меня попробовать реализовать ее в виде электронной таблицы.

Базы данных

Много лет назад, записав подробности своих первых парашютных прыжков в бумажный журнал, один знакомый программист решил автоматизировать ведение записей о своих рекордах в парашютном спорте. Если бы это произошло чуть раньше, ему понадобилось бы самостоятельно разработать структуру записи и само-

стоятельно написать программы для занесения, обновления и получения данных. Но сейчас он оценил по достоинству возможности современного коммерческого пакета для работы с базами данных — для определения новых записей и операций с ними ему пришлось потратить несколько минут, а не несколько дней.

Специальные языки

Графический интерфейс пользователя благополучно вытеснил многие устаревшие и неуклюжие текстовые языки программирования. Однако в некоторых областях специальные языки все еще находят применение. Я, например, не люблю тыкать мышью в кнопки на виртуальном калькуляторе, когда мне нужно вычислить что-нибудь достаточно простое:

```
n = 1000000
47 * n * log(n)/log(2)
```

Или, скажем, вместо того чтобы составлять запрос с помощью вычурных комбинаций текстовых полей и кнопок, я предпочитаю программировать на языке, в котором возможны конструкции вида:

```
(design and architecture) and not building
(дизайн И архитектура) И НЕ строительство
```

Окна, которые раньше описывались с помощью сотен строк исполняемого кода, теперь определяются всего лишь несколькими десятками строк при помощи HTML. Языки, может быть, и «вышли из моды» для большинства пользователей, но они все еще весьма полезны в некоторых приложениях.

3.6. Принципы

Истории, которые здесь приводились в качестве примеров, охватывают промежуток в несколько десятилетий, и рассказывается в них о десятке различных языков. Мораль одна: не пишите большую программу, когда можно написать маленькую. Бóльшая часть примеров иллюстрирует то, что Полья в книге «Как это решить» (Polya, How to Solve It) называет *парадоксом изобретателя*: «более общую проблему обычно решить проще». В программировании это означает, что может быть непросто решить проблему с выбором из 23 возможных ситуаций, но если обобщить ее на n ситуаций, решить, а затем положить n = 23, то это может существенно упростить дело.

В этой главе мы сосредоточились на одном из возможных эффектов применения соответствующих структур данных — уменьшении объемов программ. Правильная разработка структуры данных может давать и другой эффект, включая уменьшение требований к ресурсам и улучшение переносимости и модифицируемости кода. Приведенный ниже комментарий Ф. Брукса в главе 9 его книги «Мифический человеко-месяц» (Fred Brooks, Mythical Man Month) относится к уменьшению требований к памяти, но его может взять на вооружение любой программист, стремящийся к улучшению любых параметров своей программы: «Когда у программиста иссякнут все возможности по уменьшению занимаемой памяти, чаще всего ему лучше будет забыть о своей программе, вернуться к самому началу и заново продумать структуру данных. Представление данных — это действительно *суть* программирования».

Вот еще несколько принципов, которыми вы сможете воспользоваться, когда отключитесь от своего кода и задумаетесь о представлении данных.

Повторяющиеся действия выполняйте над массивами

Длинная последовательность однотипных команд лучше всего выражается с помощью простейшей структуры данных — массива.

Инкапсулируйте сложные структуры

Когда вам нужно работать со сложной структурой данных, определяйте ее в абстрактных терминах и выражайте операции с ней как операции с некоторым классом.

Используйте самые совершенные средства везде, где это возможно

Гипертекст, пары «имя-значение», электронные таблицы, базы данных, специальные языки и подобные вещи — все эти средства являются весьма эффективными в своих областях.

Данные должны определять структуру программы

В этой главе я пытался показать, каким образом данные могут определять структуру программы, позволяя заменить сложный код на простой, работающий с правильно выбранной структурой. Хотя детали могут меняться, суть дела остается неизменной: хорошие программисты тщательно вникают в организацию входных, выходных и промежуточных данных, прежде чем начать писать программу.

3.7. Задачи

1. На момент выхода второго издания этой книги в печать индивидуальный доход в Соединенных Штатах облагается налогами следующим образом. Имеется двадцать пять различных ставок, максимальная из которых равна приблизительно 40%. Раньше правила были еще сложнее. Учебник по программированию предлагал использование 25 операторов if в качестве разумного подхода к вычислению подоходного налога в 1978 году. Последовательность ставок выглядит следующим образом: 0,14; 0,15; 0,16; 0,17; 0,18 и так далее, причем для последующих элементов разность соседних становится больше 0,1. Есть комментарии?

```
if income <= 2200
    tax = 0
else if income <= 2700
    tax =  14 * (income - 2200)
else if income <= 3200
    tax = 70 +  15 * (income - 3200)
else if income <= 3700
    tax = 145 + 0 16 * (income - 3200)
```

```
else if income <= 4200
    tax = 225 +  17 * (income - 3700)
    ...
else
    tax = 53090 + 0 70 * (income - 102200)
```

2. Имеется рекуррентное определение последовательности k-го порядка:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k} + c_k ,$$

где $c_1,\ldots,c_k$ — вещественные коэффициенты. Напишите программу, считывающую значения $k, a_1, \ldots a_k, \ldots, c_1, \ldots, c_{k+1}$ и m и выводящую значения $a_1,\ldots,a_m$. Насколько она сложна по сравнению с программой, вычисляющей одно конкретное рекуррентное соотношение пятого порядка без использования массивов?

3. Напишите функцию для изготовления заголовков (banner), на вход которой подается одна буква, а на выходе получается массив символов, изображающих эту букву.

4. Напишите функции для следующих задач обработки дат:

   - вычисление промежутка между двумя датами в днях;
   - определение дня недели по дате;
   - изготовление календаря на заданный месяц определенного года в виде массива символов.

5. Эта задача представляет собой подмножество проблемы расстановки переносов в английских словах. Ниже приведен список правил для корректной расстановки переносов в словах, заканчивающихся на букву «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 -clic l-ic h-ic f-ic d-ic -bic a-ic -mac i-ac

   Правила должны применяться в приведенном порядке, поэтому правильными будут, например, варианты eth-nic (соответствующее правилу h-nic) и clin-ic (попадающее под вариант n-ic). Как бы вы представили эти правила в функции, которая должна возвращать положение переноса для суффикса передаваемого ей слова?

6. Напишите генератор писем по образцу, способный формировать отдельный документ для каждой записи из базы данных. Создайте несколько небольших схем для него и несколько входных файлов с данными, чтобы проверить правильность его работы.

7. Словари обычно предназначены для того, чтобы узнавать значение слова (его определение), а в задаче 1 в главе 2 рассматривался пример словаря анаграмм. Разработайте программу для орфографического словаря и для словаря рифм. Подумайте над словарями, которые могли бы содержать целочисленные последовательности (например, 1, 1, 2, 3, 5, 8, 13, 21, ...), химические соединения или стихотворные размеры.

8. [S. C. Johnson] Для отображения любой из десяти цифр можно использовать семиэлементный дисплей. Цифры при этом будут выглядеть так (рис. 3.2).

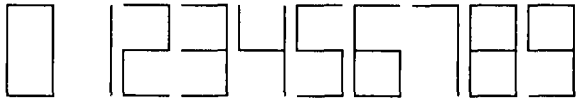


Рис. 3.2. Изображения цифр на семисегментном индикаторе

Сегменты при этом обычно нумеруются следующим образом (рис. 3.3).

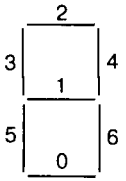


Рис. 3.3. Нумерация сегментов семисегментного индикатора

Напишите программу, позволяющую управлять отображением 16-битного целого с помощью пяти таких дисплеев, то есть пяти цифр. На выходе должен получаться массив из 5 байт, причем i-й бит j-го байта должен быть установлен в 1 тогда и только тогда, когда «горит» сегмент i цифры j.

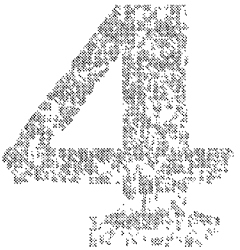
3.8. Дополнительная литература

Данные могут определять структуру программ, но только умные программисты могут структурировать большие системы программного обеспечения. Подзаголовок книги Стива Макконнелла (Steve McConnell, Code Complete, Microsoft Press, 1993) точно отражает содержание этого тома из 860 страниц: «*Практический справочник по конструированию программного обеспечения*». Справочник этот весьма способствует быстрому обретению мудрости.

Главы этой книги с восьмой по двенадцатую посвящены данным и весьма близки по содержанию тексту данной главы. Макконнелл начинает с простых вещей, таких как объявление переменных и правильный выбор имен, а затем обращается к более сложным проблемам, список которых включает написание программы обработки таблиц и использование абстрактных типов данных. Главы с четвертой по седьмую посвящены разработке программ; они также представляют собой углубленное рассмотрение вопросов, которые обсуждались в данной главе.

Для создания больших программных комплексов необходимо обладать достаточно широким спектром знаний, начиная с оптимизации простых функций и заканчивая руководством проектами. Упомянутая книга Макконнелла вместе с другими его книгами: «Быстрая разработка» (Steve McConnell, Rapid Development, Microsoft Press, 1996) и «Руководство по выживанию в программных проектах» (Steve McConnell, Software Project Survival Guide, Microsoft Press, 1998) перекрывает весь диапазон. Его книги читаются легко, но никогда не следует забывать, что содержащиеся в них знания добыты тяжелым трудом.

Как писать правильные программы



В конце 60-х годов много говорилось о возможности автоматической верификации программ. К сожалению, за последующие десятилетия никаких систем автоматической проверки так и не появилось. Несмотря на отсутствие результата, исследования, которые были проведены в области верификации программ, дали нам гораздо больше, чем черный ящик, переваривающий программу и выдающий вердикт «хорошая» или «плохая», — люди достигли понимания глубинных основ программирования.

Цель этой главы — показать, каким образом понимание этих основ может помочь в написании программ. Один из читателей охарактеризовал подход большинства программистов к этому вопросу следующим образом: «Напишите программу, перебросьте ее текст через стенку в соседний кабинет, и пусть отделы гарантии качества и тестирования разбираются с ошибками». В этой главе мы опишем альтернативный подход. Прежде чем обратиться к самой теме, мы рассмотрим ее со всех сторон. Умение правильно кодировать — лишь одно из множества требований для написания правильных программ. Самое сложное — то, о чем мы говорили в предыдущих трех главах, — правильно поставить задачу, разработать алгоритм и выбрать подходящую структуру данных. Если вы справитесь с этим, кодирование, скорее всего, не вызовет проблем.

4.1. Двоичный поиск бросает вызов

Даже если на этапе разработки все было проделано наилучшим образом, программиста зачастую ожидает написание весьма сложного кода. Эта глава посвящена одной задаче, для которой требуется особенно аккуратное кодирование, — двоичному поиску. Мы уже рассмотрели ранее эту проблему и сделали набросок алгоритма. Теперь мы воспользуемся основными принципами верификации, чтобы правильно написать программу.

Впервые мы столкнулись с двоичным поиском в разделе 2.2 (см. главу 2), где нужно было определить, содержит ли отсортированный массив x [0..n-1] элемент t[1]. Мы точно знали, что n неотрицательно и что каждый последующий элемент массива не меньше предыдущего. Кроме того, мы знали, что если n=0, то массив пуст. Тип элементов t и x совпадает; псевдокод работает одинаково хорошо с целыми и вещественными числами и со строками. Ответ помещается в целое число p (от слова позиция): если элемент в массиве отсутствует, в p помещается число -1. В противном случае p неотрицательно и не превышает n-1, и t==x[p].

Алгоритм двоичного поиска решает задачу, уменьшая размер поддиапазона, в котором может находиться t (если это число вообще есть в массиве). В начале диапазоном является весь массив. Затем его средний элемент сравнивается с t, после чего половина диапазона исключается из рассмотрения. Процесс продолжается до тех пор, пока число t не будет найдено или пока диапазон, в котором оно должно находиться, не окажется пуст. Для массива из n элементов двоичный поиск выполняет около $\log_2 n$ сравнений.

Большинство программистов полагают, что по подобному описанию написать программу не составляет труда. Они ошибаются. Единственный способ заставить вас поверить в это: отложите книгу и попробуйте сами написать программу. Попробуйте!

Я предлагал проделать подобную работу слушателям курсов для профессиональных программистов. Им давалось несколько часов на то, чтобы сделать программу по данному им описанию на произвольном, выбранном ими языке, кроме того, разрешалось воспользоваться псевдокодом высокого уровня. К концу предложенного времени почти все отвечали, что они готовы. На проверку их кода с помощью тестов отводилось еще тридцать минут. Статистика по нескольким классам, в которых обучалось более ста человек, получилась такая: 90% программистов нашли ошибки в своих программах (и я не всегда был уверен в правильности текстов тех, у кого явных ошибок обнаружено не было).

Я был удивлен. Имея в запасе практически неограниченное время, только 10% профессиональных программистов смогли написать эту небольшую программу правильно. Эта задача оказалась не по зубам не только моим ученикам: раздел 6.2.1 третьего тома книги Кнута «Искусство программирования для ЭВМ: сортировка и поиск» посвящен истории этого алгоритма, и в нем Кнут отмечает, что, хотя первая программа двоичного поиска была опубликована в 1946 году, программа, не содержащая ошибок, появилась лишь в 1962 году.

4.2. Пишем программу

Основная идея двоичного поиска состоит в том, что мы всегда знаем, что если t вообще есть в массиве, то оно должно быть в выбранном нами поддиапазоне. Мы будем использовать выражение mustbe (range) для отражения того факта, что если t есть в массиве, то оно есть и в исследуемом на данном этапе диапазоне range. Мы

[1] Обратитесь к задаче 1 в главе 5 и ее решению, если вам нужна помощь в наведении критики на короткие имена переменных, формат определения функции двоичного поиска, обработку ошибок и другие проявления стиля программирования, жизненно важные для больших программных проектов.

воспользуемся этой записью, чтобы превратить описание двоичного поиска в набросок программы (листинг 4.1):

Листинг 4.1. Набросок программы двоичного поиска

```
initialize range to 0 .n-1
/* инициализируем диапазон  0  n-1 */
loop
/* цикл */
    { invariant  mustbe(range) }
/* инвариант  число t должно быть в диапазоне range */
    if range is empty, /* если диапазон пуст */
        break and report that t is not in array
       /* цикл завершается, t в массиве нет */
    compute m, the middle of the range
    /* находим m - середину диапазона */
    use m as a probe to shrink the range
    /* используем значение m для сужения диапазона */
        if t is found during the shrinking process,
        break and report its position
       /* если при этом находим t, */
       /* завершаем цикл и возвращаем позицию t */
```

Важной частью этой программы является *инвариант цикла* (loop invariant), заключенный в фигурные скобки. Это утверждение, относящееся к состоянию программы, называется *инвариантом*, поскольку оно остается истинным в начале и в конце каждой из итераций; это формализация того интуитивного утверждения, которое мы сделали выше.

Теперь займемся усовершенствованием нашей программы, заботясь о сохранности инварианта в ходе всех наших действий. Прежде всего, нам нужно как-то представить диапазон. Для этого мы будем использовать два индекса l и u, задающие нижнюю и верхнюю границы диапазона l..u. Функция двоичного поиска в разделе 9.3 главы 9 представляет диапазон другим способом: заданием его начала и длины. Логическая функция mustbe(l, u) утверждает, что если число t вообще есть в массиве, то оно обязательно принадлежит диапазону x[l..u], включая границы диапазона.

Затем нужно заняться инициализацией. Какими должны быть значения l и u, чтобы утверждение mustbe(l, u) было истинным? Очевидный выбор: 0 и n-1: утверждение mustbe(0, n-1) сводится к тому, что если t есть в массиве x, то оно принадлежит x[0..n-1], а именно это и известно нам с самого начала. Следовательно, инициализация будет состоять в присваивании значений l=0 и u=n-1.

Затем нужно проверить диапазон на наличие в нем вообще каких-либо элементов, после чего найти его середину m. Диапазон l..u является пустым, если l>u, и в этом случае мы помещаем в переменную p специальное значение -1, завершая цикл.

```
if (l>u)
    p=-1, break
```

Оператор break завершает внутренний цикл, к которому он относится. Следующее выражение вычисляет значение m (середину диапазона):

```
m = (l+u)/2
```

Здесь используется целочисленное деление: 6/2=3, и 7/2=3. Итак, наша усовершенствованная программа выглядит теперь следующим образом (листинг 4.2):

Листинг 4.2. Вторая версия программы двоичного поиска

```
l = 0, u = n-1
loop
    {invariant, mustbe(l, u) }
    if l>u
        p= -1, break
    m = (l + u) /2
        use m as a probe to shrink the range l  u
        if t is found during the shrinking process,
        break and note its position.
        /* русский перевод см  в предыдущем листинге */
```

Теперь нужно в явном виде записать последние три строки. Для этого необходимо сравнить t и x[m], после чего выполнить определенное действие для сохранения инварианта. Действие это будет выполняться по следующей схеме:

```
case
    x[m] < t  action a
    x[m] == t  action b
    x[m] > t  action c
```

Действие b, очевидно, будет состоять в присваивании p=m и выходе из цикла, поскольку выполнение условия означает, что мы нашли элемент, равный t. Оставшиеся два случая симметричны, мы займемся первым, а второй получим из него по соображениям симметрии (это одна из причин, по которым мы будем очень аккуратны при проверке кода в следующем разделе).

Если x[m]<t, мы понимаем, что x[0]<=x[1]<=...<=x[m]<t, поэтому t не может принадлежать диапазону x[0..m]. Поскольку мы знаем также, что t не может *не* принадлежать x[l..u], мы понимаем, что если t где-нибудь есть, то оно принадлежит диапазону x[m+1..u], что мы запишем в форме mustbe(m+1, u). Поэтому мы восстановим инвариант mustbe(l, u) путем присваивания l = m+1. Уточнив, таким образом, предыдущий набросок, получим листинг 4.3.

Листинг 4.3. Двоичный поиск на псевдокоде: версия 3

```
l = 0, u = n - 1
loop
    { mustbe(l, u) }
    if l > u
        p = -1, break
    m = (l + u) / 2
    case
        x[m] < t, l = m + 1
        x[m] == t  p = m, break
        x[m] > t, u = m - 1
```

Программа получилась короткой: десяток строк кода и один инвариант. Простейший способ проверки программы — запись инварианта и сохранение его неизменным в процессе написания программы — оказался очень полезным при переписывании наброска алгоритма на псевдокоде. Этот процесс дал нам некоторую уверенность в программе, но мы ни в коем случае не можем считать ее правильной. Прежде чем читать дальше, потратьте несколько минут, чтобы проверить, правильно ли она работает.

4.3. Понимание программы

Когда мне приходится сталкиваться со сложной проблемой, требующей аккуратного программирования, я пытаюсь записать код с уровнем детализации, который

только что был продемонстрирован. Затем я использую методы верификации для повышения собственной уверенности в правильности программы. Верификация на этом уровне рассматривается в главах 9, 11 и 14.

В этом разделе мы изучим процесс верификации программы двоичного поиска в его наиболее подробном виде. На практике я анализирую программы гораздо менее формально. Текст программы, приведенный в листинге 4.4, наполнен утверждениями, формализующими интуитивные соображения, использовавшиеся при написании кода.

Хотя код разрабатывался сверху вниз (вначале — общая идея, затем — детализация), анализ корректности будет производиться снизу вверх: мы будем рассматривать строки по отдельности, а затем посмотрим, как они работают вместе при решении задачи.

ВНИМАНИЕ

Дальше будет скучно. Переходите сразу к разделу 4.4, если почувствуете сонливость.

Начнем с первых трех строк. Утверждение в строке 1: `mustbe(0, n-1)` является истинным по определению утверждения `mustbe` — если `t` вообще имеется в массиве, то оно должно быть в диапазоне `x[0..n-1]`. Операции присваивания в строке 2 (`l = 0` и `u = n - 1`) делают истинным утверждение в строке 3: `mustbe(l, u)`.

Теперь мы переходим к сложной части: циклу в строках 4–27. Аргументировать его правильность будем тремя утверждениями, каждое из которых связано с инвариантом цикла.

- **Инициализация.** Инвариант является и остается истинным при первом проходе цикла.
- **Сохранность.** Если инвариант является истинным в начале цикла, то он останется истинным и после окончания выполнения тела цикла.
- **Завершение.** Выполнение цикла будет окончено, а желаемый результат будет сохранен (в данном случае результат состоит в том, что переменной `p` присваивается правильное значение). Чтобы показать это, нам придется воспользоваться следствиями из истинности инварианта.

Для доказательства правильности инициализации отметим, что утверждение в строке 3 совпадает с утверждением в строке 5. Для доказательства правильности сохранности и завершения мы будем обращаться к строкам 5–21. Когда мы начнем обсуждать строки 9 и 21 (операторы `break`), мы установим свойства этапа завершения, и если мы доберемся до строки 27, то гарантируем сохранность инварианта, поскольку утверждение в этой строке совпадает с утверждением в строке 5.

Листинг 4.4. Анализ корректности программы двоичного поиска

```
1  { mustbe(0, n-1) }
2  l = 0, u = n-1
3  { mustbe(l, u) }
4  loop
5     { mustbe(l, u) }
6     if l > u
7        { l > u && mustbe(l, u) }
8        { t is not in the array }
9        p = -1, break
10    { mustbe(l, u) && l <= u }
11    m = (l + u) / 2
```

```
12.     { mustbe(l, u) && l <= m <= u }
13      case
14.         x[m] < t
15.             { mustbe(l, u) && cantbe(0, m) }
16              { mustbe(m+1, u) }
17              l = m + 1
18.             { mustbe(l, u) }
19          x[m]== t
20              { x[m] == t }
21.             p = m; break
22.         x[m] > t.
23              { mustbe(l, u) && cantbe(m, n) }
24              { mustbe(l, m-1) }
25.             u = m-1
26.             { mustbe(l, u) }
27      { mustbe(l, u) }
```

Успешная проверка в строке 6 делает возможным утверждение в строке 7: если t есть в массиве, оно должно быть между элементами l и u и при этом l>u. Из этого следует строка 8: t в массиве нет. Поэтому мы корректно завершаем цикл в строке 9, предварительно выполнив присваивание p = -1.

Если утверждение в строке 6 ложно, мы переходим на строку 10. Инвариант все еще остается истинным (мы не делали ничего, что могло бы это изменить). Поскольку условие в строке 6 оказалось ложным, мы уверены, что l<=u. Строка 11 устанавливает m равным среднему этих чисел, округленному к ближайшему меньшему целому. Поскольку среднее всегда лежит между двумя величинами и округление не может сделать его меньшим l, мы можем сделать утверждение в строке 12.

Теперь проанализируем все три ситуации оператора case в строках 13–27. Проще всего разобраться со второй альтернативой в строке 19. Вследствие утверждения из строки 20 мы имеем право присвоить переменной p значение m и выйти из цикла. Это второе из двух мест, где возможен выход из цикла, так что мы доказали корректность завершения процедуры.

Теперь рассмотрим одну из двух симметричных ветвей оператора case. Поскольку при написании кода мы работали с первой ветвью, теперь мы займемся второй (строки 22–26). Возьмем утверждение в строке 23. Первая часть его — инвариант, который нигде ранее в цикле не был изменен. Вторая часть истинна, поскольку t<x[m]<=x[m+1]<=...<=x[n-1], так что мы можем быть уверены, что t не может быть равным одному из элементов с индексом, большим m-1; это записывается коротко, как cantbe(m, n-1). Логика говорит нам, что если t должно быть между l и u и оно не может быть «выше» m или находиться точно в этой позиции, следовательно, оно должно быть между l и m-1 (если оно вообще где-то есть); отсюда следует и строка 24. Выполнение строки 25 при условии истинности строки 24 сохраняет истинность строки 26 — просто по определению операции присваивания. Таким образом, эта ветвь оператора case заново устанавливает инвариант в строке 27.

Доказательство корректности строк 14–18 выглядит точно так же, поэтому теперь мы можем считать, что мы доказали правильность всех трех ветвей оператора case. Одна из ветвей корректно завершает цикл, а другие две сохраняют истинность инварианта.

Этот анализ показывает, что если цикл завершается, то значение p оказывается истинным. Однако мы все еще можем попасть в ситуацию с бесконечным циклом.

На самом деле именно таким был результат ошибок, допущенных большинством профессиональных программистов.

Доказательство конечности использует другой аспект диапазона l..u. Этот диапазон изначально имеет некоторый конечный размер (n), и строки с 6-й по 9-ю гарантируют, что цикл будет завершен, когда в диапазоне окажется менее одного элемента. Следовательно, нужно показать, что диапазон уменьшается при каждой итерации. Строка 12 говорит, что m всегда принадлежит текущему диапазону. Обе ветви оператора case, в которых выполнение цикла не прерывается (строки 14 и 22), исключают значение m из текущего диапазона и, следовательно, уменьшают его по меньшей мере на 1 элемент. Таким образом, программа обязательно должна завершиться.

Теперь я практически уверен, что мы можем пойти дальше и закодировать функцию на C. В следующей главе мы займемся именно этим — реализацией функции на языке C и проверкой ее корректности и эффективности.

4.4. Принципы

Это упражнение продемонстрировало множество преимуществ верификации программ: рассмотренная задача достаточно важна и требует аккуратного кодирования; при разработке программы мы пользуемся принципами верификации, а при анализе корректности используются достаточно общие методы. Главным недостатком этого примера является уровень детализации: на практике я работаю на гораздо менее формальном уровне. К счастью, все эти детали иллюстрируют набор общих принципов.

Утверждения

Соотношения между входными данным, переменными программы и результатами описывают состояние программы. Утверждения дают возможность программисту явно записать эти соотношения. Их роль при написании программы обсуждается в следующем разделе данной книги.

Последовательное выполнение

Простейшая структура программы: «сделай это, потом то». Эти структуры анализируются путем расстановки между ними утверждений и разбора каждого из операторов.

Ветвление

Ветвление осуществляется операторами if и case в различных формах. При выполнении программы выбирается одна из возможных ветвей. Корректность доказывается путем индивидуального разбора каждой из ветвей. Факт выбора данной ветви дает возможность сделать некоторые утверждения в нашем доказательстве; если мы выполняем оператор, следующий за if i>j, то мы можем сделать утверждение, что i>j, и использовать это для вывода следующего подходящего утверждения.

Циклы

Для доказательства корректности цикла нужно рассмотреть три его части: инициализацию, сохранение (инварианта) и завершение (рис. 4.1).

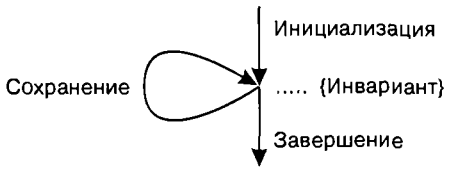


Рис. 4.1. Три составные части цикла

Сначала мы доказываем, что инвариант цикла устанавливается при инициализации, а затем показываем, что на всех итерациях его истинность сохраняется. Эти два этапа гарантируют (по методу математической индукции), что инвариант остается истинным перед и после каждой итерации цикла. Третий этап: доказательство того, что при любом варианте завершения цикла результат оказывается корректным. Все это вместе гарантирует, что если цикл завершится, то результат будет правильным. Сходимость доказывается отдельно (типичный пример — доказательство сходимости двоичного поиска, приведенное выше).

Функции

Для верификации функции мы должны определить ее назначение с помощью двух утверждений. Предусловие определяет состояние, в котором должна находиться программа в момент вызова функции, а постусловием называется то, что будет гарантироваться функцией при ее завершении. Таким образом мы можем определить функцию двоичного поиска на языке C следующим образом:

```
int bsearch(int t, int x[], int n)
/* precondition  x[0] <= x[1] <=       <= x[n-1]
    postcondition
        result == -1 => t not present in x
        0 <= result < n => x[result] == t
*/
```

Эти условия представляют собой скорее соглашения, нежели факты. В них утверждается, что если выполнены предусловия, то выполнение функции приведет к истинности постусловий. После того как будет доказано то, что тело функции обладает этим свойством, соотношением между пред- и постусловиями можно будет пользоваться без углубления в подробности реализации. Этот подход к написанию программного обеспечения называется «программированием по контракту».

4.5. Смысл верификации программ

Когда один программист пытается убедить другого в правильности своей программы, он обычно прибегает к помощи тестов. Берется программа и какой-либо вариант выходных данных, а затем она выполняется вручную. Это мощное средство

для поиска ошибок, которым легко пользоваться. Одно из главных преимуществ верификации программ описанным выше способом заключается в том, что программист получает язык, на котором он может выразить свое понимание текста программы.

Далее в главах 9, 11 и 14 данной книги мы воспользуемся методами верификации при разработке сложных программ. Каждую новую строку кода мы будем объяснять на этом языке; он особенно удобен для выбора инвариантов цикла. Важные пояснения будут превращаться в утверждения в тексте программы. Выбор утверждений, которые нужно включать в настоящие программы, — это искусство, которым можно овладеть лишь на практике.

Язык верификации часто используется после написания кода при его мысленном прогоне. Нарушения утверждений во время тестирования помогают обнаружить ошибки, а анализ этих нарушений позволяет избавиться от ошибок, не наплодив новых. Во время отладки следует исправлять и код, и неправильные утверждения. Нужно понимать свою программу в каждый момент времени и не поддаваться искушению «менять что-нибудь, пока она не заработает». В следующей главе иллюстрируется роль утверждений при тестировании и отладке. Утверждения жизненно важны в процессе сопровождения программы: если вы пытаетесь разобраться в программе, которую никогда раньше не видели и которую никто другой тоже не видел уже много лет, утверждения о состоянии программы могут дать вам неоценимую информацию.

Эти методы — лишь малая часть того, что нужно знать, чтобы правильно писать программы. Ключом к правильности часто является простота кода. С другой стороны, несколько профессиональных программистов, знакомых с этими методиками, поделились со мной опытом, который был не чужд и мне самому: при написании программы «сложная» ее часть обычно работает с первого раза, тогда как ошибки обычно содержатся в простых частях. Когда вы сталкиваетесь с чем-то сложным, вы сосредоточиваетесь и используете мощные формальные методы. В легких местах вы возвращаетесь к старым методам программирования и получаете характерные для этих методов результаты. Я бы сам в это не поверил, если бы это не случилось со мной. Подобные вещи являются хорошей причиной для более частого использования правильных методов.

4.6. Задачи

1. Хотя наше доказательство правильности двоичного поиска было достаточно трудоемким, оно все еще не вполне закончено. Как вы докажете, что программа не содержит ошибок времени выполнения (деление на ноль, переполнение, выход за границы диапазона или массива)? Если вы знакомы с дискретной математикой, можете ли вы формализовать доказательство в некоторой логической системе?

2. Если этот вариант двоичного поиска был для вас слишком прост, попробуйте возвращать в переменной `p` позицию первого вхождения `t` в массив `x` (если вхождений несколько, наш алгоритм возвращал произвольное вхождение). Ваш код должен быть логарифмическим по числу сравнений; задачу можно решить за $\log_2 n$ сравнений.

3. Напишите и проверьте рекурсивную программу двоичного поиска. Какие части кода и доказательства остаются неизменными по сравнению с итерационной версией, а какие меняются?
4. Добавьте вспомогательные переменные для определения количества сравнений и используйте методы проверки программ для доказательства того, что это количество действительно пропорционально логарифму размера массива.
5. Докажите, что программа завершает работу, если x — положительное целое число:

```
while x != 1 do
    if even(x)
        x = x/2
    else
        x = 3*x + 1
```

6. [C. Sholten] Дэвид Грис в своей книге «Наука программирования» (David Gries, Science of Programming) назвал эту задачу «Задачей о кофейной банке». Дается кофейная банка, в которой есть несколько черных бобов и несколько белых, и, кроме того, дается большая куча черных бобов. Затем следующий процесс выполняется до тех пор, пока в банке не останется один боб.

   Случайным образом выбираются два боба. Если они одного цвета, они выкидываются, а вместо них из кучи берется черный и кладется в банку. Если они разных цветов, белый боб возвращается в банку, а черный выбрасывается.

   Докажите, что процесс сходится. Что вы можете сказать о цвете последнего оставшегося боба в зависимости от начального количества белых и черных бобов?
7. Мой коллега столкнулся со следующей задачей при написании программы, рисовавшей линии на растровом экране. Массив из n пар вещественных чисел $(a_i b_i)$ определял n линий вида $y = a_i x + b_i$. Эти строки были упорядочены на интервале [0,1] в том смысле, что y[i] < y[i] + 1 для всех i между 0 и n-2 и всех значений x из отрезка [0,1] (рис. 4.2).

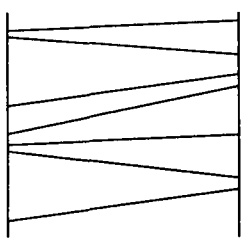


Рис. 4.2. Линии не имели общих точек на вертикальных прямых

Другими словами, линии не имели общих точек на вертикальных прямых (x = 0 и x = 1). Ему нужно было найти две линии, между которыми находилась произвольная точка с координатами (x, y), где x принадлежала отрезку [0,1]. Как можно быстро решить эту задачу?

8. Двоичный поиск работает принципиально быстрее последовательного. Для поиска в массиве из n элементов в двоичном поиске выполняется примерно $\log_2 n$ сравнений, а для последовательного — в среднем $n/2$. Хотя часто этого вполне достаточно, иногда приходится добиваться еще более высокого быстродействия. Если нельзя улучшить логарифмическую зависимость числа сравнений, можно ли переписать код так, чтобы он стал работать быстрее? Для определенности предположите, что вам нужно найти число в отсортированном массиве из n=1000 целых чисел.

9. В качестве упражнения в проверке программ попробуйте точно описать поведение на входе и выходе следующих программных блоков и покажите, что код соответствует требованиям. Первая программа реализует векторное сложение a = b + c.

```
i = 0
while i < n
    a[i] = b[i] + c[i]
    i = i+1
```

Этот и следующие два фрагмента используют цикл while вместо for i = [0, n). В следующем фрагменте вычисляется максимальное значение в массиве x.

```
max = x[0]
i = 1
while i<n do
    if x[i] > max
        max = x[i]
    i = i+1
```

А эта программа последовательного поиска возвращает первое вхождение t в массив x[0..n-1].

```
i = 0
while i<n && x[i] != t
    i = i+1
if i >= n
    p = -1
else
    p = i
```

Эта программа вычисляет n-ю степень числа x за время, пропорциональное логарифму n. Такую рекурсивную программу легко закодировать и проверить; итеративная версия достаточно сложна и предоставляется читателю в качестве дополнительного задания.

```
function exp(x, n)
        pre n >= 0
    post result = x^n
if n = 0
    return 1
else if even(n)
    return square(exp(x, n/2))
else
    return x*exp(x, n-1)
```

10. Добавьте ошибки в функцию двоичного поиска и посмотрите, сможете ли вы их обнаружить с помощью метода проверки и в чем они будут проявляться.

11. Напишите и докажите правильность рекурсивной функции двоичного поиска на языке C или C++, причем функция должна соответствовать следующей декларации:

```
int binarysearch(DataType x[], int n)
```

Используйте только одну эту функцию и не вызывайте никаких других.

4.7. Дополнительная литература

«Наука программирования» Дэвида Гриса (David Gries, Science of Programming, Springer-Verlag, 1987) является превосходным введением в область методик верификации программ. Начинается она с курса логики, затем рассматривает верификацию и разработку программ с формальной точки зрения и, наконец, переходит к обсуждению программирования на конкретных языках. В этой главе я попытался набросать потенциальные преимущества метода верификации программ; единственный способ для программиста научиться эффективно пользоваться этими методиками — изучить их по книге наподобие указанной.

Немного программирования

Пока что вы все сделали правильно. Вы проникли в самую суть задачи. Вы смогли удовлетворить всем требованиям, правильно выбрав алгоритм и структуры данных. Вы использовали методы верификации программ для написания красивого псевдокода, в правильности которого вы уверены на все 100%. Как же вам вставить получившийся бриллиант в оправу? Как добавить новую функцию в большую программу? Для этого необходимо приложить немного усилий в области программирования.

Программисты в большинстве своем склонны впадать в искушение пойти по легкому пути: записать функцию на одном из языков и включить ее в систему, а дальше надеяться всей душой, что она будет работать. В одном случае из 1000 этот метод действительно срабатывает. Но в прочих 999 такая стратегия ведет к катастрофе.

Мудрые программисты пишут тестовые программы (scaffolding) для упрощения проверки функции. Данная глава посвящена реализации двоичного поиска, написанного на псевдокоде и представленного в предыдущей главе, в виде функции на языке C. На языках C++ и Java код будет выглядеть почти так же, а метод в целом будет верен практически для всех языков программирования. Закончив кодировать функцию, мы проверим ее с помощью тестовой программы, переходя от простых ко все более сложным действиям, в частности к оценке времени выполнения программы. Эта процедура может показаться слишком долгой для такой маленькой функции, но зато в результате получим программу, в работоспособности которой мы будем уверены.

5.1. От псевдокода к C

Предположим, что массив `x` и искомый объект `t` принадлежат к одному типу — `DataType`. Тип этот можно будет определить оператором `typedef`:

```
typedef int DataType
```

Таким образом можно определить и длинное целое вещественное число с плавающей точкой и любой другой тип. Массив реализуется с помощью двух глобальных переменных:

```
int n,
DataType x[MAXN]
```

Несмотря на то что это «дурной тон» программирования для языка C, он соответствует способу доступа к данным в классах C++; кроме того, глобальные переменные позволяют уменьшить объем тестовой программы. Итак, мы должны получить функцию на языке C, отвечающую следующим требованиям:

```
int binarysearch(DataType t)
/*  precondition  x[0]<=x[1]<=  .<=x[n-1]
    postcondition
    result == -1 => элемент t в массиве x отсутствует
    0 <= result < n => x[resul] == t
*/
```

Бóльшая часть операторов на псевдокоде (см., например, листинг 4.3) непосредственно переводятся на язык C строчка за строчкой (как и на большинство других языков). Когда программа на псевдокоде помещает значение в переменную p, функция на C будет возвращать это значение оператором return. Оператор loop на псевдокоде заменяется бесконечным циклом на C. В результате получаем следующий код:

```
for (..) {
    if (l > u)
        return -1,
    /* и так далее */
}
```

Мы можем переделать это в цикл «пока» (while), изменив условие проверки:

```
while (l<=u) {
    /* тело цикла */
}
```

В итоге получим законченный вариант нашей функции двоичного поиска на языке C (листинг 5.1).

Листинг 5.1. Функция двоичного поиска на языке C

```
int binarysearch(DataType t)
    /* возвращает любое вхождение t в отсортированный массив x[0  n-1]
либо -1 в случае отсутствия вхождений */
{    int l, u, m,
    l = 0,
    u = n-1,
    while (l <= u) {
        m = (l + u)/2,
        if (x[m]<t)
            l= m+1;
        else if (x[m]==t)
            return m,
        else /* x[m]>t */
            u = m-1,
    }
    return -1,
}
```

5.2. Тестовая программа

Проверку функции начинают с попытки выполнения ее вручную. Тестовые примеры с массивами из 0, 1 и 2 элементов зачастую позволяют отловить мелкие ошибки. Для массивов большего размера производить проверку вручную оказывается слишком утомительным, поэтому следует перейти к написанию тестовой программы, вызывающей проверяемую функцию. В нашем случае для этого оказывается достаточно пяти строк на языке С.

Листинг 5.2. Тестовая программа, вызывающая функцию двоичного поиска

```
while (scanf("%d %d", &n, &t) != EOF) {
    for (i=0; i<n; i++)
        x[i]=10*i;
    printf(" %d\n", binarysearch(t));
}
```

Мы можем начать с программы из двадцати строк (объединив листинги 5.1 и 5.2). Однако следует ожидать дальнейшего увеличения размеров программы по мере усложнения предлагаемых тестов.

При вводе чисел 2 и 0 тестовая часть создаст массив из двух элементов, причем x[0]=0, x[1]=10, а затем выведет результат поиска: элемент 0 найден в позиции 0:

```
2 0
 0
2 10
 1
2 -5
 -1
2 5
 -1
2 15
 -1
```

Вводимый текст выделен **полужирным**. Мы видим, что число 10 также благополучно обнаруживается функцией поиска в позиции 1. Последние 6 строк вывода демонстрируют три попытки поиска отсутствующих элементов. Каждый раз возвращаемое значение было правильным (-1). Итак, программа корректно обрабатывает все возможные варианты поиска в массиве с двумя различными элементами. По мере того как программа проходит аналогичные тесты на массивах большего размера, мы с радостью убеждаемся в ее правильности. Но при работе с большими массивами необходимость вводить что-либо с клавиатуры начинает занимать слишком много времени, поэтому мы создаем тестовую программу для автоматизации проверки (приведена в разделе 5.4).

Далеко не все программы проходят такой тест. Ниже приведена функция двоичного поиска, которую мне предлагали в качестве решения несколько профессиональных программистов.

Листинг 5.3. Функция двоичного поиска (с ошибками)

```
int badsearch(DataType t)
{    int l,u,m;
     l=0;
```

```
    u=n-1;
    while (l<=u) {
        m = (l + u)/2;
/* printf("  %d %d %d\n", l, m, u); */
        if (x[m]<t)
            l = m;
        else if (x[m]>t)
            u = m;
        else
            return m;
    }
    return -1;
}
```

К закомментированному оператору printf мы вскоре вернемся. Попытаемся обнаружить ошибки в этом коде?

Итак, программа благополучно прошла первые два теста. Она нашла 20 в позиции 2 и 30 в позиции 3 в 5-элементном массиве:

```
5 20
 2
5 30
 3
5 40
```

Когда я ввел команду на поиск числа 40, программа зависла в бесконечном цикле. Почему?

Для получения ответа на этот вопрос я вставил оператор printf, который в листинге 5.3 закомментирован. Он выводит значения l, m и u на каждой итерации цикла:

```
5 20
  0 2 4
 2
5 30
  0 2 4
  2 3 4
 3
5 40
  0 2 4
  2 3 4
  3 3 4
  3 3 4
  3 3 4
```

В первый раз мы находим 20 с первой попытки, а во второй раз находим 30 со второй попытки. С третьей попытки программа уходит в бесконечный цикл. Мы могли бы найти ошибку, если бы попытались (безуспешно) доказать сходимость алгоритма.

Когда мне нужно отладить небольшой алгоритм в большой программе, я иногда использую пошаговое выполнение большой программы. При работе с алгоритмом с использованием тестовых программ, как в данном случае, быстрее и эффективнее оказывается вставить операторы printf, чем пользоваться хитрыми отладчиками.

5.3. Искусство вставки утверждений

Как мы рассмотрели ранее, в главе 4, утверждения оказались весьма полезными при разработке алгоритма двоичного поиска. Они направляли нас при его написании и помогали доказать его корректность. Теперь мы попытаемся вставить их в код нашей программы, чтобы убедиться, что она выполняется именно так, как мы это себе представляли.

Для выражения уверенности в том, что некоторое утверждение является истинным, мы будем использовать оператор assert. Например, оператор assert(n>=0) не выполнит никаких действий, если n будет неотрицательно, но выдаст какое-либо сообщение об ошибке, если n окажется отрицательно (например, запустит отладчик). Перед тем как возвращать позицию найденного элемента, мы можем вставить соответствующее утверждение:

```
else if (x[m] == t) {
    assert(x[m] == t);
    return m;
} else
    . . .
```

Это утверждение просто повторяет условие оператора if. Мы можем усилить утверждение, введя дополнительные требования, чтобы найденная позиция принадлежала исходному диапазону [0..n-1]:

```
assert(0 <= m && m < n && x[m] == t);
```

При завершении цикла в том случае, когда искомый элемент отсутствует, мы знаем, что нижняя и верхняя границы l и u «пересеклись», и делаем вывод, что элемент в массиве отсутствует. Может возникнуть желание выразить это утверждение в форме, в которой оно будет означать, что мы нашли два соседних элемента, ограничивающих искомый:

```
assert(x[u]<t && x[u+1]>t);
return -1;
```

Логика тут в том, что если мы видим стоящие рядом в отсортированном массиве элементы 1 и 3, то мы можем быть уверены, что элемента 2 в массиве нет. Однако это утверждение будет время от времени приводить к ошибке даже в правильной программе. Почему?

Когда количество элементов n=0, переменная u инициализируется значением -1, поэтому обращение к элементу с этим индексом приведет к выходу за границы массива. Чтобы это утверждение заработало, нужно слегка изменить его, добавив проверку границ:

```
assert((u<0 || x[u] < t) && (u+1 >= n || x[u+1] > t));
```

Это утверждение позволит найти ошибки в некорректных программах.

Мы показали сходимость поиска, доказав, что с каждой итерацией диапазон сужается. Этот факт можно также проверить во время выполнения программы, добавив немного дополнительных вычислений и одно утверждение. Инициализируем переменную size значением n+1, а затем добавим следующие строки после оператора for:

```
oldsize = size.
size = u - l + 1.
assert(size < oldsize);
```

Мне стыдно признаться, сколько раз я напрасно пытался отладить функцию двоичного поиска, когда дело было в том, что входной массив не был отсортирован. Определив функцию sorted, приведенную ниже, мы сможем высказать утверждение assert sorted()) (листинг 5.4).

Листинг 5.4. Проверка массива на упорядоченность

```
int sorted()
{    int i.
    for (i = 0. i < n-1. i++)
        if (x[i] > x[i+1])
            return 0.
    return 1.
}
```

Следует быть аккуратным, если вы хотите выполнять эту ресурсоемкую проверку только один раз, перед выполнением поиска. Добавление проверки в цикл приведет к тому, что двоичный поиск будет выполняться за время, пропорциональное $n \log n$[1].

Утверждения удобны для тестирования функции внутри тестирующей программы, а также по мере перехода от проверки компонентов к проверке системы. Некоторые проекты определяют утверждение assert с помощью программы предварительной обработки, так, чтобы эти утверждения были удалены при компиляции и не приводили к «накладным расходам» при выполнении программы. С другой стороны, Тони Хоар сравнивал программиста, использующего утверждения только при отладке и отключающего их при финальной компиляции программы, с моряком, который носит спасательный жилет на берегу и снимает его, выходя в плавание.

Глава 2 книги Стива Магуира «Создание серьезных программ» (Steve Magiure, Writing Solid Code, Microsoft Press, 1993) посвящена использованию утверждений в промышленных программных системах. Он подробно излагает несколько жизненных примеров использования утверждений в продуктах и библиотеках Microsoft.

5.4. Автоматизация тестирования

Предположим, что вы уже наигрались со своей программой и почти уверены в своей непогрешимости. Вам надоело скармливать ей тестовые примеры вручную. Следующий этап — построение тестовой программы, предусматривающей авто-

---

[1] Основание логарифма при оценке времени работы программы обычно не пишется. Дело в том, что данное выражение означает не то, что программа выполнится, скажем, за $n \log n$ тактов, а то, что при $n \to \infty$ время выполнения программы растет так же, как функция $n \log n$. Логарифм по любому основанию мы всегда можем выразить через натуральный: $n \log_a n = n \frac{\ln n}{\ln a} = \left(\frac{1}{\ln a}\right)(n \ln n)$. Первый сомножитель — константа, и функция растет как $n \ln n$. Поэтому указывать какое-либо конкретное значение основания логарифма не имеет смысла. При реальных прикидках чаще всего используется основание 2. — *Примеч. ред.*

матизированное тестирование. Главный цикл тестовой программы будет перебирать размер массива от наименьшего возможного значения 0 до наибольшего разумного:

```
for n=[0, maxn]
    print "n=", n
    /* test value n*/
```

Оператор print извещает программиста о ходе выполнения теста. Некоторые программисты ненавидят его: он лишь заполняет экран цифрами, не сообщая никакой полезной информации. Другие находят некоторое успокоение в наблюдении за ходом тестирования. Кроме того, знание точного места появления ошибок может оказаться полезным.

Первая часть цикла проверяет тот случай, когда все элементы в массиве различны. Кроме того, в массив добавляется лишний элемент, который не учитывается функцией поиска (листинг 5.5).

Листинг 5.5. Автоматизированное тестирование: все элементы различны

```
for i = [0,n]
    x[i] = 10*i
for i = [0, n)
    assert(s(10*i) == i)
    assert(s(10*i -5) == -1)
assert(s(10*n - 5) == -1)
assert(s(10* n) == -1)
```

Чтобы облегчить проверку различных функций, мы можем определить тестируемую функцию как макрос:

```
#define s binarysearch
```

Утверждения проверяют все возможные результаты успешных и неуспешных вызовов функции поиска. Кроме того, проверяется элемент, расположенный за границами диапазона поиска (он не должен быть найден).

Вторая часть цикла проверяет функцию поиска на массиве из одинаковых элементов (листинг 5.6).

Листинг 5.6. Автоматизированное тестирование: массив из одинаковых элементов

```
for i = [0, n)
    x[i] = 10
if n == 0
    assert (s(10) == -1)
else
    assert(0 <= s(10) && s(10) < n)
assert(s(15) == -1)
assert(s(5) == -1)
```

Мы проверяем функцию поиска имеющегося элемента, а также элементов, отличных от него.

Подобные проверки позволяют выявить почти все возможные ошибки. Диапазон значений n=1..100 позволяет проверить случай с пустым массивом, все стандартные размеры массивов (1, 2), несколько массивов с размерами, равными степеням двойки, и ещё некоторые, размеры которых отличаются от степеней двойки на 1. Выполнять эти тесты вручную было бы нестерпимо скучно. Компьютерного

же времени было затрачено совсем немного. При установке MAXN = 1000 на все эти тесты требуется всего несколько секунд работы моей машины.

5.5. Время выполнения

Расширенное тестирование укрепляет нашу уверенность в правильности программы. А как можно укрепить уверенность в правильности оценки времени выполнения алгоритма ($\log_2 n$)? Вот тест главного цикла программы определения времени работы (листинг 5.7).

Листинг 5.7. Определение времени работы алгоритма

```
while read(algnum, n, numtests)
    for i = [0, n)
        x[i] = i
    starttime = clock()
    for testnum = [0, numtests)
        for i = [0, n)
            switch(algnum)
                case 1: assert(binarysearch1(i) == i)
                case 2: assert(binarysearch2(i) == i)
    clicks = clock() - starttime
print algnum, n, numtests, clicks, clicks/(1e9 * CLOCKS_PER_SEC * n *
numtests)
```

Этот код вычисляет среднее время выполнения двоичного поиска в массиве из n различных элементов. Вначале массив инициализируется, затем производится поиск каждого элемента в массиве numtests раз. Оператор switch позволяет выбрать тестируемый алгоритм (тестовые программы всегда должны быть рассчитаны на проверку различных вариантов одного алгоритма). Оператор print выводит три входных значения и два выходных: количество «тиков» таймера (на эту величину всегда нужно обращать внимание) и среднее количество наносекунд на один поиск (ее проще интерпретировать).

Вот пример сеанса работы с такой программой на компьютере с процессором Pentium II 400 МГц, где вводимый мной текст выделен **полужирным**:

```
1 1000 10000
1    1000    10000    3445    344.5
1 10000 100
1    10000    1000    4436    443.6
1 100000 100
1    100000    100    5658    565.8
1 1000000 10
1    1000000    10    6619    661.9
```

Первая строка проверяет алгоритм 1 (рассмотренный нами алгоритм двоичного поиска) для массива из 1000 элементов. Выполняется 10 000 тестов. На это уходит 3445 тиков (в этой системе 1 тик соответствует одной миллисекунде), что дает 344,5 тика на операцию поиска. Каждая последующая проверка увеличивает n в 10 раз и соответственно уменьшает количество тестов в то же количество раз. Полное время выполнения всех тестов составляет примерно $50+30\log_2 n$ наносекунд.

Затем я написал программу из трех строк для создания массива данных для проверки скорости работы. Результаты представлены на графике, показывающем среднее время поиска, действительно ведущее себя как логарифм n. Задача 7 посвящена потенциальной ошибке в этой тестовой программе. Обязательно изучите ее, прежде чем верить моим данным.

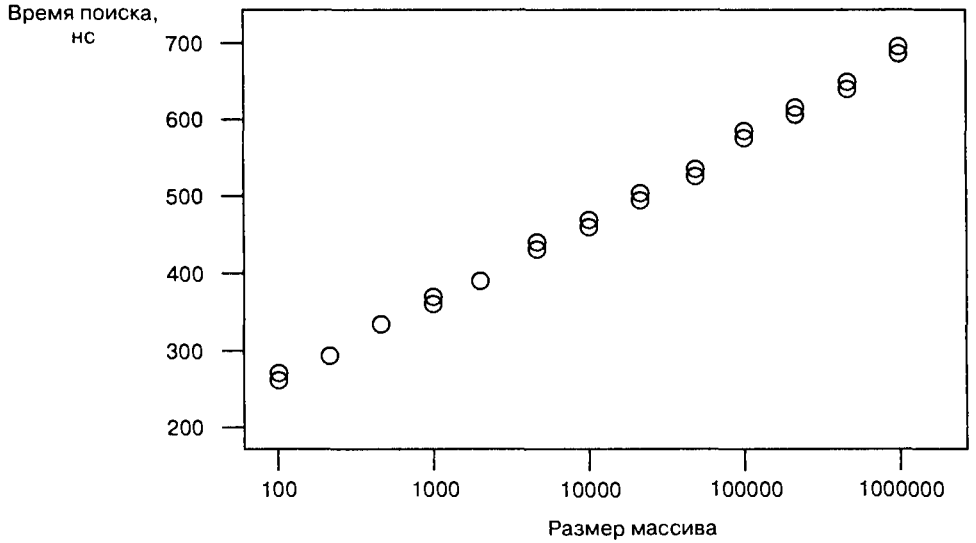


Рис. 5.1. Время выполнения двоичного поиска

5.6. Окончательная программа

Я верю, что моя реализация двоичного поиска на языке C верна. Почему? Я аккуратно написал псевдокод на подходящем языке, затем воспользовался аналитическими методами для его верификации. Я перевел программу с псевдокода на C строку за строкой и проверил ее работу на различных вариантах входных данных и наблюдал выводимые результаты. Я добавил в ее текст утверждения, чтобы убедиться, что реальная ее работа совпадает с теоретически ожидаемой. Компьютер тоже внес свой вклад, проверив программу огромным количеством тестов. Наконец, простейшие эксперименты показали, что время ее выполнения растет именно так, как и должно.

Теперь я могу спокойно использовать написанную мной функцию двоичного поиска в заранее отсортированном массиве в большой программе. Я был бы сильно удивлен, если бы в логике программы нашелся какой-нибудь изъян. Правда, меня не удивило бы появление других ошибок. Не забыл ли вызвавший функцию отсортировать массив? Действительно ли должно возвращаться значение -1, если элемента в массиве нет? Если искомый объект содержится в массиве в нескольких экземплярах, какой именно нужен пользователю? И так далее, и тому подобное.

Можете ли вы доверять этой программе? Можете на меня положиться. Можете скачать копию программы с сайта этой книги (http://netlib.bell-labs.com/cm/cs/pearls/). В нее включены все функции, которые мы уже видели, и несколько вариантов двоичного поиска, рассматриваемых далее в главе 9. Функция main этой программы выглядит следующим образом:

```
int main(void)
{    /* probe1(); */
    /* test(25); */
    timedriver();
    return 0;
}
```

Закомментировав все вызовы, кроме одного, вы можете работать с конкретными вариантами входных данных, проверять программу тестовыми массивами и измерять время ее работы.

5.7. Принципы

В этой главе мы затратили много усилий на решение небольшой задачи. Задача может казаться маленькой, но она не так проста. Вспомните, что, хотя первая версия двоичного поиска была опубликована в 1946, корректный код, обрабатывающий все значения n, появился лишь в 1962. Если бы программисты прошлого воспользовались методами, изложенными в этой главе, им бы не понадобилось потратить 16 лет на отладку двоичного поиска.

Тестовые программы

Лучшая тестовая программа — это та, которую проще всего написать. Для некоторых задач простейшая тестовая программа обязательно содержит графический интерфейс пользователя, реализованный на языках типа Visual Basic, Java или Tcl. При использовании каждого из этих языков мне приходилось тратить полчаса на небольшие программы, направленные на управление мышью и визуализацию вывода. Для решения многих алгоритмических задач мне кажется более разумным использовать более простые (и лучше переносимые) языки, используя текстовый интерфейс.

Кодирование

Я считаю наиболее удобным набросать сложную функцию на подходящем псевдокоде высокого уровня, а затем перевести ее на нужный язык программирования.

Тестирование

Гораздо проще проверить маленькую функцию, включив ее в состав тестовой программы, чем проверять ее в большой системе.

Отладка

Сложно отладить процедуру, изолированную в тестовой программе, но еще сложнее сделать это, если она уже включена в состав большой системы. В разделе 5.10 данной главы рассказывается о проблемах отладки больших систем.

Время работы

Если время работы не имеет значения, гораздо проще реализовать линейный поиск. Большинство программистов могут правильно его написать с первого раза. Поскольку для нас время работы оказалось достаточно важным, чтобы усложнить программу добавлением двоичного поиска, нам пришлось выполнить несколько экспериментов, чтобы удостовериться в правильности его поведения.

5.8. Задачи

1. Прокомментируйте стиль программирования, использованный в этой главе и в книге в целом. Обратите внимание на имена переменных, формат вызова и спецификацию функции двоичного поиска, формат программы и так далее.
2. Переведите описание функции двоичного поиска на псевдокоде в другие языки программирования и напишите тестовые программы для проверки этих функций. Чем может выбранный язык помочь программисту или, наоборот, затруднить дело?
3. Добавьте ошибок в функцию двоичного поиска. Обнаруживаются ли ошибки с помощью ваших тестов? Как помогает тестовая программа в отладке функции? Это упражнение лучше всего выполнять вдвоем, причем один из программистов должен вносить ошибки, а второй — их отлавливать.
4. Повторите упражнение 3, оставляя функцию в неприкосновенности и добавляя ошибки в вызывающий ее код (например, можно «забыть» отсортировать массив).
5. [R. S. Cox] Обычная ошибка — применение двоичного поиска к неотсортированному массиву. Проверять упорядоченность массива перед каждым вызовом — слишком расточительная процедура. Как можно добавить частичную проверку в нашу функцию с гораздо меньшими затратами ресурсов?
6. Напишите программу с графическим интерфейсом пользователя для изучения двоичного поиска. Стоит ли ваша «овчинка» (удобство отладки) выделки (время разработки)?
7. Программа проверки времени работы в разделе 5.5 содержит возможную ошибку. Если мы осуществляем поиск каждого следующего элемента, мы создаем очень удобную ситуацию для использования кэша. Если мы знаем, что в некотором приложении последовательные операции поиска будут

выполняться для соседних элементов массива, эта проверка отражает действительную ситуацию. Однако в этом случае двоичный поиск уже не будет самым оптимальным вариантом. Если мы ожидаем, что нужно будет искать случайные элементы массива, нам придется инициализировать и перетасовать вектор перестановок:

```
for i = [0, n)
    p[i] = i
scramble(p, n)
```

а затем производить поиск в случайном порядке:

```
assert(binarysearch1(p[i])) == p[i])
```

Измерьте и сравните скорости выполнения этих двух программ.

8. Тестовые программы используются в программистской практике, честно говоря, недостаточно и редко обсуждаются в книгах. Изучите все примеры тестовых программ, которые сможете найти; в крайнем случае загрузите их с сайта этой книги (http://netlib.bell-labs.com/cm/cs/pearls/). Напишите тестовую программу для проверки какой-нибудь сложной функции, которую вы написали.

9. Скачайте тестовую программу search.c с сайта этой книги и поэкспериментируйте с ней с целью определения времени выполнения функции двоичного поиска на своем компьютере. Какими средствами вы воспользуетесь для формирования исходных данных для поиска и для сохранения и анализа результатов?

5.9. Дополнительная литература

Книга Кернигана и Пайка «Практика программирования» (Kernighan, Pike, Practice of Programming) была выпущена издательством Addison-Wesley в 1999 году. Пятьдесят страниц этой книги посвящено отладке (глава 5) и тестированию (глава 6). Авторы рассматривают достаточно важные темы, такие как невоспроизводимые ошибки и регрессионное тестирование, которые остались за рамками этой книги.

Их книга из девяти глав покажется интересной и увлекательной любому программисту-практику. Помимо уже упомянутых тем, в книге рассматриваются вопросы стиля программирования, алгоритмы и структуры данных, разработка и реализация программ, интерфейсы, производительность, переносимость. В книге собраны ценные данные из собственного опыта работы авторов.

В разделе 3.8 нашей книги мы уже ссылались на фундаментальный труд Стива Макконнелла «Завершенный код» (Steve McConnell, Code Complete, Microsoft Press, 1993). В главе 25 этого труда рассматриваются вопросы тестирования модулей, а в главе 26 — отладка.

5.10. Отладка

Общеизвестный факт: отладка программ — процесс весьма трудоемкий. Опытные разработчики работают так, что со стороны может показаться, будто у них все получается легко. Обезумевшие от горя программисты описывают специалисту проявления ошибки, за которой они безуспешно охотились уже несколько часов, и через несколько минут эксперт находит ошибку. Опытный программист никогда не забывает о том, что должно существовать простое логическое объяснение любой странности в поведении системы.

Хороший пример — байка из исследовательского центра IBM Yourktown Heights Research Center. Программист настроил новую рабочую станцию. Все было хорошо, пока он сидел, но он не мог войти в систему, если он стоял. Поведение системы было на 100% воспроизводимо: он всегда мог войти в систему сидя и никогда не мог сделать этого стоя.

Большинство из нас просто посмеются над этой байкой. Как мог компьютер узнать, стоит человек или сидит? Специалисты по отладке уверены, что причину всегда можно отыскать. Легче всего высказать гипотезу об электрических наводках. Может быть, под ковром был какой-то провод, или дело в статическом электричестве? Но электрические помехи редко дают 100% воспроизводимость. Наконец был задан правильный вопрос: каким образом программист входил в систему сидя, и каким — стоя? Попробуйте поэкспериментировать сами.

Проблема была в клавиатуре. Две клавиши были переставлены местами. Когда программист сидел, он вводил текст вслепую и не обращал никакого внимания на наименование клавиш, однако когда он вставал, ему приходилось вводить текст, глядя на клавиатуру, поэтому пароль оказывался неправильным. Поняв это, эксперт быстро поменял клавиши, и проблема была решена.

Банковская компьютерная система в Чикаго исправно работала много месяцев, но неожиданно прекратила функционировать при попытке работы с международными данными. Программисты потратили много дней на проверку кода, но не могли найти ни одной команды, которая могла бы досрочно завершить программу. Когда они стали уточнять условия, при которых возникала проблема, они обнаружили, что программа завершала работу при вводе данных, относившихся к стране Эквадор. При более внимательном рассмотрении выяснилось, что, когда пользователь вводил название столицы Эквадора (Quito), программа воспринимала это как команду на завершение работы (quit).

Боб Мартин однажды наблюдал случай, когда программа срабатывала «много раз по одному разу». Первая транзакция обрабатывалась правильно, но во всех последующих возникали ошибки. После перезагрузки первая транзакция снова обрабатывалась правильно, а все последующие — неправильно. Когда Мартин сказал, что система работает «много раз по одному разу», программисты поняли, что нужно искать переменную, которая инициализировалась правильно в момент загрузки программы, но не сбрасывалась после обработки транзакции.

Во всех рассмотренных случаях правильные вопросы быстро наводили опытных программистов на скрытые ошибки: «Что вы делаете по-разному стоя и сидя?

Можно посмотреть, как вы входите в систему сидя, а как стоя?», «Что именно вы набирали в тот момент, когда программа завершила работу?», «Работала ли программа правильно вообще когда-нибудь? Сколько раз?»

Рик Лемонс как-то сказал, что лучший урок по отладке программ он получил на шоу фокусника. Фокусник исполнил около полудюжины невероятных трюков, и Лемонс почувствовал, что готов поверить в них. Затем он напомнил себе, что невозможное невозможно, и начал раздумывать над фокусами. Она начал с того, в чем был уверен — с законов физики, — и из них он смог получить простые объяснения для каждого фокуса. Это отношение к реальности делает Лемонса одним из лучших специалистов по отладке, которых я когда-либо видел.

Лучшая книга по отладке — это «Медицинские детективы» Бертона Руше (Berton Roueche, The Medical Detectives, Penguin, 1991). Герои этой книги занимаются поиском ошибок в сложных системах, начиная от слегка приболевших людей и заканчивая очень больными городами. Методы решения задач, которыми они пользуются, непосредственно применимы к отладке компьютерных систем. Эти правдивые истории увлекательнее иного вымысла.

Часть

Производительность

Простая и мощная программа, которая приводит в восторг пользователей и не досаждает создателям, — вот конечная цель программиста и предмет обсуждения предыдущих пяти глав.

Теперь сосредоточим наше внимание на одном конкретном свойстве хороших программ — эффективности. Малоэффективные программы огорчают пользователей долгими паузами и упущенными возможностями.

Итак, мы собираемся обсудить способы улучшения производительности.

В главе 6 приводятся возможные подходы и их комбинации. Три последующие главы посвящены трем основным методам повышения эффективности, расположенным в порядке их применения при разработке программы.

Глава 7 данной книги показывает, как предварительные оценки на ранних стадиях разработки позволяют убедиться в достаточной эффективности создаваемой программы.

Глава 8 посвящена методам разработки алгоритмов, некоторые из которых могут разительно сократить время работы модуля.

Глава 9 рассказывает об оптимизации программ, которая обычно проводится на последних стадиях реализации систем.

Глава 10, завершающая вторую часть данной книги, рассматривает дополнительный аспект эффективности — использование памяти.

Существуют три важные причины, по которым следует изучать методы повышения производительности программ. Во-первых, производительность важна для многих прикладных приложений. Я готов поспорить, что любому читателю этой книги не раз приходилось тщетно ждать завершения операции, глядя в монитор и мечтая о том, чтобы программы работали быстрее. Одна моя знакомая управляющая разработкой программного обеспечения считает, что половина затрат на разработку программ уходит на повышение производительности. К производительности некоторых программ, включая программы реального времени, базы данных и интерактивные программы, предъявляются особенно строгие требования.

Во-вторых, изучение методов повышения производительности полезно в общеобразовательном смысле. Помимо практической пользы, это неплохая тренировка для программиста. В главах этой части данной книги мы постараемся охватить широкий диапазон достижений человеческого разума, начиная с теории алгоритмов и заканчивая предварительными оценками, которые делаются исходя из соображений здравого смысла. Главное — это способность вашего ума воспринимать новую информацию. Более всего эту способность будет стимулировать глава 6, в которой мы будем рассматривать задачи с различных точек зрения.

Можно было бы, конечно, рассмотреть все эти темы на примере других задач, таких как разработка пользовательских интерфейсов или устойчивость и безопасность систем... Но *производительность* обладает ключевым преимуществом: ее можно измерить. Можно говорить о том, что одна программа работает в 2,5 раза быстрее другой, тогда как сравнение интерфейсов сводится к вопросам вкуса.

Самая главная причина, по которой нужно повышать производительность, лучше всего высказана в бессмертной фразе из фильма «Top Gun» (1986): «Я чувствую жажду... жажду скорости!»[1]

[1] Фраза на английском звучит так: «I feel the need ... the need for speed». — *Примеч. ред.*

Производительность в перспективе

Следующие три главы посвящены трем различным подходам к повышению производительности программ. В этой главе мы выясним, как можно использовать эти три подхода одновременно. Каждый из них применяется на своем уровне разработки компьютерной системы. Сначала мы займемся конкретной программой, а затем систематизируем наши соображения по поводу разных уровней разработки программ.

6.1. Пример

Возьмем статью Эндрю Эппеля об эффективном моделировании задачи многих тел (Andrew Appel, SIAM Journal on Scientific and Statistical Computing, Jan 1985, 6, 1, pp. 85–103). Применив многоуровневый подход к улучшению производительности, он сократил время работы программы с 1 года до 1 дня.

Программа решала классическую задачу многих тел — расчет взаимодействия n объектов (планет, звезд, галактик) в собственном гравитационном поле в трехмерном пространстве. Исходными данными служили их массы, начальные положения и скорости. В двухмерной задаче входные данные могли бы выглядеть следующим образом (рис. 6.1).

В статье Эппеля описываются две астрофизические задачи, в которых n=10 000. Изучая результаты моделирования, физики могли проверить соответствие теорий астрономическим наблюдениям. Подробнее о задаче и последующих решениях, в которых использовался подход Эппеля, рассказывается в книге Пфалзнера и Гиббона (Pfalzner, Gibbon, Many-Body Tree Methods in Physics, Cambridge University Press, 1996).

В первоначальном варианте моделирующей программы время делится на маленькие «шаги», и вычисляется перемещение каждого объекта на очередном шаге. Так как для каждого из объектов нужно учесть его взаимодействие со всеми прочими объектами, один шаг по времени требует порядка n^2 операций. Эппель подсчитал, что на выполнение 1000 шагов такого алгоритма при n=10 000 потребуется примерно год вычислений на имевшемся у него компьютере.

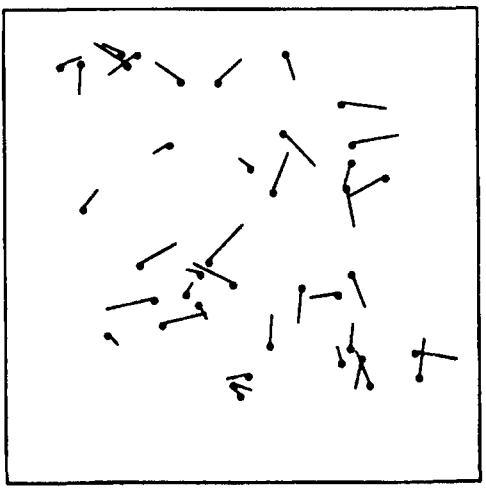


Рис. 6.1. Исходные данные задачи о многих телах

Последний вариант программы Эппеля решал задачу меньше чем за день, то есть в 400 раз быстрее, чем первый. С тех пор его методы использовались многими физиками. В нашем кратком пересказе его статьи будут пропущены многие важные детали, за которыми вам следует обратиться к упоминавшейся книге Пфалзнера и Гиббона. Самое главное, что огромное увеличение производительности было достигнуто благодаря многоуровневому подходу.

Алгоритмы и структуры данных

Прежде всего Эппель занялся поисками подходящего алгоритма. Ему удалось уменьшить затраты на выполнение одного шага по времени с $O(n^2)$ до $O(n\log n)$[1] благодаря выбору правильного представления объектов на бинарном дереве, где объекты верхнего уровня представляли собой кластеры физических объектов. Сила, действующая на конкретный объект, может вычисляться как сумма сил,

[1] Запись $O(n^2)$ можно понимать как «пропорционально n^2». Выражения типа $15n^2+100n$ и $n^2/2-10$ могут быть отнесены к этому классу. Формально запись $f(n)=O(g(n))$ означает, что существуют такие значения констант c и n_0, для которых $f(n)\le cg(n)$ при любом $n\ge n_0$. Точное определение можно найти в учебниках по теории алгоритмов и дискретной математике, а в разделе 8.5 (см. главу 8) иллюстрируется важность этого понятия в процессе разработки алгоритмов.

действующих на него со стороны больших кластеров. Эппель показывает, что это приближение не вносит погрешностей в модель. Дерево состоит из $\log_2 n$ уровней, а получающийся алгоритм с временем выполнения $O(n \log_2 n)$ близок по духу алгоритму «разделяй и властвуй» из раздела 8.3. Правильный выбор структуры данных уменьшил время выполнения исходной программы примерно в 12 раз.

Оптимизация алгоритма

Простейший вариант алгоритма выполняется с небольшим шагом по времени с целью корректно обработать редкую ситуацию, в которой два объекта оказываются близко друг к другу. Представление объектов в виде дерева позволяет обнаружить такие объекты и обработать их отдельно от прочих. Это позволяет увеличить шаг по времени вдвое и, соответственно, вдвое уменьшает общее время работы программы.

Реорганизация структуры данных

Дерево исходного набора объектов не очень хорошо подходит для представления этого набора в последующих состояниях. Реорганизация структуры на каждом шаге требует небольших вычислительных затрат, но зато уменьшает количество расчетов для близких объектов, что вдвое сокращает время выполнения программы.

Оптимизация кода

Благодаря использованию дерева точность вычислений повысилась, что позволило перейти от 64-битных чисел с двойной точностью к 32-битным — с одинарной. Эта замена также увеличила скорость работы программы вдвое. Профилирование программы показало, что 98% времени тратилось на выполнение одной функции. Эта функция была переписана на ассемблере, что увеличило скорость работы программы еще примерно в 2,5 раза.

Аппаратура

После всего вышеописанного для выполнения программы все еще требовалось 2 дня вычислений на компьютере, стоившем четверть миллиона долларов, и при этом запускать программу нужно было несколько раз. Эппель перенес программу на несколько более дорогой компьютер, оснащенный аппаратным устройством для выполнения операций с плавающей точкой, что вдвое уменьшило время выполнения программы.

Все эти усовершенствования привели к тому, что программа стала работать в 400 раз быстрее, чем первоначальный вариант. Однако эти достижения были не бесплатными. Простой вариант алгоритма умещается в несколько десятков строк,

тогда как быстрая программа занимает 1200 строк. Разработка и реализация быстрой программы потребовали нескольких месяцев работы. Данные о внесенных улучшениях приведены в табл. 6.1.

Таблица 6.1. Оптимизация программы «задача многих тел».

Уровень разработки	Коэффициент ускорения	Внесенные изменения
Алгоритмы и структуры данных	≈12	Двоичное дерево
Оптимизация алгоритма	≈2	Увеличение шага по времени
Реорганизация структуры данных	≈2	Подгонка кластеров под реальную ситуацию
Системно-независимая оптимизация кода	≈2	Замена двойной точности на одинарную
Системно-зависимая оптимизация кода	≈2,5	Кодирование критического участка на ассемблере
Аппаратура	≈2	Использование сопроцессора
Итого	400	

Эта таблица наглядно иллюстрирует сравнительную эффективность различных методов повышения производительности. Больше всего скорость работы повышает правильный выбор структуры данных — дерева, причем дальнейшие улучшения становятся возможными только благодаря этому выбору. Последние два действия (кодирование на ассемблере и использование сопроцессора) в данном случае не зависели от выбора структуры данных. Использование дерева не принесло бы результатов на суперкомпьютерах того времени (потоковая архитектура которых хорошо подходила для работы по простому алгоритму), так что алгоритмические улучшения не обязательно независимы от аппаратуры.

6.2. Уровни разработки

Компьютерная система разрабатывается на нескольких уровнях, начиная от программной структуры высокого уровня и заканчивая транзисторами аппаратного обеспечения. Приведенный ниже обзор основан на интуитивном подходе, так что не рассчитывайте на строгое формальное изложение[1].

Постановка задачи

Битва за быстродействие может быть выиграна или проиграна уже при постановке задачи. Когда я писал этот абзац, один из моих поставщиков сообщил, что не

[1] Я учился тому, что рассказываю в этой главе, по статье Р. Редди и А. Ньюэлла «Многократное ускорение работы систем» (в сборнике «Перспективы информатики» под ред. А. К. Джонса, Academic Press, 1977). В этой статье рассказывается о возможности увеличения производительности на различных уровнях, и в особенности о возможностях аппаратуры и системного программного обеспе-

смог доставить товар, поскольку заказ был потерян где-то по пути между моим департаментом и отделом снабжения моей компании. Отдел снабжения был наводнен однотипными заказами: только в моем департаменте 50 человек направили туда индивидуальные заказы. Дружеская беседа между нашим менеджером и менеджером отдела снабжения привела к тому, что эти заказы были объединены вместе, что упростило административную работу обоих отделов и ускорило работу одной небольшой части компьютерной системы в 50 раз. Хороший системный аналитик всегда следит за такими вещами, как до поставки системы заказчику, так и после того.

Иногда хорошая спецификация дает результат, превосходящий ожидания пользователя. В главе 1 мы видели, что иногда учет важных деталей задачи позволяет на порядок улучшить производительность и во столько же раз сократить длину кода программы. Зависимость эффективности программы от качества постановки задачи может быть достаточно сложной; к примеру, качественная обработка ошибок может несколько замедлить работу компилятора, но она ускорит работу с ним в среднем, поскольку уменьшено будет количество его запусков.

Структуризация системы

Деление большой системы на модули — важнейший этап определения ее будущей производительности. Набросав схему системы в целом, разработчику следует прикинуть, соответствует ли ее ожидаемая производительность заявленным требованиям. Подобные приближенные вычисления подробно рассматриваются в главе 7. Поскольку обычно проще повысить эффективность системы в процессе ее создания, а не после завершения основной части работы, анализ производительности является жизненно важной частью процесса разработки.

Алгоритмы и структуры данных

Скорость работы модуля обычно определяется структурами данных и алгоритмами, которые с ними работают. Наибольший выигрыш в производительности программы Эппеля был получен в результате замены алгоритма, работающего за $O(n^2)$ операций, на алгоритм, работающий за $O(n \log n)$ операций. В главах 2 и 8 данной книги описаны аналогичные примеры.

Оптимизация кода

Эппелю удалось ускорить свою программу в 5 раз путем внесения небольших изменений в ее код. Глава 9 целиком посвящена этому вопросу.

Системное программное обеспечение

Иногда легче поменять систему, в которой работает наша программа, чем саму эту программу. Например, в новой базе данных могут быстрее обрабатываться возни-

кающие запросы, или другая операционная система может лучше отвечать требованиям задач реального времени, или новый компилятор может лучше оптимизировать формируемый машинный код.

Аппаратное обеспечение

Быстрый компьютер может увеличить производительность программы. Компьютеры общего назначения обычно достаточно быстры, а ускорение для них может быть достигнуто увеличением тактовой частоты одного процессора или добавлением параллельно работающих процессоров. Звуковые карты, графические ускорители и прочие подобные устройства разгружают центральный процессор системы, а работа при этом выполняется небольшими и быстрыми специализированными процессорами. Разработчики игр всегда используют такие устройства для ускорения работы программ. Подобные специальные процессоры цифровой обработки сигналов позволяют детским игрушкам и бытовой технике разговаривать. Решение Эппеля — добавить к имеющейся машине ускоритель операций с плавающей точкой — лежит где-то между двумя крайностями.

6.3. Принципы

Поскольку предупреждение заболевания лучше лечения оного, нам следует всегда помнить о том, что заметил Г. Белл, когда он занимался созданием компьютеров для Digital Equipment Corporation: «*Самые дешевые, быстрые и надежные компоненты компьютерной системы — это те, которых в ней нет*».

Отсутствующие программные компоненты также являются наиболее правильными (в них никогда не возникает ошибок), самыми надежными (в них нельзя влезть), их проще всего разрабатывать, документировать, проверять и поддерживать. Важность простоты проекта трудно переоценить.

Однако бывает, что проблемы с производительностью нельзя просто обойти. Тогда помочь программисту может концепция многоуровневой разработки.

Если нужно ускорить систему незначительно

Если нужно ускорить систему незначительно, достаточно работать на наиболее удобном и подходящем «уровне». У большинства программистов есть свой способ повысить эффективность, действующий как коленный рефлекс: «*измени алгоритм*» или «*оптимизируй очередь*» — эти советы они дают сразу же. Прежде чем начать работать на каком-либо уровне, следует рассмотреть все остальные и выбрать тот, который дает наибольший выигрыш в производительности при наименьших усилиях.

Если нужно существенно ускорить работу

Если нужно существенно ускорить работу системы, работать следует на нескольких уровнях одновременно. Огромное повышение производительности, как в слу-

чае Эппеля, возможно лишь при атаке задачи со всех сторон, на нескольких уровнях, причем обычно это требует больших усилий. Когда изменения на одном уровне не зависят от изменений на других уровнях (как часто бывает), коэффициенты ускорения перемножаются.

В главах 7, 8 и 9 данной книги обсуждается ускорение программ на различных уровнях. При работе на конкретном уровне нужно всегда держать в голове общий вид задачи.

6.4. Задачи

1. Предположим, что компьютеры сейчас работают в 1000 раз быстрее, чем в те далекие дни, когда Эппель проводил свои эксперименты. Пусть на вычисления отводится то же время — около 1 дня. Как увеличится «размер задачи» (n) для $O(n^2)$ и $O(n \log n)$ алгоритмов?
2. Обсудите возможные улучшения на различных уровнях следующих задач:
   - перемножение 500-значных целых чисел;
   - Фурье-анализ;
   - моделирование цепей сверхбольших интегральных схем;
   - поиск строки в большом текстовом файле.

   Подумайте о взаимозависимостях предложенных улучшений.
3. В задаче Эппеля переход от двойной точности к одинарной увеличил скорость работы вдвое. Выберите подходящий тест и определите, как изменится время работы в вашей системе.
4. Эта глава посвящена скорости работы программы. Однако эффективность можно оценивать и по другим критериям: ошибкоустойчивость, надежность, защищенность, стоимость, отношение цена/производительность, точность, устойчивость к некорректному вводу и тому подобное. Обсудите, как можно подходить на различных уровнях к улучшению этих критериев.
5. Обсудите стоимость внедрения современных технологий на различных этапах разработки программы. Учтите все составляющие стоимости, включая время разработки (календарное и рабочее), стоимость обслуживания и стоимость продукта.
6. Старая и достаточно известная поговорка гласит: «эффективность вторична по отношению к правильности» — скорость программы не важна, если результаты ошибочны. Верно это или нет?
7. Обсудите, как можно подходить к проблемам повседневной жизни на разных уровнях. Рассмотрите, например, травмы, получаемые в автокатастрофах.

6.5. Дополнительная литература

Статья Батлера Лэмпсона «Советы разработчику компьютерных систем» (Butler Lempson, Hints for Computer System Design, IEEE Software 1, 1, January 1984). Бóльшая часть советов автора относится к производительности; лучше всего раскрыты вопросы интегрированной разработки систем на аппаратном и программном уровнях. Электронный вариант статьи можно найти по адресу: http://research.microsoft.com/~lampson/33-Hints/Abstract.html.

Предварительные оценки

Во время непринужденной беседы о разработке программного обеспечения Боб Мартин (Bob Martin) спросил меня: «Сколько воды вытекает из Миссисипи за день?» Так как до этого по его вопросам мне казалось, что мы хорошо понимаем друг друга, я вежливо переспросил: «Извините?» Когда он повторил свой вопрос, я понял, что мне остается лишь попытаться развеселить бедного парня, который, судя по всему, повредился в уме от перенапряжения, вызванного необходимостью руководить большим отделом программного обеспечения.

Мой ответ был примерно таким. Я оценил, что в это время года река имеет в ширину примерно милю, а в глубину — около 20 футов (то есть около 1/150 мили). Скорость течения я оценил как 5 миль в час или 120 миль в день. Я перемножил эти величины и получил:

```
1 миля * 1/250 мили * 120 миль/день = 1/2 куб  мили/день
```

Итак, по порядку величины из Миссисипи вытекало около половины кубической мили в день. Ну и что?

Тогда Мартин вытащил из ящика стола коммерческое предложение по системе коммуникаций, которую его организация готовила к летним Олимпийским играм, и проделал аналогичные вычисления. Он определил характерное значение одного из ключевых параметров, о которых мы говорили, определив время, необходимое для почтовой пересылки одного символа. Его вычисления были так же просты, как и для реки Миссисипи, и намного более показательны. Из них следовало, что предложенный вариант системы мог бы работать, только если бы в каждой минуте было по 120 секунд, а то и больше. Мартин отослал этот вариант обратно в комиссию по разработке в тот же день. Беседа происходила за год до Олимпийских игр, система была закончена в срок и работала безукоризненно.

Таким ярким (хотя и несколько эксцентричным) способом Мартин представил мне методику предварительных оценок. Инженеры в своих институтах с успехом используют ее по любому поводу, но в программировании предварительные оценки незаслуженно обойдены вниманием.

7.1. Основы мастерства

Приведенные ниже памятки могут оказаться полезными при проведении предварительных оценок.

Два ответа лучше, чем один

Когда я спросил Петера Вейнбергера (Peter Weinberger), сколько воды вытекает из Миссисипи за день, он ответил: «столько же, сколько и втекает». Затем он оценил, что бассейн водосбора этой реки составляет приблизительно 1000 на 1000 миль и что в среднем за год выпадает 1 фут осадков (1/50 000 мили). Это дает

```
1000 миль * 1000 миль * 1/5000 миль/год = 2000 куб. миль/год
2000 куб  миль/год / 400 дней/год = 1/2 куб  мили/день
```

Итак, получилось немногим более половины кубической мили в день. Очень важно дважды проверять все расчеты, а особенно «быстрые» предварительные оценки, подобные моей.

Можно сделать еще одну проверку — нечестную. В альманахе я нашел следующие данные: из реки вытекало 640 000 куб. футов в секунду. На основании этих данных получаем:

```
640 000 куб. футов/сек *3600 сек/час = 2.3*10^9 куб  футов/час
2 3*10^9 куб. футов/час * 24 часа/день = 6*10^10 куб  футов/день
6*10^10 куб. футов/день / (5000 футов/миля)^3 = 6*10^-0 куб. футов/день /
(125*10^9 куб  футов/куб  миля) = 60/125 куб. миль/день = 1/2 куб. мили/день
```

Такое точное совпадение оценок друг с другом и с ответом из альманаха является прекрасным примером слепого везения, на которое не стоит особо рассчитывать.

Быстрые проверки

Полья (Polya) посвящает три страницы своей книги «Как это решается» (How to solve it) проверке размерности, которую он характеризует как *«хорошо известное, быстрое и эффективное средство проверки геометрических и физических формул»*.

Первое правило: размерность всех слагаемых должна совпадать друг с другом и с размерностью результата — можно складывать футы с футами и получать снова футы, но нельзя сложить футы с секундами.

Второе правило: размерность произведения равна произведению размерностей. Приведенные выше примеры иллюстрируют оба правила:

```
(мили + мили) * мили * мили/день = куб  мили/день
```

Простая таблица позволит уследить за размерностями в сложных выражениях, подобных вышеприведенному. Сначала следует записать три сомножителя:

1000 миль	1000 миль	1 миля
		5000 дней

Затем сократить это выражение, что даст результат в 200 куб. миль в год:

~~1000 миль~~	~~1000 миль~~	~~1 миля~~	200 куб. миль
		~~5000~~ дней	

Потом мы умножаем результат на год, состоящий из 400 дней:

~~1000~~ ~~миль~~	~~1000~~ ~~миль~~	~~1~~ ~~миля~~	~~200~~ куб. миль	годы
		~~5000~~ дней		400 дней

После сокращения получаем уже знакомый результат: 0,5 куб. мили в день.

~~1000~~ ~~миль~~	~~1000~~ ~~миль~~	~~1~~ ~~миля~~	~~200~~ куб. миль	~~годы~~	1
		~~5000~~ ~~дней~~		~~400~~ дней	2

Таблица помогает следить за размерностями во время вычислений.

Размерность позволяет проверить вид уравнений. Проверяйте произведение и частное с помощью правила, известного со времен логарифмической линейки: *мантиссу и экспоненту можно вычислять независимо.* Сумму также можно легко проверить. Рассмотрим три возможных варианта результата сложения:

```
 3142        3142        3142
 2718        2718        2718
+1123       +1123       +1123
-----       -----       -----
  983        6982        6973
```

В первой сумме слишком мало цифр, а во второй есть ошибка в последнем знаке (который легко проверить). Метод «отбрасывания девяток» позволяет убедиться в ошибочности третьего результата: сумма цифр слагаемых дает 8 по модулю 9[1], а сумма цифр ответа дает 7 по модулю 9. При правильном суммировании суммы цифр слагаемых и результата должны давать один и тот же результат по модулю 9.

Помимо всего прочего не следует забывать и о здравом смысле. Следует усомниться в вычислениях, в результате которых получается, что из Миссисипи вытекает 400 литров воды в день.

Правила большого пальца

Впервые я выучил «*правило 72*» на курсах по бухгалтерскому учету. Предположим, что вы кладете сумму на y лет, а ставка составляет r процентов в год. Правило гласит, что если $y \times r = 72$, то сумма удвоится. Это приближение достаточно точно: положив $1000 под 6% на 12 лет, мы получим $2012, а положив ту же тысячу под 8% на 9 лет, получим $1999. «Правило 72» применимо к любому экспоненциальному процессу. Если колония бактерий растет со скоростью 3% в час, то она удвоится в размере за день.

Любое удвоение возвращает программиста к *правилу большого пальца*, поскольку $2^{10} = 1024$, то есть десять удвоений — это примерно тысяча, 20 удвоений — миллион и так далее.

Предположим, что программе с экспоненциальным алгоритмом требуется десять секунд на некоторую задачу при n=40. Увеличение n на 1 увеличивает время выполнения на 12% (это мы могли узнать, построив график результатов измерений в полулогарифмическом масштабе). «Правило 72» говорит нам, что время выполнения удваивается при увеличении n на 6, то есть увеличивается в 1000 раз, когда n увеличивается на 60. При n=100 программе потребуется около 10 000 секунд

[1] Остаток от деления на 9. — *Примеч. ред.*

(несколько часов). А что произойдет, когда n достигнет 160, а время станет равным 10^7 секунд? Много это или мало?

Вам может показаться сложным запомнить, что в году $3{,}155 \times 10^7$ секунд. Однако правило Тома Даффа (Tom Duff) — π секунд равно нановеку — запомнить гораздо проще. Поскольку программе требуется около 10^7 секунд, нам придется подождать несколько месяцев.

Практика

Умение делать правильные оценки приходит лишь с опытом. Попробуйте справиться с задачами в конце этой главы и с опросником в приложении 2 (аналогичный опросник однажды научил меня скромности в том, что касается предварительных оценок). Раздел 7.8 данной главы рассказывает о быстрых вычислениях в повседневной жизни. Где бы вы ни работали, у вас наверняка есть множество возможностей попрактиковаться в предварительных оценках. Сколько орешков в пакетике арахиса? Сколько времени ваши коллеги проводят в очередях (кофе, обед, ксерокс и так далее)? Сколько всего заработной платы выплачивается сотрудникам вашей компании? А когда вам действительно станет скучно за обедом, спросите коллег о том, сколько воды вытекает из Миссисипи за день.

7.2. Оценка производительности

Обратимся теперь к быстрым вычислениям в компьютерной технике. Пусть узел структуры данных (например, связного списка или хэш-таблицы) содержит целое число и указатель на другой узел:

```
struct node { int i; struct node *p; };
```

Задача на предварительную оценку: поместятся ли два миллиона таких узлов в 128 мегабайт памяти вашего компьютера?

Системный монитор у меня на машине показывает, что обычно из 128 Мбайт свободно около 85. (Я проверил этот факт, изменяя программу сдвига массива из главы 2 данной книги до тех пор, пока не начался свопинг на диск.) Но сколько памяти занимает один узел? Во времена 16-битных компьютеров указатель и целое заняли бы вместе 4 байта. Во время подготовки этого издания чаще всего использовались 32-битные целые и указатели, поэтому ответ будет 8 байт. Часто мне приходится компилировать программы и в 64-битном режиме, поэтому узел может занять и 16 байт. Ответ в конкретной системе можно получить, выполнив одну строку на языке C:

```
printf("sizeof(struct node)=%d\n", sizeof(struct node));
```

В моей системе на каждую запись отводится 8 байт, как я и предполагал. 16 Мбайт данных легко поместятся в свободных 85 Мбайт оперативной памяти.

Но почему же в действительности мой компьютер со 128 Мбайт памяти начал «дергать винтом» как сумасшедший, когда я создал 2 миллиона таких записей? Ответ прост: я использовал динамическое выделение памяти, используя функцию malloc языка C (аналогичную оператору new языка C++). Я предполагал, что на узел может приходиться 8 байт дополнительной информации, и тогда все данные заняли бы 32 Мбайт. На самом деле на запись приходилось еще 40 байт дополни-

тельной информации, итого: 48 байт. Два миллиона записей заняли, таким образом, 96 Мбайт. (В некоторых системах и компиляторах на одну запись действительно приходилось 8 байт дополнительной информации.)

В приложении 3 описана программа, определяющая объем памяти, занимаемый некоторыми часто используемыми структурами. Первые строки вывода получены с помощью оператора sizeof:

```
sizeof(char) = 1  sizeof(short) = 2  sizeof(int) = 4  sizeof(float) = 4
sizeof (struct *) = 4  sizeof(long) = 4  sizeof(double) = 8
```

Именно такие значения я и ожидал получить от своего 32-битного компилятора. Далее программа вычисляла разницу между последовательными указателями, возвращаемыми подпрограммой выделения памяти; это правдоподобный способ определить объем записи. (Подобные грубые догадки всегда следует проверять другими средствами.) Так я узнал, что благодаря этой подпрограмме-пожирателю памяти запись объемом 1–12 байт занимает 48 байт, запись объемом 13–28 байт занимает 64 байта памяти и так далее. Мы вернемся к этой модели памяти далее в главах 10 и 13.

Попробуем решить еще одну задачку. Вы знаете, что время работы вашего алгоритма определяется n^3 операций извлечения корня, причем n=1000. Сколько времени потребуется вашей программе на извлечение миллиарда кубических корней?

Чтобы найти ответ на свой вопрос, я начал с небольшой программы на C:

```
#include <math.h>
int main(void)
{
  int i, n = 1000000000;
  float fa;
  for (i = 0; i < n; i++)
    fa = sqrt(10.0);
  return 0;
}
```

Я запустил программу с помощью утилиты, позволяющей определить время выполнения. (Обычно я измеряю это время с помощью старых электронных часов с секундомером, которые лежат у меня рядом с компьютером; ремешок у этих часов давно оторвался, но они вполне годятся для определения времени выполнения программ.) Я выяснил, что программе требовалось 0,2 секунды на вычисление 1000 квадратных корней, 2 секунды на вычисление 10 миллионов и 20 секунд на вычисление 100 миллионов. На миллиард должно было потребоваться около 200 секунд.

Но может ли квадратный корень в реальной программе вычисляться за 200 наносекунд? Должно быть гораздо больше. Вероятно, функция вычисления квадратного корня кэшировала последний аргумент в качестве начального значения. Повторный вызов функции с тем же аргументом сильно ускорял ее работу по сравнению с реальной ситуацией. С другой стороны, в реальности функция могла работать и быстрее, потому что я компилировал программу, отключив оптимизацию (оптимизатор удалял основной цикл, поэтому программа выполнялась за нулевое время). Приложение 3 данной книги посвящено расширению этой маленькой программы в большую, определяющую время выполнения всех примитивных операций языка C в данной системе.

Как быстро работает сеть? Чтобы определить это, я использую команду ping имя_компьютера. Для связи с компьютером в том же здании требуется несколько

миллисекунд (это время инициализации связи). В удачный день я могу связаться с компьютером на другом побережье США за 70 мс (5000 миль в один конец; если бы сигнал шел со скоростью света, ему бы потребовалось 27 мс). В неудачный — программа выходит по тайм-ауту, прождав 1000 мс. Измерение скорости копирования большого файла показывает, что по сети 10 Мбит Ethernet передается около 1 Мбайт/с (то есть используется 80% полосы пропускания). Аналогично, 100 Мбит Ethernet позволяет переслать 10 Мбайт/с.

Немного поэкспериментировав, вы получите данные обо всех ключевых параметрах задачи. Разработчики баз данных должны знать время помещения записи в базу и считывания ее оттуда. Имеющим дело с графикой полагается знать стоимость основных операций с экраном. Время, затраченное на экспериментирование, окупится разумностью принятых решений.

7.3. Запас прочности

Результаты любых вычислений надежны ровно настолько, насколько надежны исходные данные. Хорошие исходные данные могут дать точные результаты даже при выполнении быстрых прикидок, и эти результаты могут оказаться полезными. Дональд Кнут однажды написал программу сортировки файла на диске и обнаружил, что она работает вдвое медленнее, чем предсказывали его расчеты. Кропотливая проверка показала, что из-за ошибки в программном обеспечении жесткие диски его компьютера работали вдвое медленнее заявленной производителем скорости (на протяжении года с момента установки). После исправления ошибки пакет Кнута стал работать с ожидаемой скоростью, а все другие программы, обращавшиеся к диску, тоже стали выполняться быстрее.

Однако часто ненадежными данными тоже можно пользоваться, чтобы получить ответ по порядку величины (опросник в приложении 2 поможет вам оценить качество своих догадок). Если вы предполагаете, что достоверность такого-то результата 20%, а другого — 50%, и при этом оказывается, что в любом случае результат отличается от спецификации в 100 раз в большую или меньшую сторону, повышать точность нет необходимости. Но не стоит так уж полагаться на результат, полученный с 20% точностью. Подумайте о том, что мне довелось несколько раз слышать от Вика Высоцкого (Vic Vyssotsky).

«Многие из вас, должно быть, помнят фотографии моста "Galloping Gertie", который разорвался на части во время бури в 1940 году. Висячие мосты разрывались задолго до того, это случается с ними и в наши дни. Объясняется это феноменом аэродинамической подъемной силы, и инженерам для правильных расчетов существенно нелинейных действующих сил, требуется использовать математику и концепции Колмогорова. До 50-х не было человека, который мог бы сделать это правильно. Почему же, в отличие от всех прочих, не разрывается Бруклинский мост?

Потому что Джон Роблинг (John Roebling) знал достаточно, чтобы отдавать себе отчет в том, чего он не знает. Его заметки и письма, касающиеся постройки Бруклинского моста, сохранились до наших дней, и они являются отличным примером того, как хороший инженер поступает, когда он сознает пределы своей осведомленности. Джон Роблинг сделал мост в шесть раз прочнее, чем требовалось из всех расчетов, основывавшихся на известных данных. Кроме того, он добавил сеть диагональных опор для упрочения моста. Взгляните на них, когда у вас будет такая возможность; они уникальны.

Когда Роблинга спросили, не рухнет ли его мост, как многие до того, он ответил, что этого не случится, потому что мост сделан с шестикратным запасом прочности.

Роблинг был хорошим инженером, и он построил хороший мост с шестикратным запасом прочности. Поступаем ли мы таким же образом? Я предлагаю уменьшать предполагаемую производительность систем реального времени в шесть раз в оценочных расчетах, чтобы учесть все то, чего мы можем не знать. В утверждениях о надежности и доступности нам следует уменьшать наши ожидания в десять раз, чтобы скомпенсировать недостаток наших знаний. Нам следует строить так, как Джон Роблинг, а не так, как его современники — ни один из их мостов не устоял до наших дней, причем четверть из них рухнули в течение первых десяти лет после завершения строительства.

Можем ли мы считать себя инженерами, подобными Роблингу?»

7.4. Закон Литтла

Большая часть вычислений при предварительных оценках основываются на очевидных правилах: полная стоимость равна стоимости одного объекта, умноженной на количество объектов. Иногда требуется использовать более тонкие соотношения. Брюс Вейд (Bruce Weide) пишет об одном удивительно универсальном законе следующее:

«*Операционный анализ, введенный Деннингом и Бузеном (Denning, Buzen, Computing Surveys, 10, 3, November 1978, 225–261), может использоваться очень широко, а не только в моделях очередей в компьютерных сетях. Авторы очень хорошо объясняют свой метод, но из-за ограниченности целей статьи они не раскрыли общность закона Литтла. Методы проверки никак не связаны с компьютерными сетями и вообще с компьютерами. Представьте себе любую систему, обменивающуюся объектами с внешней средой. Закон Литтла гласит, что "среднее количество объектов в системе равно произведению средней скорости ухода объектов из системы на среднее время, проводимое каждым из них в системе". (Если входящий поток равен исходящему, то скорость ухода объектов равна скорости их прихода.)*

Я преподаю этот метод анализа производительности в Университете штата Огайо. При этом я всегда пытаюсь особо подчеркнуть тот факт, что этот закон имеет гораздо более широкую область применения и не ограничен компьютерными системами. Например, если вы стоите в очереди в ночной клуб, можно попытаться определить время ожидания, понаблюдав некоторое время и оценив скорость, с которой люди заходят в двери клуба. Можно, однако, воспользоваться законом Литтла: "В этом зале помещается примерно 60 человек, причем каждый из них проводит там в среднем 3 часа, так что скорость, с которой мы туда заходим, — 20 человек в час. В очереди стоит 20 человек, так что через час я буду внутри. Лучше пойти домой и почитать «Жемчужины программирования», это будет полезнее". Так что вы понимаете, насколько этот закон может быть полезен».

Петер Деннинг (Petter Denning) кратко формулирует этот закон следующим образом: «*Среднее количество объектов в очереди равно произведению скорости входа и среднего времени ожидания*». Применяя его к винному погребу, Деннинг получает вот что: «*У меня есть 150 ящиков с вином, при этом я покупаю и выпиваю 25 ящиков в году. Сколько времени каждый из них хранится в моем погребе? Закон Литтла говорит, что нужно разделить 150 ящиков на 25 ящиков в год, что дает 6 лет*».

Затем он обращается к более серьезным задачам. «*Время отклика в многопользовательской системе может быть проверено с помощью закона Литтла и равенства входного и выходного потоков. Пусть* n *пользователей со средним временем размышления* z *подключены к некоторой системе с временем отклика* r. *Каждый пользователь постоянно переключается между режимами размышления и ожидания отклика системы, поэтому полное количество задач в метасистеме (состоящей из пользователей и компьютерной системы) постоянно и равно* n. *Если рассматривать вывод компьютерной системы и пользователей, мы получим мета-систему со средней загрузкой* n, *средним временем отклика* z+r *и пропускной способностью* x *(измеряемой в задачах в единицу времени). Закон Литтла утверждает, что* n = x × (z+r), *откуда* r=n/x – z».

7.5. Принципы

Делая предварительные оценки, всегда пользуйтесь известным советом Эйнштейна:

СОВЕТ

Alles sollte so einfach wie moglich gemacht werden, aber nicht einfacher. (Все следует сделать простым, насколько это возможно, но не проще.)

Мы обеспечиваем надежность быстрых прикидок, добавляя в систему запас прочности, призванный компенсировать наши ошибки в оценке параметров и неполноту знания задачи.

7.6. Задачи

Опросник в приложении 2 содержит дополнительные задачи на эту тему.

1. Здания Bell Labs расположены в 1000 миль от реки Миссисипи, но зато от них всего несколько миль до спокойной Пассаик (Passaic River). Однажды целую неделю шли сильные дожди, и в номере газеты Star-Ledger от 10 июня 1992 года появилось интервью с одним из инженеров, который сказал следующую фразу: «Вода в реке текла со скоростью около 200 миль в час, в пять раз быстрее, чем обычно». Есть комментарии?
2. На каких расстояниях курьер на велосипеде со сменным носителем данных в сумке обеспечит более высокую скорость передачи, чем высокоскоростная линия связи?
3. Сколько времени понадобится на то, чтобы заполнить дискету текстом, вводя его с клавиатуры?
4. Предположим, все процессы в мире замедлились в миллион раз. Сколько времени понадобится вашему компьютеру на выполнение одной команды? За сколько времени жесткий диск совершит один оборот? Сколько времени понадобится головкам этого диска, чтобы найти нужную дорожку? А как быстро вы введете свое имя с клавиатуры?
5. Докажите, что «отбрасывание девяток» действительно дает возможность проверять сложение. Как можно проверить «правило 72»? В какой формулировке его можно доказать?

6. Специальная комиссия США дает следующие сведения на 1998 год: население Земли — 5,9 миллиарда человек, годовой прирост — 1,33%. Если бы такой прирост сохранялся и дальше, сколько человек жило бы на Земле в 2050 году?
7. В приложении 3 описаны программы, дающие возможность определить стоимость структур данных и элементарных операций в вашей системе. Прочитав описание моделей, выпишите на бумагу свои предположения о возможных результатах для вашей системы. Скачайте программы с сайта книги, запустите их и сравните результаты со своими предположениями.
8. Используйте быстрые вычисления для оценки времени работы программ, схемы которых приведены в этой книге.
   1) Оцените требования к памяти и скорость работы программ.
   2) «О-большое» может рассматриваться как формализация быстрых вычислений — оно учитывает скорость роста функции, но отбрасывает постоянные множители. Используйте данные о времени работы алгоритмов в главах 6, 8, 11, 12, 13, 14 и 15 для оценки скорости программ, реализующих эти алгоритмы. Сравните свои оценки с данными экспериментов, которые привожу я.
9. Предположим, что системе требуется 100 обращений к диску для проведения одной транзакции (в некоторых системах их нужно меньше, а в некоторых — больше). Сколько транзакций в час может обработать дисковая подсистема?
10. Оцените смертность в вашем городе в процентах от населения в год.
11. [P. J. Denning] Придумайте схему доказательства закона Литтла.
12. Вы прочитали в газетной статье, что среднее время жизни монеты в четверть доллара в США — 30 лет. Как можно проверить это утверждение?

7.7. Дополнительная литература

Моя самая любимая книга о здравом смысле в математике — Даррелл Хафф «Как обманывать при помощи статистики» (Darrell Huff, How To Lie With Statistics, 1954). Примеры уже устарели, но принципы бессмертны. Более современный подход к тем же проблемам изложен в книге Джона Аллена Паулоса «Математическая безграмотность и ее последствия» (John Allen Paulos, Innumeracy: Mathematical Illiteracy and Ins Consequences, Farrar, Straus and Giroux, 1990).

Физики хорошо знакомы с предметом обсуждения этой главы. После выхода моей статьи в Communications of the ACM Ян Волицкий (Jan Wolitzky) написал мне:

«Мне частенько приходилось слышать, что предварительные оценки называли "приближенными оценками Ферми" по имени знаменитого физика. История гласит, что Энрико Ферми, Роберт Оппенгеймер и другие члены Манхэттенского проекта ждали взрыва первой атомной бомбы за низкой стеной в нескольких километрах от эпицентра. Ферми нарвал множество маленьких листочков бумаги, которые он подбросил в воздух, увидев вспышку взрыва. Когда ударная волна миновала их, он

измерил расстояние, на которое были отброшены бумажки, и быстро рассчитал мощность взрыва атомной бомбы, которая была позже точно подтверждена дорогостоящим измерительным оборудованием».

Информацию на эту тему можно найти в Интернете по следующим запросам: «предварительные оценки», «задачи Ферми».

7.8. Быстрые вычисления в повседневной жизни (дополнение)

В ответ на публикацию в Communications of the ACM я получил много интересных писем. Один из читателей рассказывал о рекламном объявлении, в котором говорилось, что коммивояжер проехал на новой машине 100 000 миль за год. В году 2000 рабочих часов (50 недель по 40 часов в неделе), а средняя скорость движения коммивояжера составляет около 50 миль в час; при этом не учитывается время, потраченное на беседы с покупателями, но произведение этих цифр уже дает величину, указанную в объявлении. Таким образом, это утверждение почти выходит за границы достоверности.

В повседневной жизни нам предоставляется множество возможностей для оттачивания способностей к быстрым вычислениям. Сколько денег вы потратили в прошлом году на рестораны? Однажды я с ужасом услышал, как один из жителей Нью-Йорка подсчитал, что они с женой за месяц тратят на такси больше, чем платят за квартиру. А для читателей из Калифорнии, которые, возможно, не знают, что такое такси, есть другая задачка: за какое время можно наполнить плавательный бассейн из шланга для поливки газона?

Некоторые читатели отмечали, что лучше всего предварительным оценкам можно научиться в детстве. Роджер Пинкхэм (Roger Pinkham) пишет:

«Я учитель и много лет пытался учить всех, кто готов был слушать, искусству выполнять приближенные оценки. Удивительно, насколько тщетными оказались мои попытки. Кажется, для этого требуется особый, критический стиль мышления.

Мой отец вбил это в меня. Я жил на берегу Майна и однажды, будучи еще маленьким ребенком, подслушал разговор между отцом и его другом, Гомером Поттером. Тот говорил, что две леди из штата Коннектикут вылавливали 200 фунтов омаров в день. Отец ответил: "Проверим. Пусть сеть вынимается каждые 15 минут и в нее в среднем попадается 3 омара. За час их будет 12, а за день — около 100. Я в это не верю!"

"Так или иначе, но это правда! — возмутился Гомер. — Ты вечно ни во что не веришь!"

Отец все равно не верил, и на этом все и закончилось. Через две недели Гомер сказал: "Помнишь тех двух леди, Фред? Они вылавливают 20 фунтов в день".

Отец проворчал: "Вот теперь я верю"».

Некоторые читатели рассматривали вопрос обучения детей оценочным расчетам с разных точек зрения. Наиболее популярными были вопросы типа: «За сколько ты бы дошел пешком до города Вашингтон?» и «Сколько листьев мы сгребли в кучи в этом году?». Подобные вопросы, если их правильно задавать, развивают любопытство в детях и учат их сомневаться и задумываться над разными вещами. С другой стороны, дети на некоторое время успокаиваются, пытаясь найти ответ, — это тоже полезно.

Методы разработки алгоритмов

В главе 2 данной книги рассказывалось о том, как изучение алгоритмов помогает сделать программы проще. Упрощение программы — это обычное следствие применения теории алгоритмов. В этой главе речь пойдет о более редком и более сильном влиянии теории алгоритмов на программы: сложные алгоритмы зачастую дают огромное увеличение производительности.

Здесь мы рассмотрим четыре алгоритма решения одной небольшой задачи, причем основной упор будет сделан на методы разработки программ. Некоторые из алгоритмов могут показаться слишком сложными, но их сложность окупается. Первой из рассматриваемых программ требуется 15 дней на решение задачи характерным размером 100 000 шагов (в дальнейшем характерное число шагов, за которое выполняется задача, мы будем называть просто *размером*), тогда как последний вариант программы решает ту же задачу за 5 миллисекунд.

8.1. Задача и простой алгоритм

Задача эта появляется при распознавании одномерного шаблона; к ее истории мы обратимся позже. На входе имеется массив x из n вещественных чисел, на выходе должна быть получена максимальная сумма любой непрерывной последовательности элементов массива. Например, если во входном массиве содержатся следующие 10 элементов (рис. 8.1):

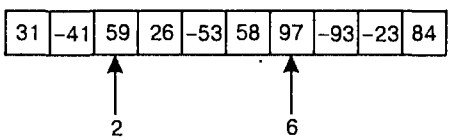


Рис. 8.1. Пример входного массива из 10 элементов

программа должна вернуть сумму x[2..6], то есть 187. Задачу легко решить с положительными числами — максимальная сумма будет равна сумме всех элементов

массива. Проблема возникает, когда появляются отрицательные числа: стоит ли включать такой элемент в выбранную последовательность, надеясь, что соседние компенсируют его? Чтобы завершить постановку задачи, скажем, что когда все элементы массива отрицательны, на выходе должна быть сумма элементов пустой последовательности элементов массива, равная нулю.

Первый вариант программы, который приходит в голову, перебирает все пары целых i и j, где 0<=i<=j<n; для каждой пары чисел вычисляется сумма x[i..j], после чего проверяется, превосходит ли она предыдущее найденное максимальное значение. Псевдокод алгоритма 1 приведен в листинге 8.1.

Листинг 8.1. Первый алгоритм решения задачи о максимальной подпоследовательности

```
maxsofar = 0
for i = [0, n)
  for j = [i, n)
    sum = 0
    for k = [i, j]
      sum += x[k]
    /* sum - сумма элементов x[i..j] */
    maxsofar = max(maxsofar, sum)
```

Программа короткая, ясная, ее легко понять. К сожалению, она работает медленно. На моем компьютере ей требуется 22 минуты, если n=10 000, и 15 дней, если n=100 000. Вопросы, связанные с временем выполнения, будут детально рассмотрены далее в разделе 8.5.

Конкретные данные о времени работы программы хорошо приводить в байках, но почувствовать реальную эффективность алгоритма можно, используя оценку эффективности с помощью понятия «О-большое», описанного в разделе 6.1. Внешний цикл выполняется ровно n раз, средний — не более n раз на каждый из проходов внешнего цикла. Перемножив эти величины, получим, что код внутри среднего цикла выполняется $O(n^2)$ раз. Внутренний цикл выполняется не более n раз, поэтому его мы тоже записываем как $O(n)$. Перемножив все эти величины, получим, что время работы алгоритма пропорционально n^3. Такие алгоритмы называются *кубическими*.

Этот пример иллюстрирует метод анализа с помощью «О-большого», а также его сильные и слабые стороны. Главный недостаток его в том, что мы все равно не знаем, сколько будет работать программа при конкретных входных данных, а знаем лишь, что количество шагов будет $O(n^3)$. Этот недостаток обычно компенсируется двумя сильными сторонами метода: анализ с помощью «О-большого» легко производить (как в примере выше), а информация об асимптотическом поведении алгоритма обычно достаточна для предварительных оценок эффективности.

В следующих разделах асимптотическое время работы алгоритма используется в качестве единственной оценки производительности программы. Если этого вам мало, обращайтесь к разделу 8.5 данной главы, в котором проиллюстрирована высокая точность анализа для данной задачи. Прежде чем читать дальше, задумайтесь на минуту и попробуйте найти более быстрый алгоритм.

8.2. Два квадратичных алгоритма

Большинство программистов реагируют на алгоритм 1 одинаково: «есть очевидный способ сделать его намного быстрее». Таких способов на самом деле два, при-

чем конкретному программисту обычно очевиден лишь один из двух. Оба алгоритма обладают квадратичным временем работы — делают $O(n^2)$ операций для массива размером n — и в обоих сумма x[i..j] вычисляется за постоянное количество шагов, а не за j-i+1 сложений, как в алгоритме 1. Однако эти два алгоритма используют принципиально различные методы вычисления сумм за конечное число шагов.

Первый квадратичный алгоритм позволяет быстро вычислить сумму благодаря тому, что сумма x[i..j] легко получается из предыдущей: x[i..j-1]. Использование этого свойства позволяет получить алгоритм 2а (листинг 8.2).

Листинг 8.2. Второй алгоритм (а) решения задачи о максимальной подпоследовательности

```
maxsofar = 0
for i = [0, n)
  sum = 0
  for j = [i, n)
    sum += x[j]
    /* sum - сумма x[i..j] */
    maxsofar = max(maxsofar, sum)
```

Операторы внутри внешнего цикла выполняются n раз, а те, что лежат внутри внутреннего, — не более чем n раз, так что время работы алгоритма составляет $O(n^2)$.

Альтернативный квадратичный алгоритм вычисляет сумму во внутреннем цикле, обращаясь к структуре данных, которая строится отдельно, до начала первого цикла. Создается массив cumarr, i-й элемент которого содержит кумулятивную сумму значений x[0..i], поэтому сумму x[i..j] можно получить как разность cumarr[j] – cumarr[i-1]. В итоге получим алгоритм 2б (листинг 8.3).

Листинг 8.3. Второй алгоритм (б) решения задачи о максимальной подпоследовательности

```
cumarr[-1] = 0
for i = [0, n)
  cumarr[i] = cumarr[i-1] + x[i]
maxsofar = 0
for i = [0, n)
  for j = [i, n)
    sum = cumarr[j] - cumarr[i-1]
    /* sum - сумма x[i..j] */
    maxsofar = max(maxsofar, sum)
```

(В задаче 5 рассматривается проблема доступа к cumarr[-1].) Этот код требует $O(n^2)$ операций; анализ проводится так же, как и для алгоритма 2а.

Рассмотренные три варианта алгоритмов исследуют все возможные пары начального и конечного индексов, вычисляя сумму элементов последовательности. Поскольку таких последовательностей $O(n^2)$, ни один алгоритм, проверяющий их все, не может быть быстрее квадратичного. Можете ли вы придумать способ обойти проблему и получить более быстрый алгоритм?

8.3. Алгоритм «разделяй и властвуй»

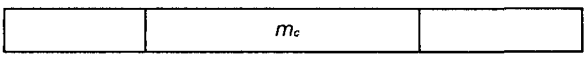
Первый из быстрых алгоритмов достаточно сложен, поэтому если вы запутаетесь в деталях, можете перейти сразу к следующему разделу, вы не слишком много потеряете при этом. Итак, алгоритм основан на следующем правиле:

Если нужно решить задачу размера n, следует рекурсивно решить две подзадачи размера приблизительно n/2, а затем объединить их решения в одно целое.

В данном случае мы имеем дело с массивом размера n, поэтому наиболее естественным способом разделения задачи на две подзадачи будет создание двух массивов приблизительно одинакового размера. Один из них мы назовем a, а другой — b.

Затем мы рекурсивно найдем подпоследовательности с максимальной суммой элементов в массивах a и b; одну из них назовем m_a, а другую — m_b.

Соблазнительная мысль о том, что мы решили задачу, потому что подпоследовательность всего массива с максимальной суммой должна равняться либо m_a, либо m_b, оказывается неправильной, потому что эта искомая подпоследовательность может и пересекать границу между a и b. Такую подпоследовательность мы назовем m_c, поскольку она пересекает границу.



Итак, наш алгоритм «разделяй и властвуй» должен рекурсивно находить m_a и m_b, после чего каким-то другим способом находить m_c и сравнивать их между собой.

Этого описания почти достаточно для написания программы. Осталось лишь определить способ работы с массивами малого размера и способ вычисления m_c. Первое сделать несложно: подпоследовательность с максимальной суммой для массива из одного элемента равна этому элементу, если он положителен, или нулю, если элемент отрицателен, а максимальная подпоследовательность пустого массива равна нулю. Для вычисления m_c отметим, что его левая часть представляет собой максимальную подпоследовательность элементов, заканчивающуюся на границе a и b и простирающуюся в a, а правая часть — такую же подпоследовательность, лежащую в b. Собрав все это вместе, получим алгоритм 3 (листинг 8.4).

Листинг 8.4. Третий алгоритм решения задачи о максимальной подпоследовательности

```
float maxsum3(l, u)
  if (l > u) /* пустой массив */
    return 0
  if (l == u) /* один элемент */
    return max(0, x[l])
  m = (l + u) / 2
  /* m  поиск максим  посл. слева от границы */
  lmax = sum = 0
  for (i = m; i >= l; i--)
    sum += x[i]
    lmax = max(lmax, sum)
  /* m  поиск максим  посл  справа от границы */
  rmax = sum = 0
  for i = (m, u]
```

```
    sum += x[i]
    rmax = max(rmax, sum)
  return max(lmax+rmax, maxsum3(l, m), maxsum3(m+1, u))
```

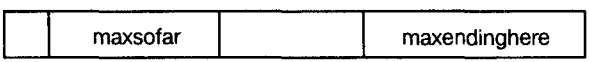
Алгоритм 3 запускается вызовом

```
answer = maxsum3(0, n-1)
```

Код достаточно сложен, в нем легко ошибиться, но он решает задачу за $O(n \log n)$ операций. Мы можем доказать это несколькими способами. Неформальное доказательство заключается в том, что алгоритм выполняет работу $O(n)$ на каждом из $O(\log n)$ уровней рекурсии. Доказательство можно сделать более строгим, используя рекуррентные соотношения. Если $T(n)$ обозначает время решения задачи размера n, то $T(1) = O(1)$ и $T(n) = 2T(n/2) + O(n)$. В задаче 15 показано, что решение этого рекуррентного соотношения: $T(n) = O(n \log n)$.

8.4. Сканирующий алгоритм

Воспользуемся простейшим алгоритмом для работы с массивами. Начинать следует с левого конца массива (элемента x[0]), затем нужно перебрать все элементы и закончить на правом конце (элемент x[n-1]), все время сохраняя информацию о наилучшей на данный момент сумме. Изначально максимальная сумма равна нулю. Предположим, что мы решили задачу для x[0..i-1], как можно расширить ее решение, добавив в него элемент x[i]? Используем те же соображения, что и для алгоритма «разделяй и властвуй»: подпоследовательность первых i элементов с максимальной суммой может либо целиком лежать в первых i элементах (хранится в maxsofar), либо заканчиваться элементом i (хранится в maxendinghere).



Вычисление maxendinghere каждый раз заново, аналогично алгоритму 3, даст нам в итоге еще один квадратичный алгоритм. Мы можем обойти это, используя тот же метод, который привел нас к алгоритму 2: вместо того, чтобы находить максимальную подпоследовательность, заканчивающуюся элементом i, мы воспользуемся максимальной подпоследовательностью, заканчивавшейся элементом i-1. В итоге получаем алгоритм 4 (листинг 8.5).

Листинг 8.5. Четвертый алгоритм решения задачи о максимальной подпоследовательности

```
maxsofar = 0
maxendinghere = 0
for i = [0, n)
  /* инвариант: значения maxendinghere и maxsofar точны для x[0..i-1] */
  maxendinghere = max(maxendinghere + x[i], 0)
  maxsofar = max(maxsofar, maxendinghere)
```

Ключ к пониманию этой программы — переменная maxendinghere. Перед первым оператором присваивания в цикле значение переменной равно максимальной сумме подпоследовательностей, заканчивающихся элементом i-1. Оператор присваивания изменяет ее содержимое таким образом, что оно становится равным максимальной сумме подпоследовательностей, заканчивающихся элементом i. Оператор увеличивает это значение добавлением x[i] до тех пор, пока это действие

оставляет сумму подпоследовательности положительной; возможные отрицательные значения заменяются нулем, поскольку максимальная по сумме подпоследовательность, заканчивающаяся элементом i, теперь является пустой. Хотя этот код труден для понимания, он прост и быстр, потому что выполняется за $O(n)$ операций. Такие алгоритмы называются *линейными*.

8.5. И что это значит?

Пока что мы действовали довольно безответственно, используя «О-большое». Пришло время поговорить о реальных вещах и измерить время выполнения программ. Я реализовал четыре приведенных выше алгоритма на языке C на компьютере Pentium II 400 МГц, измерил скорость их работы и экстраполировал данные, сведя их в табл. 8.1. Время выполнения алгоритма 2б обычно отличалось не более чем на 10% от алгоритма 2а, поэтому я не стал выделять его в отдельный столбец.

Таблица 8.1. Скорость работы алгоритмов

Алгоритм		1	2	3	4
Время выполнения в наносекундах		$1{,}3n^3$	$10n^2$	$47n\log_2 n$	$48n$
Время решения задачи размером	10^3	1,3 с	10 мс	0,4 мс	0,05 мс
	10^4	22 мин	1 с	6 мс	0,5 мс
	10^5	15 дней	1,7 мин	78 мс	5 мс
	10^6	41 год	2,8 ч	0,94 с	48 мс
	10^7	41 тыс. лет	1,7 недели	11 с	0,48 с
Максимальный размер задачи, решаемой за	1 с	920	10 000	$1{,}0\times10^6$	$2{,}1\times10^7$
	1 мин	3600	77 000	$4{,}9\times10^7$	$1{,}3\times10^9$
	1 ч	14 000	$6{,}0\times10^5$	$2{,}4\times10^9$	$7{,}6\times10^{10}$
	1 день	41 000	$2{,}9\times10^6$	$5{,}0\times10^{10}$	$1{,}8\times10^{12}$
При увеличении n в 10 раз время возрастает в		1000	100	10+	10
При увеличении времени в 10 раз n увеличивается в		2,15	3,16	10-	10

Эта таблица дает довольно много разнообразных сведений. Самый главный вывод: выбор правильного алгоритма может очень сильно влиять на время решения задачи. Это особенно подчеркивается в средней части таблицы. Последние две строки показывают, как связаны время решения задачи и ее размер.

Есть еще один важный момент. При сравнении кубического, квадратичного и линейного алгоритмов постоянные множители в выражениях для производительности значат не так уж много. Обсуждение алгоритма с временем выполнения $O(n!)$ в разделе 2.4 (см. главу 2) показывает, что для быстрорастущих функций постоянные множители значат еще меньше. Чтобы подчеркнуть важность этого момента, я поставил эксперимент, в котором разница в постоянных множителях была максимально возможной. Алгоритм 4 был реализован на компьютере Radio Shack TRS-80 Model III (бытовой компьютер 1980 года выпуска с процессором Z80 на частоте 2.03 МГц). Чтобы еще больше замедлить его работу, я использовал интерпретатор Бейсика, что замедляет программу в 10–100 раз по сравнению с компилированным кодом. Алгоритм 1 был реализован на компьютере Alpha 21164 с частотой 533 МГц. Время выполнения кубического алгоритма росло по формуле $0{,}58n^3$ нс,

а для линейного алгоритма — $19,5n$ мс (то есть $19\,500\,000n$ нс или 50 элементов в секунду). В табл. 8.2 показаны результаты экспериментов для различных размеров задачи.

Таблица 8.2. Кубический и линейный алгоритмы

N	Alpha 21164A, C, Кубический алгоритм	TRS-80, BASIC, Линейный алгоритм
10	0,6 мкс	200 мс
100	0,6 мс	2,0 с
1000	0,6 с	20 с
10 000	10 мин	3,2 мин
100 000	7 дней	32 мин
1 000 000[1]	19 лет	5,4 ч

[1] Это, скорее всего, результат расчета, а не эксперимента. — *Примеч. ред*

Отношение постоянных множителей 1/33 000 000 давало некоторое преимущество кубическому алгоритму при малых размерах задачи, но линейный алгоритм неизбежно должен был нагнать кубический при увеличении n. Оба алгоритма решают задачу за одинаковое время при n примерно равном 5800; при этом обоим компьютерам требуется около двух минут (рис. 8.2).

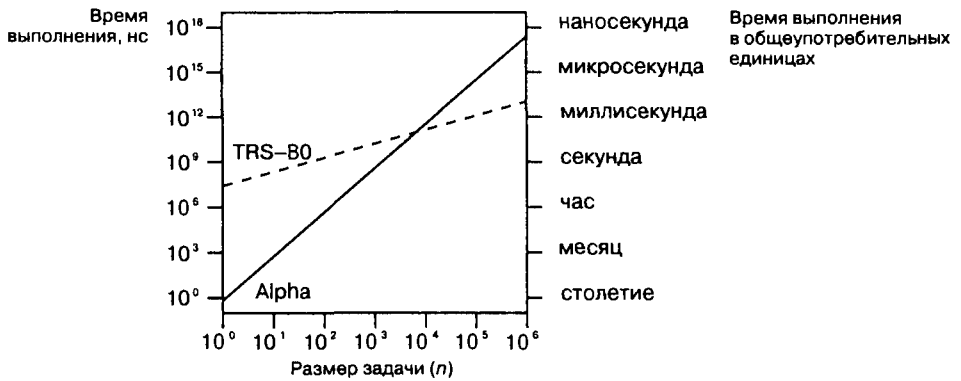


Рис. 8.2. Зависимость времени выполнения задачи от размера задачи для двух различных компьютеров, использующих разные алгоритмы

8.6. Принципы

История задачи проливает свет на методы разработки алгоритмов. Наша небольшая задачка возникла в рамках работы по распознаванию образов, которой занимался Ульф Гренандер (Ulf Grenander) в Университете Брауна. Исходная задача была двумерной, в этой форме она ставится в задаче 13 в конце главы. При этом подмножество с максимальной суммой элементов отражало наибольшее соответствие между двумя цифровыми изображениями. Поскольку в двумерном варианте решить задачу было слишком тяжело, Гренандер упростил ее и занялся вначале одномерной, чтобы вникнуть в суть дела.

Гренандер решил, что кубический алгоритм (первый) выполняется слишком медленно, и придумал алгоритм 2. В 1977 году он рассказал о задаче Майклу Ша-

мосу (Michael Shamos), который в тот же вечер придумал алгоритм 3. Когда Шамос вскоре после этого показал задачу мне, мы решили, что этот вариант алгоритма является лучшим из возможных, потому что незадолго до того исследователи показали, что несколько аналогичных задач могут быть решены только за время, пропорциональное $n \log n$. Спустя несколько дней Шамос рассказал о задаче и ее истории на семинаре Карнеги Меллона (Carnegie Mellon), где присутствовал статистик Джей Кадан (Jay Kadane), который и набросал алгоритм 4, затратив на это приблизительно минуту. К счастью, мы знаем, что никакой алгоритм не может работать быстрее этого последнего четвертого варианта. Любой правильный алгоритм решения такой задачи потребует $O(n)$ операций (см. задачу 6).

Хотя одномерная задача была решена полностью, исходная задача Гренандера в двумерной постановке остается открытой проблемой и сегодня, 20 лет спустя, когда второе издание этой книги выходит в печать. Из-за вычислительной трудоемкости всех известных алгоритмов Гренандеру пришлось отказаться от этого подхода к распознаванию образов. Читатели, которым кажется, что линейный алгоритм для одномерной задачи очевиден, могут попытаться придумать очевидный алгоритм для задачи 13.

Эта история иллюстрирует ценные методы разработки алгоритмов.

Сохранение данных во избежание повторных вычислений

Эта простая форма динамического программирования использовалась в алгоритмах 2 и 4. Используя память для сохранения результатов, мы исключаем необходимость повторного их вычисления.

Предварительная обработка данных и помещение их в структуры

Структура cumarr в алгоритме 2б позволяет быстро вычислить сумму подпоследовательности.

Алгоритмы «разделяй и властвуй»

Алгоритм 3 использует правило «разделяй и властвуй». В учебниках по теории алгоритмов описаны более сложные формы применения этого правила.

Сканирующие алгоритмы

Задачи с массивами часто могут быть решены с помощью вопроса «как можно расширить решение задачи с x[0..i-1] до x[0..i]?». Алгоритм 4 сохраняет предыдущий результат и некоторые дополнительные данные для вычисления следующего результата.

Кумулятивные суммы

Алгоритм 2б использует таблицу кумулятивных сумм, где i-й элемент содержит кумулятивную сумму первых i значений массива x. Такие таблицы часто исполь-

зуются при работе с диапазонами. Например, маркетологи вычисляют объем продаж с марта по октябрь, вычитая из годового объема продаж по октябрь годовой объем продаж по март.

Нижняя граница

Разработчики алгоритмов могут спокойно спать только тогда, когда они знают, что придуманный алгоритм — самый быстрый из возможных. Для этого им приходится доказывать существование нижней границы. Линейная скорость решения данной задачи является предметом обсуждения в задаче 6. Доказательство существования нижних границ может быть достаточно сложным.

8.7. Задачи

1. Алгоритмы 3 и 4 требуют аккуратного кодирования. Используйте методы верификации программ из главы 4 данной книги для доказательства правильности вашего кода. Аккуратно формулируйте инварианты.
2. Измерьте скорость работы четырех алгоритмов на вашем компьютере и постройте свою табл. 8.1.
3. Мы анализировали алгоритмы с помощью «О-большого». Попробуйте как можно более точно определить количество операций `max` в каждом из алгоритмов. Дает ли это возможность определить время работы программы? Сколько памяти требуется для работы каждого из алгоритмов?
4. Если элементы входного массива — случайные числа с равномерным распределением на отрезке `[-1, 1]`, каково ожидаемое значение подпоследовательности с максимальной суммой?
5. Для простоты записи в алгоритме 2б мы использовали элемент `cumarr[-1]`. Как бы вы реализовали это на C?
6. Докажите, что любой правильный алгоритм для вычисления максимальной суммы подпоследовательности должен проверять все `n` входных элементов. (Некоторые задачи допускают пропуск элементов. Изучите, например, алгоритм Сахе в решении 2.2 и алгоритм Боера-Мура поиска подстроки.)
7. Когда я занялся реализацией алгоритмов, я включил в тестовую программу сравнение результатов работы алгоритмов с результатом работы четвертого алгоритма. Программа сообщала, что в моих реализациях алгоритмов 2б и 3 были ошибки. Изучив значения ответов, я увидел, что они не были равны друг другу в точности, но были достаточно близки. В чем было дело?
8. Измените алгоритм 3 (алгоритм «разделяй и властвуй») так, чтобы он выполнялся с линейной скоростью даже в худшем случае.
9. Мы положили максимальную сумму массива из отрицательных чисел равной нулю. Предположим, что мы положили бы ее равной наибольшему элементу. Как бы вы изменили тексты программ?
10. Если бы нужно было найти подпоследовательность с наиболее близкой к нулю суммой, как бы вы это сделали? Каков был бы наиболее эффективный

алгоритм? Какие методы разработки алгоритмов можно было бы использовать для его написания? Как найти подпоследовательность с суммой, наиболее близкой к некоторому вещественному числу f?

11. Магистраль состоит из n-1 перегонов между n пунктами платежей за перевозку. За каждый перегон нужно платить свою сумму денег. Легко определить стоимость пути между любыми двумя станциями с помощью массива стоимостей за время $O(n)$ или за постоянное время с помощью таблицы с $O(n^2)$ записей. Опишите структуру данных, требующую $O(n)$ ячеек памяти, но позволяющую находить результат за постоянное время.
12. После инициализации массива x[0..n-1] нулями выполняется n следующих операций:

```
for i = [l, u]
  x[i] += v
```

где l, u и v — параметры каждой операции (l и u — целые числа, причем 0 <= l <= u < n, а v — вещественное число). После n операций выводятся значения элементов x[0..n-1]. Описанный алгоритм требует $O(n^2)$ операций. Можете ли вы сделать ту же задачу более производительной?

13. Имеется массив n × n вещественных чисел. В нем нужно найти прямоугольную подтаблицу с максимальной суммой элементов. Какова вычислительная сложность данной задачи?
14. Даны целые числа m и n и вещественный массив x[n], требуется найти целое i (0<=i<=n-m), такое, что сумма x[i]..x[i+m] максимально близка к нулю.
15. Найдите решение рекуррентного соотношения $T(n) = 2T(n/2) + cn$ при $T(1) = 0$, а n — степень двойки. Докажите результат по методу математической индукции. Что изменится, если $T(1) = c$?

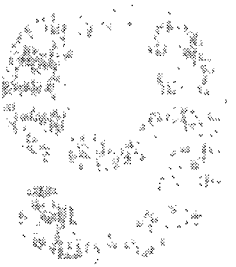
8.8. Дополнительная литература

Только глубокое знание теории и большой опыт дают возможность использовать любые методы разработки алгоритмов. Большинству программистов для их изучения нужно обратиться к учебнику по теории алгоритмов. «Структуры данных и алгоритмы» Аго, Хопкрофта и Ульмана (Aho, Hopcroft and Ullman, Data Structures and Algorithms, Addison Wesley, 1983) — подходящий учебник для студентов. Содержимое главы 10 очень близко к данной теме.

Книга Кормена, Лейзерсона и Ривеста «Введение в алгоритмы» (Cormen, Leiserson and Rivest, Introduction to Algorithms, MIT Press, 1990) объемом в 1000 страниц содержит подробное изложение данной темы. Части 1, 2 и 3 содержат основы теории алгоритмов, методы сортировки и поиска. Часть 4 особенно близка к теме данной главе. Части 5, 6 и 7 посвящены сложным структурам данных, алгоритмам на графах и другим избранным темам.

Эта и семь других книг вышли на компакт-диске Dr. Dobb's Essential Book on Algorithms and Data Structures. Диск был выпущен в 1999 году корпорацией Miller Freeman. Это бесценный справочник для любого программиста, интересующегося алгоритмами и структурами данных. На момент выхода моей книги в печать электронную версию компакт-диска можно заказать по адресу: http://www.ddj.com.

Оптимизация программ



Некоторые программисты слишком много внимания уделяют эффективности своих программ. Начиная слишком рано беспокоиться о небольших улучшениях, они создают слишком умные программы, которые потом очень сложно изменять. Другие, напротив, уделяют слишком мало внимания эффективности, поэтому их программы могут быть хорошо структурированы и красивы, но медленны и потому малополезны. Хорошие программисты думают об эффективности в контексте всего прочего — это всего лишь одна из проблем при разработке программного обеспечения, хотя и достаточно важная.

В предыдущих главах мы подошли к вопросам эффективности на высоком уровне: нас интересовала постановка задачи, структура системы, разработка алгоритма и выбор структуры данных. В этой главе мы рассмотрим низкоуровневый подход. Оптимизация программ заключается в обнаружении наиболее ресурсоемких частей существующей программы и последующем их изменении с целью увеличения производительности. Этот подход не всегда является наиболее разумным, и он не всегда эффектен, но иногда небольшие изменения в нужных местах позволяют существенно повлиять на производительность программы.

9.1. Типичная история

Однажды вечером Крис ван Вайк (Chris Van Wyk) и я беседовали об оптимизации программ. Затем он перешел к разговору об улучшении конкретной программы на языке C. Несколько часов спустя ему удалось вдвое сократить время выполнения программы из трех тысяч строк, работавшей с графикой.

Хотя время работы программы для большинства изображений было достаточно коротким, для обработки некоторых сложных картин требовалось около десяти минут. Ван Вайк начал с профилирования программы, что позволило ему определить, на выполнение каких функций уходит больше всего времени (профиль ана-

логичной программы меньшего объема показан в табл. 9.1). Запуск программы на десяти типичных тестовых изображениях показал, что почти 70% времени тратилось на функцию выделения памяти malloc.

Затем ван Вайк занялся изучением подсистемы выделения памяти. Поскольку его программа вызывала malloc из единственной функции, в которой была предусмотрена проверка ошибок, он имел возможность изменять эту функцию, а не исходный код библиотечной функции malloc. Он вставил несколько добавочных строк, которые помогли выяснить, что память под запись наиболее часто используемого типа выделялась приблизительно в 30 раз чаще, чем под запись следующего по частоте использования. Если вам известно, что большую часть времени программа тратит на выделение памяти под запись какого-либо одного известного типа, как бы вы изменили эту программу, чтобы ускорить ее работу?

Ван Вайк решил проблему, используя принцип кэширования: чаще используемые данные должны быть легко доступными. Он изменил свою программу, добавив в нее кэширование записей наиболее часто используемого типа в связном списке. После этого он мог быстро обрабатывать запросы, обращаясь к списку вместо того, чтобы вызывать подпрограмму выделения памяти. Это изменение позволило уменьшить время выполнения его программы на 45% (теперь на выделение памяти уходило только 30% времени работы программы). У новой программы было и еще одно преимущество: благодаря уменьшению фрагментации записей в памяти подсистема выделения работала быстрее, чем раньше. В решении задачи 2 этой главы показан альтернативный способ применения этого древнего метода; аналогичный подход будет использован несколько раз в главе 13.

Этот рассказ демонстрирует искусство оптимизации программ в его лучшем виде. Потратив несколько часов на измерения производительности и добавив двадцать строк к трем тысячам, Ван Вайк удвоил скорость работы программы, сохранив неизменными ее пользовательский интерфейс и легкость обслуживания. Для повышения производительности использовались стандартные средства: профилирование позволило определить наиболее ресурсоемкий участок кода, а кэширование уменьшило затраты в этом месте.

В табл. 9.1 приведен профиль типичной программы на языке C, аналогичный по форме и содержанию профилю программы Ван Вайка.

Таблица 9.1. Профиль программы на языке C

Func. time	%	Func+Child Time	%	Hit Count	Function
1413.406	52.8	1413.406	52.8	200002	malloc
474.441	17.7	2109.506	78.8	200180	insert
285.298	10.7	1635.065	61.1	250614	rinsert
174.205	6.5	2675.624	100.0	1	main
157.135	5.9	157.135	5.9	1	report
143.285	5.4	143.285	5.4	200180	bigrand
27.854	1.0	91.493	3.4	1	initbins

В этом примере бóльшая часть времени ушла на выполнение функции malloc. В задаче 2 в конце главы вам предлагается уменьшить время выполнения этой программы с помощью кэширования узлов.

9.2. Первая помощь: примеры

Перейдем от разбора большой программы к разбору нескольких небольших функций. Каждая функция — это задача, с которой мне приходилось неоднократно сталкиваться. Каждый раз бóльшая часть времени тратилась на выполнение этих функций и для устранения задержек использовались стандартные методы.

Деление с остатком

В разделе 2.3 (см. главу 2) предлагается три алгоритма циклического сдвига. В решении 2.3 реализуется алгоритм «с трюком», содержащий приведенную ниже операцию во внутреннем цикле:

```
k = (j + rotdist) % n;
```

Данные о затратах на простейшие операции из приложения 3 говорят нам, что оператор вычисления остатка от деления может быть очень ресурсоемким. Большинство арифметических операций выполняются на моем компьютере за время порядка 10 нс, тогда как деление с остатком требует около 100 нс. Время выполнения программы можно уменьшить, заменив приведенную выше команду на следующий текст:

```
k = j + rotdist
if (k >= n)
  k -= n;
```

Здесь мы заменяем «дорогостоящий» оператор % одной операцией сравнения и редко выполняющейся операцией вычитания. Но даст ли это выигрыш программе в целом?

В первом эксперименте программа выполнялась со сдвигом rotdist, равным единице. Время выполнения уменьшилось со $119n$ нс до $57n$ нс. Программа работала быстрее в два раза, причем ускорение на 62 нс было близким к предсказываемому теорией, исходя из затрат на операции.

В следующем эксперименте я сделал rotdist равным 10 и с удивлением обнаружил, что время выполнения обоих вариантов программы совпадало: $206n$ нс. Эксперименты, в результате которых был получен график в решении задачи 4 к главе 2, привели меня к следующей мысли: при rotdist = 1 алгоритм обращался к памяти последовательно и время тратилось на операцию деления с остатком. Для rotdist = 10 мы обращались к каждой десятой ячейке памяти и затраты на копирование данных из оперативной памяти в кэш становились более существенными.

Программисты прошлого твердо знали по опыту, что не было смысла пытаться улучшить вычислительную часть программы, которая бóльшую часть времени тратила на ввод и вывод данных. В современных архитектурах также бесполезно уменьшать затраты на вычисления, когда основное время уходит на доступ к памяти.

Функции, макросы и встраиваемый код

На протяжении всей главы 8 нам приходилось находить максимальное из двух чисел. Например, в разделе 8.4 мы вычисляли следующие выражения:

```
maxendinghere = max(maxendinghere, 0),
maxsofar = max(maxsofar, maxendinghere),
```

Функция max возвращает наибольший из двух аргументов:

```
float max(float a, float b)
{return a > b ? a : b; }
```

Время выполнения такой программы составляет около $89n$ нс.

Опытные программисты на C почувствуют рефлекторное желание заменить эту функцию макросом:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

Это некрасиво и может привести к возникновению ошибок. При использовании оптимизирующих компиляторов это вообще не даст никакого результата, поскольку такие компиляторы раскрывают небольшие функции (вставляют их текст в месте вызова). Однако в моей системе это изменение уменьшило время выполнения алгоритма 4 с $89n$ нс до $47n$ нс, то есть ускорило ее работу почти вдвое.

Восторг от такого ускорения пропал, когда я измерил эффект от такого же действия для алгоритма 3 (раздел 8.3 главы 8): для n = 10 000 время выполнения программы увеличилось с 10 мс до 100 с, то есть в 10 000 раз. Макрос увеличил зависимость времени выполнения алгоритма 3 от размера задачи с $O(n \log n)$ до некоторой нелинейной зависимости, близкой к $O(n^2)$. Вскоре я обнаружил, что семантика макросов приводила к тому, что алгоритм 3 рекурсивно вызывал сам себя большее число раз, чем это было необходимо, что привело к увеличению асимптотического времени выполнения. В задаче 4 приведен еще более яркий пример подобного замедления работы.

Программистам на C приходится выбирать между эффективностью и правильностью программ. На языке C++ мы вкушаем лучшие плоды обоих подходов. Язык C++ позволяет указать компилятору на необходимость раскрытия кода функции в явном виде в месте вызова. При этом совмещается ясная семантика функций и высокая производительность макросов.

Из любопытства я постарался избежать использования и макросов и функций, воспользовавшись операторами if:

```
if (maxendinghere < 0)
  maxendinghere = 0;
if (maxsofar < maxendinghere)
  maxsofar = maxendinghere.
```

Время выполнения программы не претерпело существенных изменений.

Последовательный поиск

Обратимся к последовательному поиску в таблице (которая может быть неотсортированной).

Листинг 9.1. Последовательный поиск (версия 1)

```
int ssearch1(t)
  for i = [0, n)
    if x[i] == t
```

```
      return i
  return -1
```

Эта короткая программа находит элемент в массиве x за $4{,}06n$ нс (в среднем). Поскольку в среднем приходится просматривать примерно половину элементов, на каждый из них приходится около 8,1 нс.

Цикл выглядит достаточно стройным, но жирок с него все-таки срезать можно. Во внутреннем цикле выполняется две проверки. Сначала проверяется, достиг ли индекс i конца массива, а затем выясняется, не равен ли элемент x[i] искомому. Первую проверку можно исключить, добавив элемент-метку в конце массива.

Листинг 9.2. Последовательный поиск (версия 2)

```
int ssearch2(t)
  hold = x[n]
  x[n] = t
  for (i = 0; ,i++)
    if x[i] == t
      break
  x[n] = hold
  if i == n
    return = -1
  else
    return i
```

Время работы уменьшилось до $3{,}87n$ нс, то есть быстродействие возросло на 5%. Мы предполагаем, что массив уже выделен, поэтому имеется возможность временно затереть значение x[n]. Нужно быть аккуратным при сохранении значения x[n] и его восстановлении. В большинстве приложений этого не требуется, так что мы исключим эту операцию из следующей версии программы.

Теперь внутренний цикл содержит только операцию увеличения индекса, обращение к элементу массива и проверку. Можно ли еще что-нибудь урезать? В последней версии нашей программы количество повторов цикла сокращается в 8 раз. Дальнейшее сокращение не ускорило работу программы.

Листинг 9.3. Последовательный поиск (версия 3)

```
int ssearch3(t)
  x[n] = t
  for (i = 0; ; i += 8)
    if (x[i] == t) { break }
    if (x[i+1] == t) { i += 1; break }
    if (x[i+2] == t) { i += 2; break }
    if (x[i+3] == t) { i += 3; break }
    if (x[i+4] == t) { i += 4; break }
    if (x[i+5] == t) { i += 5; break }
    if (x[i+6] == t) { i += 6; break }
    if (x[i+7] == t) { i += 7; break }
  if i == n
    return -1
  else
    return i
```

При этом время поиска сокращается до $1{,}07n$ нс, то есть наблюдается 56% ускорение. На старых компьютерах уменьшение дополнительных затрат может дать

ускорение работы на 10 или 20%. На современных машинах раскрытие цикла позволяет избежать остановки конвейера, уменьшить количество ветвей и увеличить параллелизм на уровне инструкций.

Вычисление расстояний на сфере

Конечная задача типична для приложений, в которых обрабатываются географические или геометрические данные. Входные данные состоят из двух частей. Первая часть: множество (set) S, состоящее из 5000 точек на поверхности сферы. Каждая точка определяется долготой и широтой. После помещения точек в некоторую структуру данных программа считывает вторую часть входных данных: последовательность из 20 000 точек (заданных долготой и широтой). Для каждой из точек последовательности требуется определить ближайшую к ней точку множества S. Расстояние измеряется по дуге большого круга (или как угловое расстояние между лучами, проведенными из центра сферы в заданные точки).

Маргарет Райт пришлось решать аналогичную задачу в Стенфордском университете в начале 80-х, когда она работала с картами, на которых представлялись данные о распределении некоторых генетических особенностей. Она решала задачу «в лоб», представив S в виде массива значений ширины и долготы. Ближайшая к данной точка находилась путем перебора всех точек и вычисления расстояния до них по сложной геометрической формуле с десятью синусами и косинусами. Программа была простой в написании и выдавала хорошие карты для небольших наборов данных, но для больших карт требовалось несколько часов работы большого компьютера, что выходило далеко за рамки бюджета проекта.

Мне случалось работать с геометрическими задачами и раньше, поэтому Райт попросила меня помочь ей в этом. Потратив на это воскресенье, я придумал несколько причудливых алгоритмов и структур данных для решения задачи. К счастью, как это теперь ясно, для каждого из них потребовалось бы написать много сотен строк кода, потому я не попытался их закодировать. Когда я описал структуры данных Эндрю Эппелю (Andrew Appel), ему пришла в голову замечательная идея. Вместо того чтобы решать проблему на уровне структур данных, следовало использовать простейшую структуру для хранения точек (массив), но заняться стоимостью вычисления расстояний между точками. Как бы вы воспользовались этой идеей?

Вычислительная стоимость могла быть уменьшена путем изменения представления данных о точках. Вместо того чтобы хранить их ширину и долготу, положение точки на поверхности глобуса можно описать ее координатами в декартовой системе (x, y, z). Поэтому новая структура данных представляла собой массив, в котором хранились как угловые, так и декартовы координаты всех точек. По мере обработки точек последовательности угловые координаты легко преобразовывались в декартовы с помощью нескольких тригонометрических функций, после чего вычислялись расстояния от них до точек из набора S. Расстояние вычислялось как сумма квадратов разностей соответствующих координат, что обычно быстрее, чем вычислять одну тригонометрическую функцию (и уж конечно, быстрее, чем вычислить 10 таких функций). Данные в приложении 3 подтверждают мои слова.

Ответ оказывается правильным, поскольку угловое расстояние между точками монотонно растет с ростом евклидового расстояния между ними.

Хотя этот подход требует дополнительной памяти для хранения данных, он дает значительный выигрыш во времени. Когда Райт внесла требуемые изменения в свою программу, время расчета сложных карт уменьшилось с нескольких часов до 30 секунд. В данном примере оптимизация кода позволила решить задачу в десятке дополнительных строк кода, тогда как изменение алгоритма и структуры данных потребовало бы многих сотен новых строк.

9.3. Оптимизируем двоичный поиск

Обратимся к одному из наиболее ярких примеров оптимизации программ. Мы возьмемся за задачу 8 из главы 4: требуется реализовать двоичный поиск в таблице из тысячи целых чисел. Помните, что обычно оптимизировать двоичный поиск не требуется — алгоритм и так достаточно эффективен. Именно поэтому мы игнорировали возможность микроскопического повышения эффективности в главе 4, когда нашей задачей стояло получение простой и верной программы, которую было бы легко сопровождать. Однако иногда оптимизация поиска оказывает существенное влияние на производительность большой системы в целом.

Разрабатывать быструю версию двоичного поиска мы будем постепенно, в четыре этапа. Промежуточные программы достаточно сложны, но существует веская причина изучить их внимательно: последняя версия программы работает в 2–3 раза быстрее, чем исходная из раздела 4.2 (глава 4). Прежде чем вы продолжите чтение, попробуйте найти в листинге 9.4 явные «излишки», от которых можно избавиться.

Листинг 9.4. Двоичный поиск (исходная программа)

```
l = 0; u = n-1
loop
  /* инвариант: если t есть в массиве, то оно лежит в диапазоне x[l..u]
*/
  if l > u
    p = -1; break
  m = (l + u) /2
  case
    x[m] < t: l = m+1
    x[m] == t: p = m; break
    x[m] > t: u = m-1
```

Мы начнем работу над оптимизацией двоичного поиска с изменения условия задачи. Пусть требуется найти *первое* вхождение t в массиве x[0..n-1] (приведенная в листинге 9.4 версия возвращает произвольное вхождение). Именно такое условие будет стоять в разделе 15.3 (глава 15). Основной цикл программы будет выглядеть так же, как и в листинге 9.4. Мы сохраним индексы l и u, ограничивающие область поиска элемента t, но инвариант будет другим: x[l] < t <= x[u] и l < u. Мы предположим, что n >= 0, x[-1] < t и x[n] >= t (но программа никогда не будет обращаться к фиктивным элементам). Новая версия двоичного поиска приведена в листинге 9.5.

Листинг 9.5. Двоичный поиск: возвращение первого вхождения

```
l = -1, u = n
while l+1 != u
  /* инвариант: x[l] < t && x[u] >= t && l < u */
  m = (l + u)/2
  if x[m]<t
    l = m
  else
    u = m
/* утверждение  l+1 = u && x[l] < t && x[u] >= t */
p = u
if p >= n || x[p] != t
  p = -1
```

В первой строке мы инициализируем инвариант. При повторных проходах тела цикла инвариант сохраняется при помощи оператора if: легко доказать, что обе ветви сохраняют инвариант. При завершении работы мы можем быть уверены, что если t есть в массиве, то его позиция возвращается в переменной u. Этот факт более формально выражен в *утверждении*. Последние две строки присваивают p индекс первого вхождения t в массив x или -1 в случае отсутствия t в массиве.

Хотя эта программа решает более сложную задачу, чем предыдущая, она потенциально более эффективна, поскольку на каждой итерации выполняется только одно сравнение t с элементом массива x. Предыдущей программе в некоторых случаях приходилось производить две такие проверки.

Следующая версия программы будет использовать известный нам факт, заключающийся в том, что n = 1000. Мы представим диапазон по-другому, используя нижнюю границу l и добавку i вместо нижней и верхней границ (l и u). Код будет гарантировать, что i всегда равно какой-либо степени двойки. Это свойство легко сохранить после того, как оно впервые окажется верным, но сделать его верным достаточно сложно (поскольку размер массива — 1000). Поэтому перед основным циклом программы нужно поставить условный оператор и оператор присваивания, которые сузят начальный диапазон до 512 (наибольшая степень двойки, меньшая 1000). Значения l и l+i позволяют задать диапазон -1..511 или 488..1000. Переделав программу, получим листинг 9.6.

Листинг 9.6. Двоичный поиск: новое представление границ диапазона

```
i = 512
l = -1
if x[511] < t
  l = 1000 - 512
while i != 1
  /* инвариант  x[l] < t && x[l+i] >= t && i = 2^j */
  nexti = i / 2
  if x[l+nexti] < t
    l = l + nexti
    i = nexti
  else
    i = nexti
/* утверждение. i == 1 && x[l] < t && x[l+i] >= t */
p = l+1
if p>1000 || x[p] != t
  p = -1
```

Доказательство правильности этой программы проводится так же, как и для предыдущей. Этот вариант обычно работает медленнее предшествующего, но он дает возможность дальнейшего повышения производительности.

Следующая программа является упрощением данной. Она учитывает возможность автоматической оптимизации, обеспечиваемой при компиляции. Упрощается первый оператор if, убирается переменная nexti, и присваивания, где она участвовала, удаляются из внутреннего оператора if.

Листинг 9.7. Двоичный поиск: упрощение предыдущей версии

```
i = 512
l = -1
if x[511] < t
  l = 1000 - 512
while i != 1
  /* инвариант  x[l] < t && x[l+i] >= t && i = 2^j */
  i = i/2
  if x[l+i] < t
    l = l+i
/* утверждение: i == 1 && x[l] < t && x[l+i] >= t */
p = l+1
if p>1000 || x[p] != t
  p = -1
```

Хотя корректность алгоритма доказывается тем же путем, что и раньше, мы уже лучше понимаем его работу на интуитивном уровне. Когда первая проверка оказывается неправильной и переменная l остается нулевой, программа находит биты p в порядке убывания значимости.

Последняя версия программы написана не от отчаяния. Она удаляет издержки на структуру цикла и деление i/2 путем раскрытия цикла в явном виде. Поскольку i принимает десять конкретных значений, мы можем закодировать их в нашей программе. Тогда нам не придется их вычислять во время ее выполнения.

Листинг 9.8. Двоичный поиск: последняя версия

```
l = -1
if (x[511] <t) l = 1000 - 512
  /* утверждение: x[l] <t && x[l+512] >= t */
if (x[l+256] < t) l += 256
  /* утверждение: x[l] < t && x[l+256] >= t */
if (x[l+128] < t) l += 128
if (x[l+64] < t) l += 64
if (x[l+32] < t) l += 32
if (x[l+16] < t) l += 16
if (x[l+8] < t) l += 8
if (x[l+4] < t) l += 4
if (x[l+2] < t) l += 2
  /* утверждение: x[l] < t && x[l+2] >= t */
if (x[l+1] < t) l += 1
  /* утверждение. x[l] < t && x[l+1] >= t */
p = l+1
if p>1000 || x[p] != t
  p = -1
```

Утверждения позволяют лучше понять работу программы. Проведя анализ одного оператора if и поняв, что происходит в обоих случаях, вы легко разберетесь и со всем остальным.

Я сравнивал «простой» двоичный поиск из раздела 4.2 (глава 4) с этой оптимизированной версией в разных системах. В первом издании этой книги приводилось время работы программ с различными уровнями оптимизации на четырех компьютерах и пяти языках программирования. Коэффициент ускорения менялся от 1,38 до 5 (что соответствует сокращению времени работы на 80%). Когда я измерил скорость работы исходной и оптимизированной программ на моем нынешнем компьютере, я с удивлением обнаружил, что время работы для `n = 1000` сократилось с 350 до 125 нс (сокращение времени работы на 64%).

Результат показался мне слишком хорошим, и вскоре обнаружилось, что он действительно не был вполне корректным. Тестовая программа осуществляла последовательный поиск элементов (сначала `x[0]`, затем `x[1]` и так далее), что давало подпрограмме двоичного поиска преимущество, поскольку допускало предсказание ветвления алгоритма. Изменив тестовую программу так, чтобы поиск элементов производился случайным образом, я получил новый результат: 418 нс для основной версии и 266 нс — для оптимизированной (36% сокращение времени).

Мы разобрали предельный случай оптимизации. Очевидная версия программы двоичного поиска (которая казалась достаточно стройной) была заменена оптимизированной, в которой нет уж совсем ничего лишнего. (Эта функция была написана впервые в начале 60-х. Я услышал о ней от Гая Стила (Guy Steele) в начале 80-х, а он изучал ее в MIT. Вик Высоцкий использовал этот код в Bell Labs в 1961 году. Каждый оператор if псевдокода можно было реализовать тремя командами IBM 7090.)

Средства верификации программ из главы 4 оказались в этой задаче жизненно необходимыми. Их использование обеспечило нам уверенность в правильности получившейся программы. Когда я в первый раз увидел последний вариант программы без утверждений и вывода, он показался мне чем-то сверхъестественным.

9.4. Принципы

Главный принцип оптимизации программ заключается в том, что оптимизировать их нужно как можно реже. Это обобщение объясняется следующими утверждениями.

Важность эффективности

Есть много других качеств программного обеспечения, не менее важных, чем эффективность. Дон Кнут считал досрочную оптимизацию программ корнем зла, потому что она может сделать программу неправильной, малофункциональной и затруднить ее сопровождение. Оставьте заботу об эффективности для тех случаев, когда в ней действительно возникнет необходимость.

Средства измерения

Когда эффективность становится важной, начинать следует с профилирования программы для определения участков с наибольшими временны́ми затратами.

Профилирование программы обычно показывает, что бо́льшая часть времени уходит на выполнение некоторых определенных действий, тогда как прочие команды выполняются редко (в разделе 6.1 единственная функция выполнялась 80% времени работы программы). Профилирование указывает места возможных улучшений, а что касается всех остальных мест — следуйте мудрому правилу «не чини того, что не сломано». Данные о стоимости отдельных операций, аналогичные приведенным в приложении 3, могут помочь программисту понять причины недостаточной эффективности его программы.

Уровни разработки

В главе 6 данной книги мы уже говорили о том, что к проблемам эффективности можно подходить на разных уровнях. До оптимизации следует подумать о других возможностях ускорить работу программы.

Когда вместо ускорения получается замедление

Замена оператора % оператором if иногда приводила к ускорению работы программы вдвое, а иногда никак не влияла на скорость работы программы. Замена функции макросом ускорила одну функцию вдвое и замедлила другую в 10 000 раз. После «улучшения» программы нужно испытать то, что получилось, на репрезентативных данных. Множество других примеров учит нас прислушиваться к предупреждению Юрга Нивергельта (Jurg Nievergelt), обращенному к оптимизаторам: «Тот, кто играет с битами, рискует быть битым[1]».

Итак, мы обсудили, стоит ли оптимизировать программу. Если мы все-таки решили заняться этим опасным делом, нам все еще нужно выбрать подходящее средство для этого. В приложении 4 вы найдете список общих рекомендаций по оптимизации программ. Все разобранные выше примеры иллюстрируют эти принципы.

Графическая программа Ван Вайка

Стратегия его решения заключается в *использовании общего*, а именно *кэшировании* наиболее часто используемых данных.

Деление с остатком

Решение использовало *алгебраическую эквивалентность*, позволяющую заменить дорогостоящую операцию нахождения остатка на более дешевое сравнение.

Функции, макросы и встраиваемый код

Устранение иерархии процедур заключается в замене функции на макрос. Это дает возможность ускорить код почти в 2 раза. Что касается явного раскрытия макросов, то оно результата не дает.

---

[1] People who play with bits should expect to get bitten — в английском языке используется игра слов (bit — укус или бит (binary digit); bitten — покусанный). — *Примеч. ред.*

Последовательный поиск

Использование маркеров (граничных элементов) для объединения проверок ускорило работу на 5%. *Раскрытие цикла* дало еще 56%.

Вычисление расстояний на сфере

Хранение декартовых координат вместе со сферическими является примером расширения структуры данных. Использование декартовых координат и евклидовой длины (более дешевое с вычислительной точки зрения) возможно благодаря *алгебраической эквивалентности*.

Двоичный поиск

Объединение проверок уменьшило количество сравнений во внутреннем цикле с двух до одного. *Алгебраическая эквивалентность* позволила изменить представление верхней и нижней границ, а *раскрытие цикла* устранило издержки на его структуру.

Пока что мы занимались только оптимизацией программ по времени выполнения. При необходимости можно оптимизировать их по другим параметрам, таким как обращение к файлу подкачки или увеличение количества обращений к кэшу. Помимо оптимизации по времени выполнения чаще всего приходится заниматься оптимизацией для уменьшения объема требуемой памяти. Эта тема и является предметом обсуждения в следующей главе.

9.5. Задачи

1. Профилируйте одну из собственных программ и воспользуйтесь описанными в этой главе методами для повышения ее быстродействия.
2. На сайте этой книги находится программа на языке C, профиль которой приведен в табл. 9.1. Она представляет собой небольшую часть программы из главы 13. Попробуйте осуществить ее профилирование в своей системе. Если только ваша функция malloc не обладает феноменальным быстродействием, бóльшую часть времени программа будет тратить именно на ее вызов. Попробуйте увеличить быстродействие программы тем же методом, что и Ван Вайк.
3. Какие свойства алгоритма циклического сдвига позволили заменить оператор % оператором if, а не более ресурсоемким оператором while? Определите, в каких случаях использование оператора while вместо % оправдано.
4. Для положительного целого n, не большего размера массива, следующая рекурсивная функция возвращает максимальный элемент массива:

```
float arrmax(int n)
{
  if (n == 1)
    return x[0]
```

```
  else
    return max(x[n-1], arrmax(n-1));
}
```

Если реализовать max как функцию, то максимальный элемент массива размерностью 10 000 будет найден за несколько миллисекунд. Если реализовать max как макрос вида

```
#define max(a, b) ((a) > (b) ? (a)   (b))
```

алгоритму потребуется 6 секунд на нахождение максимального из 27 элементов и 12 секунд на нахождение максимального из 28 элементов. Придумайте, при каких входных данных поведение программы снова станет нормальным. Проанализируйте время выполнения математически.

5. Как ведут себя различные алгоритмы двоичного поиска, если их применить к неотсортированным массивам?
6. Библиотеки C и C++ позволяют определить тип символа с помощью функций isdigit, isupper и islower. Как бы вы реализовали эти функции?
7. Имеется длинная последовательность байтов (например, миллиард). Каким образом можно быстро подсчитать количество единичных битов (то есть, не равных 0)?
8. Как можно использовать маркеры в программе, ищущей максимальный элемент массива?
9. Поскольку последовательный поиск проще двоичного, он более эффективен для небольших массивов. С другой стороны, растущее логарифмически количество сравнений обеспечивает преимущество двоичного поиска на больших объемах данных. Размер массива, для которого оба алгоритма работают за одинаковое время, зависит от оптимизации обеих программ. Как низко вы можете опустить эту границу и как высоко можно ее поднять? Чему равна эта граница на вашем компьютере, когда обе программы оптимизированы примерно одинаково?
10. Д. Б. Ломет (D. B. Lomet) обнаружил, что хэширование позволяет решить задачу о поиске целого числа среди тысячи более эффективно, чем оптимизированный двоичный поиск. Реализуйте несколько программ с кэшированием и сравните их быстродействие с двоичным поиском по быстродействию и занимаемой памяти.
11. В начале 60-х Вик Берец (Vic Berecz) обнаружил, что бóльшая часть времени работы моделирующей программы в фирме Сикорски уходила на вычисление тригонометрических функций. Дальнейшие исследования показали, что аргументы функций всегда принимали значения, кратные пяти градусам. Как ему удалось уменьшить время работы программы?
12. Иногда программу удается оптимизировать с помощью математического, а не программистского подхода. Для вычисления значения полинома

$$y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$$

нижеследующая программа делает $2n$ умножений. Придумайте функцию, которая будет делать это быстрее.

```
y = a[0]
xi = 1
for i = [1. n]
  xi = x * xi
  y = y + a[i]*xi
```

9.6. Дополнительная литература

В разделе 3.8 (глава 3) уже упоминалась книга Стива Макконнелла «Завершенный код». Глава 28 этой книги посвящена стратегии оптимизации программ. В ней дается обзор вопросов производительности и подробное описание подходов к оптимизации. В главе 29 собран отличный набор правил оптимизации.

Приложение 4 настоящей книги содержит подборку правил оптимизации вместе с описанием их применения.

Экономия памяти

Некоторые из моих знакомых, прочитав этот заголовок, скажут: «Проблема устарела!» В старые злые времена программисты, говорят, и вправду были ограничены рамками своих маленьких ЭВМ, но те дни давно прошли. Новая философия проста: «Гигабайт здесь, гигабайт там — памяти хватит на всех». И в этом подходе есть своя правда — многие программисты работают на больших компьютерах и им редко приходится беспокоиться о размерах своих программ.

Но частенько долгие размышления над уменьшением объема программы оказываются чрезвычайно благотворными. Иногда появляются новые идеи, помогающие упростить программу. Уменьшение объема может иметь положительные побочные эффекты, заключающиеся в сокращении времени выполнения программы: она быстрее грузится и лучше помещается в кэш-памяти. Меньший объем данных обычно требует меньше времени на обработку. Время передачи данных по сети обычно прямо пропорционально их объему. Несмотря на то что память стоит дешево, она все равно может являться серьезным ограничением. В небольших компьютерах, управляющих игрушками и бытовой техникой, памяти обычно немного. А на больших компьютерах обычно решают сложные задачи, так что и тут может возникнуть необходимость сокращать объем программы.

Итак, мы будем помнить о важности стоящей перед нами задачи. Теперь посмотрим, какие имеются методы для ее решения.

10.1. Ключ к успеху — простота

Простота программы обычно связана с ее функциональностью, устойчивостью, скоростью и объемом. Деннис Ритчи (Dennis Ritchie) и Кен Томпсон (Ken Thompson) разработали операционную систему Unix на компьютере с памятью объемом 8192 18-битных слов. В их статье о новой системе было сказано: «*На систему и программное обеспечение были наложены весьма жесткие ограничения.*

Учитывая противоречивые стремления к разумной эффективности и выразительности средств, ограничение на размер системы стимулировало не просто экономию памяти, но определенную элегантность и стройность схемы».

Фред Брукс почувствовал эффективность простых решений, когда писал программу расчета заработной платы для крупной компании в середине 50-х. Узким местом программы было представление данных о подоходном налоге в штате Кентукки. В законе тарифы определялись с помощью таблицы, в которой по вертикали откладывался доход, а по горизонтали — льготы. Хранение таблицы в явном виде требовало тысячи слов памяти, что превосходило возможности компьютера.

Сначала Брукс попробовал решить проблему, попытавшись подобрать математическую функцию, которая описывала бы зависимость тарифа от аргументов, но данные изменялись настолько неравномерно, что никакая простая функция тут бы не подошла. Зная, что таблица была составлена членами законодательного собрания, у которых не могло быть пристрастия к заумным математическим функциям, Брукс обратился за сведениями к одному из них, чтобы понять, каким именно образом была составлена эта хитрая таблица. Он обнаружил, что налог штата Кентукки представлял собой простую функцию дохода, остававшегося *после* отчисления федерального налога. Тогда он написал программу, которая вычисляла федеральный налог по имевшимся таблицам, а затем использовала оставшийся доход и таблицу из нескольких десятков машинных слов для вычисления требуемого налога штата Кентукки.

Изучив контекст возникновения задачи, Брукс смог заменить исходную задачу более простой. На решение исходного варианта требовалось несколько тысяч слов, тогда как упрощенная задача была решена с использованием минимального объема памяти.

Простота может уменьшить и объем кода. В главе 3 приведены примеры больших программ, которые были заменены маленькими с более подходящими структурами данных. В этих примерах упрощение программы позволило уменьшить объем кода с тысяч до сотен строк, причем исполняемый модуль, вероятно, также уменьшился в объеме где-то на порядок.

10.2. Пример

В начале 80-х я помог советом программисту, работавшему с географической базой данных. Каждому из двух тысяч соседей присваивался индивидуальный номер (`0..1999`), а его место жительства отмечалось точкой на карте. Система давала пользователю возможность обратиться к одной из двух тысяч точек с помощью сенсорной панели. Программа преобразовывала выбранную точку в пару чисел (`x, y`) в диапазоне `0..199` — панель имела площадь примерно 4 квадратных фута, а программа обеспечивала разрешение в четверть дюйма. Затем по паре точек (`x, y`) нужно было определить, кого из двух тысяч соседей выбрал пользователь. Поскольку никакие две точки не могли иметь совпадающие координаты, программист представил карту с помощью массива индексов точек 200×200 (соответствующий элемент имел значение `0..1999` или `-1` для пустой точки). Нижний левый угол массива

мог бы выглядеть так, как показано на рис. 10.1 (пустые клетки — это точки, в которых ничего нет).

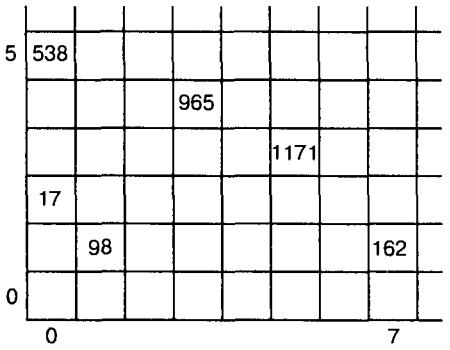


Рис. 10.1. Массив точек

На этом рисунке точка 17 имеет координаты (0, 2), точка 538 — (0, 5), а остальные места в первом столбце пусты.

Массив было легко описать в программе, и он обеспечивал быстроту доступа. Программисту пришлось выбирать между 32- и 16-разрядными целыми. Если бы он выбрал 32-разрядные, массив из 200*200 = 40 000 элементов занял бы 160 Кбайт памяти. Поэтому программист выбрал более краткое представление, и массив занял 80 Кбайт, то есть 1/6 часть памяти объемом в половину мегабайта. В начальный период использования системы никаких проблем не возникало. Однако с ростом системы начались проблемы с памятью. Программист попросил меня помочь уменьшить объем памяти, занимаемый этой структурой. Что бы вы ему посоветовали?

Эта задача — классический пример на использование разреженных структур данных. Пример относится к далекому прошлому, но недавно мне пришлось столкнуться с аналогичной проблемой представления матрицы 10 000×10 000 с миллионом активных элементов в 100 Мбайт памяти.

Очевидный способ представления разреженных матриц состоит в создании массива, соответствующего столбцам матрицы, и связных списков, содержащих активные элементы в соответствующих столбцах. Рисунок 10.2 пришлось повернуть на 90° по часовой стрелке, чтобы он лучше уместился в книге.

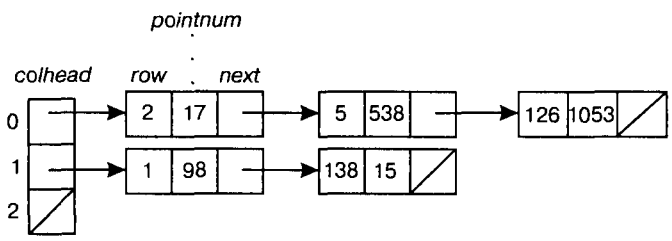


Рис. 10.2. Разреженная матрица

На нем изображены три точки в первом столбце: точка 17 имеет координаты (0, 2), точка 538 имеет координаты (0, 5), а точка 1053 имеет координаты (0, 126). Во втором столбце у нас две точки, а в третьем — ни одной. Точку с координатами (i, j) можно найти с помощью следующего кода:

```
for (p = colhead[i], p != NULL; p = p->next)
  if p->row == j
    return p->pointnum
return -1
```

В худшем случае для поиска элемента потребуется проверка 200 узлов, а в среднем — всего десяти.

В этой структуре используется массив из 200 указателей и 200 записей, каждая из которых содержит целое и два указателя. Приложение 3 говорит нам, что указатели займут 800 байт. Если мы выделим память под записи как под массив с 2000 элементов, каждая из них займет 12 байт памяти, а все вместе они займут 24 800 байт. Если бы мы воспользовались функцией malloc, описанной в этом приложении, на каждую запись приходилось бы 48 байт, поэтому структура выросла бы в объеме от 80 до 96,8 Кбайт.

Программисту нужно было реализовать структуру на версии языка FORTRAN, в которой не поддерживались указатели и структуры. Поэтому мы использовали массив из 201 элемента для представления столбцов и два параллельных массива по 2000 элементов для представления точек. На рис. 10.3 изображены эти три массива, причем стрелки указывают на те элементы, индексы которых хранятся в нижнем массиве. Нумерация элементов массивов в языке FORTRAN начинается с единицы, а не с нуля, поэтому изображение на рисунке немного не соответствует тому, что в действительности было сделано в программе.

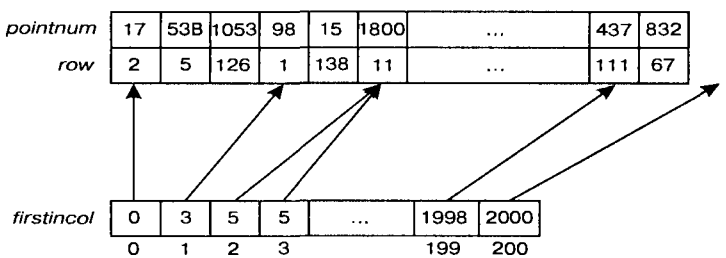


Рис. 10.3. Индексация с помощью массивов

Точки в столбце i находятся в массивах row и pointnum между позициями firstincol[i] и firstincol[i+1]-1. Хотя столбцов всего 200, нам приходится определять элемент firstincol[200], чтобы это условие выполнялось всегда. Для обращения к точке с координатами (i, j) используется следующий код:

```
for k = [firstincol[i], firstincol[i+1])
  if row[k] == j
    return pointnum[k]
return -1
```

Эта версия программы использует два массива по 2000 элементов и один с 201 элементом. Программист реализовал эту схему именно так, как она здесь описана, используя 16-разрядные целые числа, и она заняла 8402 байта памяти. Программа работает несколько более медленно, чем в исходном варианте, поскольку в среднем для обращения к точке требовалось 10 обращений к элементам массивов. Однако ее производительности вполне хватало пользователям. Благодаря хорошей модульной структуре системы в целом нам удалось внести изменения за несколько часов, изменив лишь несколько функций. Производительность программы не ухудшилась, а нам удалось освободить 70 Кбайт памяти, которых нам так не хватало.

Если бы нам показалось, что структура все равно занимает слишком много места, мы бы могли еще более уменьшить ее в объеме. Поскольку значения элементов массива row не превышают 200, их можно хранить в 1-байтовых беззнаковых целых (char). Это уменьшает занимаемый объем памяти до 6400 байт. Можно и вовсе избавиться от массива row, помещая соответствующую координату точки в ее запись:

```
for k = [firstincol[i], firstincol[i+1])
  if point[pointnum[k]] row == j
    return pointnum[k]
return -1
```

Это уменьшает занимаемый объем до 4400 байт.

В реальной системе скорость поиска точек была важна для взаимодействия с пользователем и для других функций, которые пользовались тем же интерфейсом для своих целей. Если бы время доступа не имело значения, мы могли бы добавить в структуру данных точки обе ее координаты и достигли бы максимального уменьшения объема, но нам приходилось бы последовательно перебирать их все для проверки наличия точки с данными координатами. Даже если не добавлять в структуру точки эти поля, объем памяти структуры можно уменьшить до 4000 байт с помощью «ключевой индексации»: поиск осуществляется в массиве, i-й элемент которого содержит 2 поля по 1 байту, в которых записаны значения row и col точки i.

Этот пример иллюстрирует несколько основных моментов, связанных со структурами данных. Мы имеем дело с классической задачей о представлении *разреженного массива* (разреженным массивом называется такой массив, в котором бо́льшая часть элементов имеет одно и то же значение, обычно нулевое). Решение просто по смыслу, и его легко реализовать. Мы используем несколько способов экономии памяти. Нам не приходится заводить отдельный массив для хранения последнего элемента в столбце, поскольку он идет непосредственно перед первым элементом в следующем столбце. Это типичный пример, в котором некоторая величина вычисляется заново, вместо сохранения ее в памяти. Аналогично, нет необходимости в массиве row, если есть массив col, потому что к последнему мы обращаемся только через массив firstincol, а значит, мы всегда знаем, в каком столбце мы находимся. Хотя мы начали с 32-разрядного представления для массива row, мы затем сократили разрядность до 16 и даже до 8 бит. Мы начали с записей, но в конце концов вернулись к массивам, попутно экономя по килобайту памяти то в одном месте, то в другом.

10.3. Размещение данных в памяти

Хотя упрощение задачи является обычно наиболее легким путем решения проблемы, в некоторых сложных случаях этого оказывается недостаточно. В этом разделе мы изучим методы, позволяющие уменьшить объем памяти, занимаемый данными. В следующем разделе речь пойдет о памяти, занимаемой программой во время выполнения.

Не храните то, что можно вычислить

Занимаемый некоторым объектом объем памяти может быть существенно уменьшен, если его не сохранять, а вычислять заново каждый раз, когда в нем возникает потребность. Это именно тот способ, которым мы воспользовались, когда сохранили матрицу точек и искали в ней нужную каждый раз заново. Вместо таблицы простых чисел можно написать функцию, проверяющую заданное число на простоту. При применении этого метода мы «покупаем» память, платя за это производительностью программы, и он применим только тогда, когда объекты, которые нужно сохранить в памяти, могут быть получены путем вычислений.

Программы-генераторы используются при необходимости проверить какие-либо программы на одинаковых последовательностях случайных чисел. При этом может сравниваться их производительность или же целью может быть регрессионное тестирование правильности. В зависимости от приложения случайный объект может представлять собой файл с набором произвольных строк или граф со случайными вершинами. Вместо того чтобы хранить в памяти весь объект целиком, можно написать программу, которая будет его создавать, инициализируя генератор случайных чисел каждый раз одним и тем же конкретным числом. Огромные по объему объекты могут быть представлены в нескольких байтах, но время доступа к ним, разумеется, возрастет.

Пользователям персональных компьютеров предоставляется аналогичный выбор, когда они устанавливают программы с компакт-дисков или с DVD-дисков. «Стандартная» установка может занять несколько сотен мегабайт на жестком диске компьютера, но зато программа будет работать быстро. «Минимальная» установка позволяет оставить бóльшую часть программы на более медленном устройстве (приводе чтения компакт-дисков), но зато место на жестком диске останется свободным. Итак, память на жестком диске сохраняется за счет времени, которое тратится каждый раз при запуске такой программы.

Для многих программ, выполняемых в сетях, очень важным параметром является объем передаваемой по сети информации. Иногда можно уменьшить количество передаваемых данных, кэшируя их на компьютере пользователя в соответствии с правилом «не передавайте данные, если их можно сохранить локально».

Разреженные структуры данных

Раздел 10.2 представляет собой введение в разреженные структуры. В разделе 3.1 мы экономили память, храня разреженную трехмерную таблицу в двухмерном

массиве. Если сохраняемый ключ использовать в качестве индекса таблицы, его уже не нужно сохранять в ней. Вместо этого можно сохранить его атрибуты (например, количество обращений к нему). Приложения метода ключевой индексации описаны в каталоге алгоритмов в приложении 1. В примере с разреженной матрицей (см. выше) индексация с помощью массива firstincol позволила нам обойтись без массива col.

Использование указателей на совместно используемые объекты большого объема (длинные текстовые строки) позволяет обойтись без накладных расходов на хранение нескольких копий одного объекта, но при этом следует быть аккуратным с изменением совместно используемых объектов, потому что не все его владельцы могут желать подобного изменения. Этот метод используется в моем настольном альманахе для расчета календарей с 1821 по 2080 год. Вместо того чтобы хранить в памяти 260 разных календарей, программа сохраняет 14 канонических календарей (1 января может попадать на любой из семи дней недели, да еще год может быть високосным или не високосным) и таблицу с номером канонического календаря для каждого из 260 лет.

В некоторых телефонных системах для увеличения эффективной полосы пропускания разговор рассматривается как разреженная структура данных. Когда интенсивность сигнала в одном из направлений спадает ниже определенного уровня, тишина представляется в сжатом кодированном виде, поэтому освободившийся ресурс полосы пропускания отдается другим линиям.

Сжатие данных

Теория информации говорит нам о том, что можно уменьшить занимаемый данными объем путем их кодирования. В примере с разреженной матрицей мы сжали представление номера столбца с 32 до 16 бит, а затем и до 8 бит. На заре эры персональных компьютеров я написал программу, которая тратила бóльшую часть времени на чтение и запись длинных последовательностей десятичных чисел. Я изменил программу так, чтобы кодировать две десятичные цифры a и b в одном байте (вместо обычных двух), вычисляя значение этого байта по формуле c = 10 * a + b. Затем исходные цифры можно было восстановить двумя операциями:

```
a = c / 10
b = c % 10
```

Эта простая схема вдвое уменьшила время ввода и вывода данных, а заодно позволила вдвое сократить размер файла с данными, который стал умещаться на одной дискете вместо прежних двух. Таким образом, кодирование может уменьшить объем памяти, занимаемый записями, но записи уменьшенного объема могут требовать больше времени на кодирование и декодирование (см. задачу 6 в конце главы).

Теория информации позволяет сжать поток записей, передаваемый по какому-либо каналу, такому как поток чтения/записи файла или сетевое соединение. Запись звука с уровнем качества компакт-диска требует двух каналов с 16-разрядным представлением уровня и частотой дискретизации 44 100 Гц. Одна секунда

в этом представлении занимает 176 400 байт. Стандарт MP3 позволяет сжимать типичные звуковые файлы (в особенности музыку) во много раз. В задаче 10 вам предлагается определить эффективность некоторых широко используемых форматов представления текста, изображений, звуков и т. п. Некоторые программисты снабжают свои программы специальными алгоритмами сжатия. В разделе 13.8 (глава 13) данной книги описано сжатие файла с 75 000 английских слов до объема 52 Кбайт.

Политика выделения памяти

В некоторых случаях важен не столько объем используемой памяти, сколько метод работы с ней. Предположим, что в вашей программе используются записи трех типов (x, y и z), причем все записи одинакового размера. В некоторых языках программирования вам может показаться удобным объявить, скажем, 10 000 объектов каждого типа. Но что, если вам понадобится 10 001 запись типа x и ни одной записи типов y и z? Программа завершит работу с сообщением об ошибке, попытавшись обратиться к 10 001 записи, в то время как 20 000 других выделенных записей останутся неиспользованными. Динамическое выделение памяти позволяет избежать подобных очевидных затрат, предоставляя программисту возможность выделять память под объекты по мере необходимости.

Динамическое выделение подразумевает, что нет необходимости запрашивать память под объект до тех пор, пока он не понадобится. Можно работать с записями переменного размера, то есть выделять под объект ровно столько памяти, сколько нужно. Во времена перфокарт, когда записи состояли из 80 колонок, обычным делом было наличие на диске более половины дополняющих пробелов. В файлах переменной длины конец строки обозначался символом перевода строки, поэтому емкость дисков при использовании таких файлов удваивалась. Однажды мне удалось утроить скорость ввода и вывода программы, используя записи переменной длины. Максимальная длина записи была 250 символов, но в среднем использовалось только 80 из них.

Сборка мусора

Сборщик мусора (garbage collector) позволяет заново использовать память, отведенную под уже освобожденные записи. Алгоритм сортировки с помощью кучи (heapspot algorithm) в разделе 14.4 (глава 14) попеременно хранит в одних и тех же адресах памяти разные логические структуры данных.

Другой подход к совместному использованию памяти был применен Брайаном Керниганом (Brian Kernighan) в начале 70-х при решении задачи коммивояжера, где большая часть памяти отводилась под две матрицы 150×150. В двух матрицах, которые мы назовем a и b, хранилось расстояние между точками. Керниган знал, что диагонали этих матриц должны быть нулевыми (a[i, i] = 0) и что матрицы должны быть симметричны (a[i, j] = a[j, i]). Поэтому он поместил две треугольные матрицы в одну квадратную (c), один из углов которой изображен на рис. 10.4.

Затем Керниган обращался к элементу a[i, j] следующим образом:

```
c[max(i,j), min(i,j)]
```

0	*b*[0,1]	*b*[0,2]	*b*[0,3]
a[1,0]	0	*b*[1,2]	*b*[1,3]
a[2,0]	*a*[2,1]	0	*b*[2,3]
a[3,0]	*a*[3,1]	*a*[3,2]	0

Рис. 10.4. Объединение двух треугольных матриц

Аналогично можно обратиться и к элементу b[i,j], поменяв местами min и max. Подобное представление используется в различных программах со времен сотворения мира. Оно несколько замедлило и усложнило программу Кернигана, но переход от двух матриц по 22 500 слов к одной на машине с 30 000 слов значил довольно много. А если бы матрицы были размерностью 30 000 × 30 000, аналогичное действие было бы необходимым и на современной машине с гигабайтом памяти.

В современных системах важно учитывать работу кэш-памяти. Хотя я изучал относящуюся к этому вопросу теорию много лет, эффективность такого подхода я почувствовал, лишь когда стал работать с программами, размещавшимися на нескольких компакт-дисках. Телефонным справочником и картой США на компакт-дисках было очень удобно пользоваться, потому что компакт-диски почти не нужно было менять. Программа требовала смены диска лишь в том случае, когда я переходил от одной части страны к другой. Но когда я купил первую энциклопедию на двух компакт-дисках, мне пришлось менять диски так часто, что я вернулся к предыдущей версии, умещавшейся на одном компакт-диске. В этом примере размещения данных не учитывалась последовательность обращения к ним. В решении задачи 4 из главы 2 приведен график производительности трех алгоритмов с различными последовательностями доступа к памяти. В разделе 13.2 (глава 13) мы встретимся с приложением, в котором массивы, содержащие большие объемы данных, работают быстрее списков, потому что массивы лучше обрабатываются кэшем.

10.4. Методы уменьшения размера кода

Иногда памяти не хватает не из-за объема данных, а из-за размера исполняемого модуля программы. В давние времена мне случалось сталкиваться с графическими программами, которые состояли из многих страниц кода следующего вида:

```
for i = [17, 43] set(i, 68)
for i = [18, 42] set(i, 69)
for j = [81, 91] set(30, i)
for j = [82, 92] set(31, i)
```

где set(i, j) включает соответствующий пиксел экрана. Добавление специальных функций (например, hor и vert для рисования горизонтальных и вертикальных линий) позволило бы заменить этот код на последовательность вызовов:

```
hor(17, 43, 68)
hor(18, 42, 69)
vert(81, 91, 30)
vert(82, 92, 31)
```

Такой участок кода можно было бы заменить интерпретатором, считывающим данные из массива, наподобие такого:

```
h 17 43 68
h 18 42 69
v 81 91 30
v 82 92 31
```

Если это все еще занимает слишком много места, каждую из строк можно упаковать в 32-разрядное слово, отведя 2 бита на команду (h, v и еще две запасные) и 10 битов на каждое из трех чисел, лежащих в диапазоне 0..1023. (Преобразование должно, разумеется, осуществляться самой программой.) Этот гипотетический пример наглядно демонстрирует результат применения нескольких методов уменьшения объема кода.

Определение функции

Замена повторяющегося кода функцией упростила приведенную выше программу и уменьшила ее требования к памяти, одновременно увеличив ясность и наглядность. Это тривиальный пример разработки «снизу-вверх». Несмотря на то что нельзя игнорировать преимущества разработки «сверху-вниз», взгляд на мир сквозь призму хороших базовых объектов, компонент и функций может сделать систему проще и уменьшить занимаемый ей объем.

Корпорации Microsoft пришлось удалить несколько редко используемых функций, чтобы ужать операционную систему Windows до Windows CE, которая может выполняться в небольшом объеме памяти мобильных вычислительных систем. Уменьшенный в объеме интерфейс пользователя отлично работает на небольших компьютерах, начиная с встроенных в бытовую технику и заканчивая Palm. В то же время знакомый интерфейс дает пользователям возможность сразу начать работу, не тратя время на обучение. Уменьшенный интерфейс программирования приложений остается знакомым программистам, пишущим программы под Windows, а некоторые приложения могут работать в Windows CE без изменений.

Интерпретаторы

В графической программе нам удалось заменить последовательность команд одним интерпретатором и набором данных для обработки. В разделе 3.2 (глава 3) описан интерпретатор для формирования писем. Хотя его основное предназначение в том, чтобы программа была проще и удобнее, он также сокращает объем кода.

В книге «Практика программирования» (см. раздел 5.9 главы 5) Керниган и Пайк посвящают целый раздел (9.4) интерпретаторам, компиляторам и виртуальным машинам. Приводимые примеры иллюстрируют вывод, к которому они приходят: *«Виртуальные машины — красивая, но старая идея, которая недавно стала вновь популярной с появлением языка Java и виртуальной машины Java Virtual Machine. Это средство получения переносимых и эффективных представлений программ, написанных на языках высокого уровня»*.

Перевод на машинный код

Программисты не слишком хорошо могут управлять переводом из исходного языка на машинный, а от этого перевода также зависит объем исполняемого модуля программы. Небольшие изменения компилятора уменьшили объем одной из ранних версий Unix на 5%. Если ничто больше не помогает, можно попытаться вручную переписать программу (или ее часть) на ассемблере. Этот длительный и приводящий к ошибкам процесс обычно дает не слишком заметные результаты (в уменьшении объема кода), тем не менее он часто используется в системах с жестким ограничением памяти, таких как цифровые сигнальные процессоры.

В 1984 году компьютер Apple Macintosh был примечательной новинкой. Несмотря на слабые характеристики (128 Кб оперативной памяти), он обладал потрясающим интерфейсом пользователя и для него имелось множество программ. Команда разработчиков планировала продать огромное количество машин, но могла себе позволить установить на компьютер только 64 Кбайт ПЗУ. Несмотря на крохотный объем, функциональность ПЗУ была очень большой. Разработчикам удалось обеспечить это благодаря аккуратному кодированию всех функций (обобщение операторов, слияние функций и отказ от лишних возможностей). Наконец, они вручную переписали все функции ПЗУ на ассемблере. По их оценкам, получившийся код занимал вдвое меньший объем, чем если был он был получен компиляцией с языка высокого уровня (с тех пор компиляторы значительно улучшились). Маленький по объему код ПЗУ обладал еще одним полезным качеством: высоким быстродействием.

10.5. Принципы

«Стоимость» памяти

Что произойдет, если программа будет использовать на 10% больше памяти? В некоторых системах это увеличение не будет стоить ничего: не использовавшиеся ранее биты пойдут теперь в дело. На системах с очень малым объемом памяти программа может вообще перестать работать, потому что ей будет не хватать памяти. Если данные передавать по сети, то время их передачи, скорее всего, вырастет на 10%. В системах с кэшированием и подкачкой время работы программы может значительно возрасти, поскольку данные, бывшие в наиболее доступной процессору области памяти, переместятся в кэш второго уровня, ОЗУ или вообще на диск

(см. раздел 13.2 главы 13 и решение задачи 4 из главы 2). Нужно оценить стоимость памяти, прежде чем пытаться уменьшать эту стоимость.

Эффективное уменьшение объема

В разделе 9.4 (глава 9) мы доказали, что при выполнении программы время обычно тратится неравномерно. На выполнение некоторых функций программа тратит большую часть времени. Для памяти верно противоположное: сколько бы раз ни нужно было выполняться некоторой инструкции, на ее хранение будет отводиться один и тот же объем (за исключением тех случаев, когда исчезает необходимость подгрузки больших участков кода в ОЗУ или в кэш небольшого объема). Однако и в данных бывают наиболее выделяющиеся поглотители: данные некоторых типов обычно поглощают большую часть используемой памяти. В примере с разреженными матрицами одна структура данных использовала 15% памяти в компьютере с 512 Кбайт ОЗУ. Замена этой структуры на новую, занимающую в 10 раз меньший объем, сильно повлияла на работу системы, но если бы мы в 100 раз уменьшили структуру, все экземпляры которой занимают в памяти 1 Кбайт, эффект от этого был бы пренебрежимо мал.

Измерение объемов памяти

В большинстве систем предусмотрены так называемые мониторы производительности (performance monitor), позволяющие программисту определять объем памяти, используемый выполняемой программой. В приложении 3 приведены данные об объеме памяти, занимаемом типичными структурами в C++; эти данные могут быть особенно полезны, если их использовать совместно со сведениями, поставляемыми монитором производительности. В некоторых случаях чрезвычайно полезными оказываются специальные средства. Когда программа Дуга Макилроя (Douge McIlroy) начала занимать слишком много места, он сравнил исходный код с результатом работы компоновщика и определил объем, занимаемый каждой из строк исходного кода (некоторые макросы раскрывались в сотни строк объектного кода). Это дало ему возможность уменьшить объем его программы. Однажды мне удалось найти утечку памяти в программе, просматривая графическую иллюстрацию работы алгоритма выделения памяти.

Компромиссы

Иногда программисту приходится поступиться производительностью, функциональностью или удобством обслуживания программы ради освобождения памяти. В этой главе было приведено несколько примеров, в которых уменьшение программы в объеме положительно влияло на некоторые другие ее качества. В разделе 1.4 (глава 1) представление данных в битовом массиве дало возможность хранить все данные в ОЗУ, что сократило время выполнения с минут до секунд и длину кода с сотен до десятков строк. Да, подобный эффект оказался возможен лишь потому, что исходное решение было далеким от оптимального, но программисты, не достигшие совершенства, часто получают именно такие решения. Всегда нужно

искать методы, позволяющие улучшить все характеристики продукта, прежде чем идти на компромиссы.

Работа с окружением

Среда программирования может весьма существенно влиять на эффективность использования памяти. Важными оказываются представления, используемые компилятором и модулями времени исполнения, системами выделения памяти и подкачки. Модели стоимости памяти, аналогичные приведенной в приложении 3, помогут вам удостовериться, что вы боретесь не с собственной системой.

Использование подходящих средств

Мы рассмотрели четыре метода экономии памяти, отводимой под данные (повторное вычисление, разреженные структуры, теория информации и политики выделения), три метода уменьшения объема кода (определение функций, интерпретация, трансляция) и один глобальный принцип (простота). Если у вашего компьютера недостаточно памяти, подумайте обо всех возможных вариантах решения проблемы.

10.6. Задачи

1. В конце 70-х Стю Фелдман (Stu Feldman) написал компилятор языка FORTRAN 77, который с трудом помещался в 64 Кбайт, отводимые под код программы. Первоначально для экономии памяти он упаковал некоторые целочисленные переменные в наиболее ресурсоемких записях в четырехбитные поля. Отказавшись впоследствии от упаковки переменных, он обнаружил, что, хотя объем данных увеличился на несколько сот байт, полный объем программы уменьшился на несколько килобайт. Почему?
2. Как бы вы написали программу для построения структуры разреженной матрицы, описанной в разделе 10.2? Можете ли вы найти другие простые, но эффективные (в плане экономии памяти) структуры данных для этой задачи?
3. Каков суммарный объем всех дисков вашего компьютера? Какая его часть свободна? Сколько у вас оперативной памяти? Какая ее часть обычно доступна? Можете ли вы измерить объем различных видов кэш-памяти в вашей системе?
4. Изучите не имеющие отношения к компьютерам данные (в альманахах и тому подобном), стараясь найти примеры экономии занимаемого ими места.
5. На заре программирования Фред Брукс (Fred Brooks) столкнулся с другой проблемой представления большого массива на маленьком компьютере (помимо описанной в разделе 10.1). Вся таблица не могла храниться в массиве, поскольку на каждый из элементов приходилось несколько битов (одна

десятичная цифра, если быть точным, — я же сказал, что дело было на заре программирования). Тогда он попытался использовать численный анализ, чтобы подобрать функцию, описывающую элементы таблицы. Ему действительно удалось найти такую функцию, которая позволяла почти точно получить все элементы таблицы (ни один из них не отличался от значения функции в соответствующей точке более, чем на несколько единиц), а сама функция занимала очень мало памяти, но установленные ограничения не позволяли считать точность приближения достаточной. Как удалось Бруксу достичь требуемой точности при использовании ограниченного объема памяти?

6. При обсуждении сжатия данных в разделе 10.3 мы упомянули декодирование выражения `10 * a + b` с помощью операций `%` и `/`. Обсудите компромисс между занимаемым объемом памяти и быстродействием, на который в данном случае приходится идти, а также рассмотрите возможность замены этих операций на логические или на обращение к таблице.

7. В большинстве профилирующих программ (раздел 9.1) через равные промежутки времени записывается значение счетчика команд (program counter). Придумайте структуру данных для сохранения этих значений, которая была бы эффективной в плане быстродействия и занимаемой памяти, а данные в ней были бы достаточно информативны.

8. При использовании очевидного представления данных для сохранения даты требуется 8 байтов (ДДММГГГГ); номер социального страхования занимает 9 байтов (DDD-DD-DDDD), а имя человека занимает 25 байт (14 на фамилию, 10 на имя и 1 на первую букву отчества). Насколько вы сможете уменьшить эти числа, если перед вами стоит задача экономии памяти?

9. Уменьшите электронный словарь английского языка до минимального размера. При вычислении этого размера следует учитывать не только объем файла со словарем, но и объем программы-интерпретатора.

10. Звуковые файлы с необработанными данными (такие, как `.wav`) могут быть сжаты в формат `.mp3`. Необработанные графические данные (`.bmp`) могут быть преобразованы в `.gif` или `.jpg`. Фильмы можно переписать из `.avi` в `.mpg`. Проведите эксперименты с этими форматами файлов, чтобы определить их эффективность. Насколько эффективны эти специальные форматы сжатия по сравнению с программами сжатия общего назначения (такими, как `gzip`)?

11. Замечание читателя: «*В современных системах важен не объем написанного, но объем используемого кода*». Изучите свои программы, сравнивая их объем до и после компоновки. Как можно уменьшить этот объем?

10.7. Дополнительная литература

Двадцатое, юбилейное издание книги Фреда Брукса «Мифический человеко-месяц» (Fred Brooks, Mythical Man Month) было выпущено издательством Addison

Wesley в 1995 году. В него вошли очаровательные эссе из первого издания книги вместе с несколькими новыми, включая «Никаких серебряных пуль — случайность и закономерности в программировании». Глава 9 этой книги называется «Десять фунтов в пятифунтовой авоське»; она посвящена управлению использованием памяти в больших проектах. Автор затрагивает такие существенные вопросы, как бюджет памяти, спецификация функций и компромиссы между скоростью выполнения и занимаемой памятью.

Книги, цитируемые в разделе 8.8 главы 8, расскажут вам о теории и методах, лежащих в основе эффективных по занимаемой памяти алгоритмов и структур данных.

10.8. Пример эффективного сжатия

В начале 80-х Кен Томпсон (Ken Thompson) написал программу, решавшую шахматные эндшпили в два этапа (например, король и две ладьи против короля и коня). Другая написанная Томпсоном и Джо Кондоном (Joe Condon) программа стала чемпионом мира по шахматам среди компьютеров. Программа работала в два этапа. На этапе обучения вычислялось количество ходов до мата при всех начальных комбинациях (для набора из четырех или пяти фигур) путем обратного хода от всех возможных матовых комбинаций. Специалисты по информатике назовут этот метод динамическим программированием, а эксперты по шахматам — ретроградным анализом. В результате получалась база данных, делавшая программу всеведущей в эндшпилях. Поэтому на втором этапе программа играла идеально. Эксперты по шахматам называли ее игру «*сложной, быстрой, развернутой и трудной*», а также «*мучительно медленной и загадочной*». Программе удалось опрокинуть ставшие догматическими представления в шахматном мире.

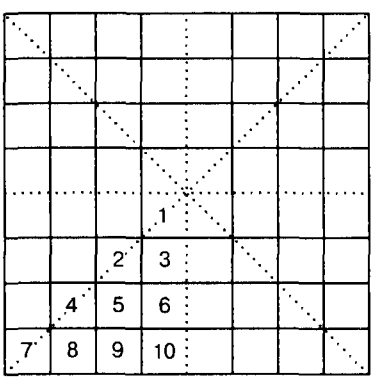


Рис. 10.5. Симметрия шахматной доски

Хранение всех комбинаций в явном виде потребовало бы слишком большой объем памяти. Томпсон использовал код комбинации в качестве ключа к файлу с информацией о ней. На каждую запись в файле отводилось 12 бит, в которых помещалось, в частности, количество ходов до матовой комбинации. Поскольку на

шахматной доске 64 клетки, положение пяти фигур можно закодировать пятью целыми числами от 0 до 63. Получающийся ключ из 30 битов предполагает наличие таблицы с 2^{30} элементами или 1,07 миллиарда 12-битных записей в базе данных, что превосходило емкость существовавших в то время дисков.

Главная идея Томпсона заключалась в том, что при зеркальных отображениях доски относительно штриховых линий на рис. 10.5 получающиеся комбинации будут иметь те же характеристики, поэтому их не нужно описывать отдельно.

Поэтому его программа предполагала, что белый король находится в одном из пронумерованных полей. Произвольную комбинацию можно привести к этому состоянию, сделав не более трех отражений. Это уменьшило объем файла до 10×64^4 или 10×2^{24} 12-битных записей. Дальше Томпсон рассуждал так. Черный король не может стоять рядом с белым, поэтому остается только 454 разрешенных положения для двух королей (при том, что белый король находится в одном из десяти пронумерованных полей). Этот факт позволил уменьшить базу до 454×64^3 или около 121 миллиона 12-битных записей, так что теперь она спокойно умещалась на один диск, целиком под нее отводившийся.

Хотя Томпсон знал, что его программа будет на компьютере в единственном экземпляре, ему все равно нужно было уместить ее на один диск. Он воспользовался симметрией данных для уменьшения занимаемого ей объема в восемь раз. Это уменьшение было жизненно важным для системы в целом. Уменьшение объема памяти ускорило и работу программы: уменьшение количества рассматриваемых в фазе обучения позиций сократило время выполнения с многих месяцев до нескольких недель.

Часть

Программный продукт

Здесь начинается самое интересное. В первой и второй частях мы заложили основу для того, чем будем заниматься в ближайших пяти главах. А займемся мы написанием интересных программ. Решаемые задачи важны сами по себе, кроме того, они иллюстрируют применение методов из предыдущих глав в реальных приложениях.

Глава 11 посвящена нескольким алгоритмам сортировки общего назначения. В главе 12 мы рассматриваем некоторую частную задачу из реального приложения (формирование случайной последовательности целых чисел), демонстрируя различные подходы к ее решению. Один из подходов заключается в использовании наборов, которые являются темой главы 13. Глава 14 посвящена «кучам» и использующим их эффективным алгоритмам сортировки и обработки приоритетных очередей. В главе 15 мы займемся решением нескольких задач, связанных с обработкой текстов, включающих поиск слов в очень длинных текстовых строках.

Сортировка

Как отсортировать последовательность записей в определенном порядке? Ответ обычно прост: используйте библиотечную функцию сортировки. Именно этот подход был применен в решении 1.1 и в программе поиска анаграмм из раздела 2.8 главы 2 (в последнем случае дважды). К сожалению, это не всегда срабатывает. Существующие подпрограммы сортировки может оказаться слишком утомительным подключать, и они могут оказаться слишком медленными для данной задачи (как в разделе 1.1). В такой ситуации у программиста нет выбора: приходится писать функцию сортировки самому.

11.1. Сортировка вставкой

Большинство картежников, сами того не сознавая, пользуются именно таким методом сортировки для упорядочения пришедших им карт. Когда игрок получает очередную карту, все предыдущие уже отсортированы, поэтому он просто вставляет ее в нужное место. Для сортировки массива x[n] в порядке возрастания начинать следует с первого элемента, считая его отсортированной подпоследовательностью x[0..0]. Затем нужно вставлять элементы x[1], ..., x[n-1] в правильные позиции, как это делается в приведенном ниже псевдокоде:

```
for i = [1, n)
  /* инвариант  элементы x[0  i-1] отсортированы */
  /* цель: вставить x[i] в нужную позицию в массиве x[0  i] */
```

Последовательность сортировки массива из четырех элементов иллюстрируется ниже. Символ «|» показывает текущее значение переменной i; элементы слева от этого символа уже отсортированы, а справа — нет.

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

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

Вставка элемента в нужную позицию производится циклом, в котором элементы перебираются справа налево, а в переменной j хранится индекс очередного вставляемого элемента. В цикле текущий элемент переставляется местами с предыдущим, если этот предыдущий элемент существует (то есть j>0) и текущий элемент еще не установлен в нужное положение (он и предыдущий элементы находятся в неправильном порядке). Итак, получившаяся программа сортировки вставкой приведена на листинге 11.1.

Листинг 11.1. Сортировка вставкой: версия 1

```
for i = [1, n)
  for (j = i, j > 0 && x[j-1] > x[j], j--)
    swap(j-1, j)
```

В тех редких случаях, когда мне нужно написать свою собственную сортировку, я начинаю именно с этой функции, потому что она очень проста — всего три очевидные строки.

Программисты, стремящиеся к оптимизации, могут счесть нерациональным вызов функции swap из тела внутреннего цикла. Программу можно ускорить, раскрыв функцию явно, хотя многие оптимизирующие компиляторы способны делать это за нас. Заменим вызов функции нижеследующим кодом, в котором переменная t используется для обмена x[j] и x[j-1].

```
t = x[j], x[j] = x[j-1]; x[j-1] = t
```

На моем компьютере вторая версия сортировки работает примерно в три раза быстрее, чем первая.

После этого улучшения появляется возможность сделать следующий шаг. Поскольку переменной t несколько раз присваивается одно и то же значение (исходно находящееся в x[i]), мы можем вынести присваивания, относящиеся к этой переменной, за рамки внутреннего цикла, а также изменить вид сравнения, что даст третью версию сортировки вставкой (листинг 11.2).

Листинг 11.2. Сортировка вставкой: версия 3

```
for i = [1, n)
  t = x[i]
  for (j = i, j > 0 && x[j-1] > t, j--)
    x[j] = x[j-1]
  x[j] = t
```

Эта программа сдвигает элементы вправо до тех пор, пока они превосходят значение t, а потом ставит t в правильную позицию. Эта функция из пяти строк чуть сложнее своих предшественников, но на моем компьютере она работает примерно на 15% быстрее, чем вторая версия той же сортировки.

Для случайного расположения элементов во входном массиве, как и в худшем случае (обратный порядок сортировки), время выполнения сортировки вставкой пропорционально n^2. Таблица 11.1 содержит данные о времени выполнения трех программ, когда на вход подается n случайных целых чисел.

Таблица 11.1. Сортировка вставкой: время выполнения различных версий программы

Программа	Объем (строк на языке C)	Время, нс
Сортировка вставкой (1)	3	$11,9n^2$
Сортировка вставкой (2)	5	$3,8n^2$
Сортировка вставкой (3)	5	$3,2n^2$

Третьей программе требуется несколько миллисекунд для сортировки n = 1000 целых чисел, треть секунды на n = 10 000 целых, и почти час на сортировку миллиона чисел. Скоро мы встретимся с программой, сортирующей миллион чисел меньше, чем за секунду. Если входной массив уже почти отсортирован, сортировка вставкой работает гораздо быстрее, поскольку все элементы сдвигаются лишь на небольшое расстояние. Алгоритм в разделе 11.3 данной главы основан именно на этом свойстве.

11.2. Простая быстрая сортировка

Этот алгоритм был впервые описан К. А. Р. Хоаром в его классической статье «Быстрая сортировка» (C. A. R. Hoare, Quicksort. Computer Journal, 5, 1, April 1962, p. 10–15). В этом алгоритме используется подход «разделяй и властвуй», уже упоминавшийся в разделе 8.3: *чтобы отсортировать массив, мы разделяем его на два подмассива и сортируем каждый из них рекурсивно.* Например, для сортировки массива из восьми элементов

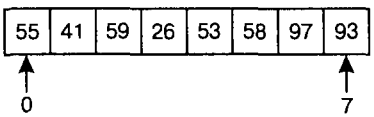


Рис. 11.1. Сортируемый массив

мы разбиваем его первым элементом (55), так чтобы все элементы, меньшие 55, расположились слева от него, а все большие — справа.

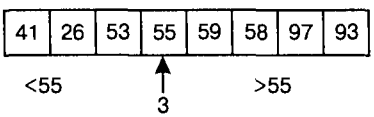


Рис. 11.2. Разбиение массива

Если затем рекурсивно отсортировать подмножества x[0..2] и x[4..7] по отдельности, весь массив окажется отсортирован.

Среднее время работы этого алгоритма оказывается существенно меньшим $O(n^2)$ сортировки вставкой, поскольку операция разбиения дает гораздо больше для достижения цели. После разбиения n элементов примерно половина из них окажется ниже выделенного элемента, а половина — выше. За то же самое время

в программе сортировки вставкой на свое место устанавливается один-единственный элемент.

Итак, у нас есть набросок рекурсивной функции. Опишем рабочую часть массива с помощью индексов l, u (нижняя и верхняя границы). Рекурсия останавливается, когда мы добираемся до массива с числом элементов, меньшим двух. Код программы приведен на листинге 11.3.

Листинг 11.3. Быстрая сортировка (набросок)

```
void qsort(l, u)
  if l >= u then
    /* не более одного элемента - ничего не делаем */
    return
  /* цель: разбить массив относительно какого-либо элемента, который
оказывается на правильном месте */
  qsort(l, p-1)
  qsort(p+1, u)
```

Для разбиения массива относительно значения t мы начнем с простой схемы, о которой я узнал от Нико Ломуто. Более быстрый вариант программы, выполняющий эту задачу, будет приведен в следующем разделе[1], но эта функция так проста, что в ней трудно ошибиться, и она ни в коем случае не может быть названа медленной. Пусть дано значение t, нужно упорядочить массив x[a..b] и вычислить индекс m такой, что все элементы, меньшие t, находятся слева от элемента x[m], а все большие — справа. Мы решим эту задачу с помощью простого цикла for, сканирующего массив слева направо, используя переменные i и m для сохранения инварианта (рис. 11.3).

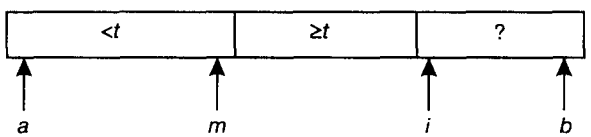


Рис. 11.3. Инвариант цикла сканирования в алгоритме быстрой сортировки

Когда проверяется i-й элемент, необходимо рассмотреть две ситуации. Если x[i] >= t, то инвариант остается истинным. Если же x[i] < t, то восстановить инвариант можно, увеличив индекс m (который указывает новое местоположение меньшего элемента) и поменяв местами x[i] и x[m]. В итоге получаем код, осуществляющий разбиение массива:

```
m = a-1
for i = [a, b]
  if x[i] < t
    swap(++m, i)
```

[1] В большинстве реализаций быстрой сортировки используется двустороннее разделение, описанное в следующем разделе. Хотя основная идея этого кода проста, детали реализации всегда казались мне весьма сложными — однажды я провел почти два дня в поисках ошибки в коротком цикле разбиения. Один из читателей черновика этой статьи пожаловался, что стандартный метод на самом деле проще, чем метод Ломуто, и в подтверждение этому набросал код программы. Я прекратил изучение этой программы, найдя две ошибки.

В алгоритме Quicksort нам нужно разбить массив x[l..u] относительно значения t=x[l], так что a = l + 1, a b = u. Инвариант цикла разбиения можно будет изобразить следующим образом (рис. 11.4).

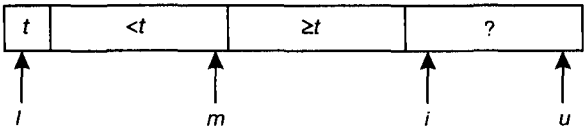


Рис. 11.4. Инвариант цикла разбиения массива

После завершения цикла получим картину, изображенную на рис. 11.5.

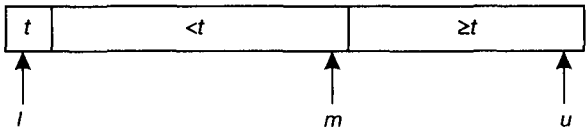


Рис. 11.5. Состояние массива после завершения цикла разбиения

Затем мы поменяем местами x[l] и x[m], что даст рис. 11.6[1].

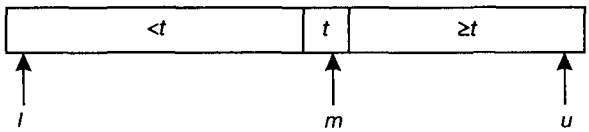


Рис. 11.6. Состояние после перестановки элементов

Теперь можно рекурсивно вызвать функцию с параметрами (l, m-1) и (m+1, u).

Получившийся код дает первую завершенную версию быстрой сортировки. Он приведен в листинге 11.4. Для сортировки массива x[n] функцию следует вызвать как qsort (0, n-1).

Листинг 11.4. Быстрая сортировка: версия 1

```
void qsort(l, u)
  if (l >= u)
    return
  m = l
  for i = [l+1, u]
    /* инвариант: x[l+1..m] < x[l] &&
    x[m+1..i-1] >= x[l] */
    if (x[i] < x[l])
      swap(++m, i)
  swap(l, m)
```

[1] Кажется соблазнительным пропустить этот этап и вызвать рекурсию с параметрами (l, m) и (m+1, u). К сожалению, при этом мы попадаем в бесконечный цикл, если t является наибольшим элементом массива. Возможно, я бы обнаружил эту ошибку в процессе проверки условия завершения, но читатель может догадаться, как я в действительности на нее наткнулся. М. Джейкоб дал элегантное доказательство некорректности такой программы: поскольку x[l] никогда не переставляется, эта сортировка будет работать только в том случае, если x[0] будет минимальным элементом массива.

```
/* x[l..m-1] < x[m] <= x[m+1, u] */
qsort1(l, m-1)
qsort1(m+1, u)
```

В задаче 2 в конце главы описана модификация алгоритма разбиения, предложенная Бобом Седжвиком (Bob Sedgewick). Она дает несколько более быстрый алгоритм qsort2.

Правильность этой программы была практически полностью доказана при ее выводе (как и должно быть). Доказательство ведется по индукции. Внешний оператор if правильно обрабатывает пустые и одноэлементные массивы, а алгоритм разбиения правильно подготавливает массивы большего размера для последующих рекурсивных вызовов. Программа не может войти в бесконечную последовательность рекурсивных вызовов, поскольку элемент x[m] при каждом таком вызове исключается. Таким же образом была доказана конечность алгоритма в разделе 4.3 (двоичный поиск).

Программа быстрой сортировки работает за время $O(n \log n)$ и требует в среднем $O(\log n)$ памяти на стеке. Под «средним» случаем подразумевается случайная перестановка не равных друг другу элементов, поступающая на вход алгоритма. Математические аргументы в пользу этого вывода аналогичны алгоритмам, приведенным в разделе 8.3. В большинстве учебников по теории алгоритмов подробно анализируется время выполнения быстрой сортировки. Кроме того, в них доказывается, что любая основанная на сравнении сортировка не может выполнить менее $O(n \log n)$ сравнений, поэтому быстрая сортировка наиболее близка к идеалу.

Функция qsort1 — это самая простая среди известных мне версия алгоритма быстрой сортировки. Она иллюстрирует важнейшие свойства этого алгоритма. Во-первых, он действительно быстр: в моей системе миллион случайных целых чисел упорядочивается чуть больше, чем за секунду, — вдвое быстрее, чем хорошо отлаженная библиотечная функция qsort языка C (последняя предназначена для сортировки различных типов данных, поэтому оказывается более медленной). Эта сортировка может быть удобной для некоторых приложений с хорошими входными данными, но она обладает и другим характерным свойством быстрых сортировок: для некоторых типов входных данных время ее работы может быть квадратичным. В следующем разделе рассмотрены более робастные быстрые сортировки.

11.3. Улучшенные быстрые сортировки

Функция qsort1 быстро сортирует массив случайных чисел, но что, если на вход будет подана уже упорядоченная последовательность? Как мы видели в разделе 2.4 главы 2, программисты часто используют сортировку для того, чтобы одинаковые элементы оказались рядом. Следовательно, нужно рассмотреть крайний случай: массив из n одинаковых элементов. Сортировка вставкой работает на таких данных очень быстро: каждый элемент сдвигается на 0 позиций, поэтому время выполнения растет как $O(n)$. Функция qsort1 справляется с такими данными очень плохо. Каждое из n-1 разбиений требует время $O(n)$ для выделения одного эле-

мента, поэтому полное время выполнения растет как $O(n^2)$. Время обработки для n = 1 000 000 возрастает с одной секунды до двух часов.

Можно обойти эту проблему, используя двусторонний алгоритм разбиения с приведенным на рис. 11.7 инвариантом.

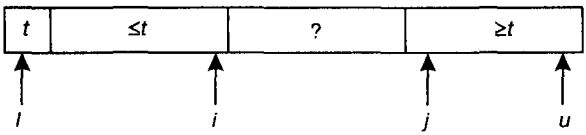


Рис. 11.7. Инвариант двустороннего разбиения

Индексы i и j инициализируются граничными индексами разбиваемого массива. Главный цикл содержит два вложенных цикла. Первый вложенный цикл сдвигает i вверх, пропуская меньшие элементы, а второй увеличивает j, пропуская большие элементы и останавливаясь на меньшем. Главный цикл проверяет, не пересекаются ли эти индексы, и переставляет соответствующие элементы.

Но как такой код будет работать в ситуации, когда все элементы равны? Первая мысль: пропустить эти элементы, чтобы не делать лишней работы, но в результате получается квадратичный для массива из одинаковых элементов алгоритм. Поэтому каждое сканирование будет останавливаться на одинаковых элементах, которые затем будут обмениваться. Хотя в этом варианте обменов будет производиться больше, чем требуется, такая программа будет превращать худший случай с массивом из одинаковых элементов в лучший, требующий почти в точности $n \log_2 n$ сравнений. Программа, реализующая описанный алгоритм разбиения, приведена в листинге 11.5.

Листинг 11.5. Быстрая сортировка (версия 3)

```
void qsort3(l, u)
  if l>= u
    return
  t = x[l], i = l; j = u+1
  loop
    do i++ while i <= u && x[i] < t
    do j-- while x[j] > t
    if i > j
      break
    swap(i, j)
  swap(l, j)
  qsort3(l, j-1)
  qsort3(j+1, u)
```

Избавляясь от квадратичного поведения в худшем случае, этот код и в среднем делает меньше обменов, чем qsort1.

Рассмотренные нами программы быстрой сортировки разбивали массив относительно первого встреченного элемента. Это хорошо подходит для случайных входных данных, но может сильно замедлить работу для некоторых упорядоченных последовательностей. Если массив уже отсортирован по возрастанию, его придется разбивать относительно первого элемента, затем относительно второго и так

далее, что потребует времени $O(n^2)$. Мы можем избежать этого, выбирая элемент для разбиения случайным образом — обменивая местами элемент x[l] со случайным элементом из диапазона x[l..u]:

```
swap(l, randint(l, u))
```

Если у вас нет функции randint, обратитесь к задаче 2 из главы 12 данной книги, которая посвящена написанию собственного генератора случайных чисел. Каким бы кодом вы ни пользовались, внимательно проследите за тем, чтобы функция randint возвращала значение из диапазона [l, u] — выход за его границы приведет к ошибкам. Объединив случайный выбор центрального элемента с двусторонним разбиением, мы получим программу быстрой сортировки, работающую за время $O(n \log n)$ для любого входного массива. Усреднение делается вызовом генератора случайных чисел, а не анализом возможного распределения входных данных.

Наша программа быстрой сортировки бóльшую часть времени тратит на сортировку очень маленьких подмножеств. Такие массивы было бы проще всего сортировать каким-либо несложным методом вроде сортировки вставкой, а не тратить на них всю мощь быстрой сортировки. Боб Седжвик разработал весьма хитроумную реализацию этой идеи. Когда функция быстрой сортировки вызывается для небольшого массива (то есть l и u близки), она не делает ничего. Реализуется это путем замены первого оператора if нашей функции на следующий код:

```
if u-l> cutoff
  return
```

Здесь cutoff — некоторое небольшое целое число. После завершения работы функции массив будет не отсортирован до конца, но разбит на небольшие группы случайно упорядоченных элементов, причем все элементы одной группы будут меньше любого элемента всех групп, расположенных справа от данной. Сортировать элементы внутри групп нужно каким-то другим методом, и тут лучше всего подходит сортировка вставкой, поскольку массив уже почти упорядочен.

Для решения задачи сортировки целиком придется выполнить два вызова:

```
qsort4(0, n-1)
isort3()
```

Задача 3 посвящена выбору оптимальной величины порога cutoff.

На последнем этапе оптимизации программы можно раскрыть вызов функции swap во внутреннем цикле (поскольку другие два вызова swap лежат вне внутреннего цикла, их раскрытие не даст ощутимого результата). Последняя версия программы Quicksort приведена в листинге 11.6.

Листинг 11.6. Быстрая сортировка: версия 4 (итоговая)

```
void qsort4(l, u)
  if u - l < cutoff
    return
  swap(l, randint(l, u))
  t = x[l], i = l, j = u+1
  loop
```

```
    do i++ while i <= u && x[i] < t
    do j-- while x[j] > t
    if i > j
      break
    temp = x[i]; x[i] = x[j]; x[j] = temp
  swap(l, j)
  qsort4(l, j-1)
  qsort4(j+1, u)
```

Задачи 4 и 11 посвящены дальнейшему улучшению производительности быстрой сортировки.

В табл. 11.2 приведены сводные данные по всем версиям быстрой сортировки. Правая колонка указывает среднее время работы в наносекундах, требуемое для сортировки массива из n случайных целых чисел. Многие алгоритмы могут вести себя как квадратичные для некоторых конкретных входных данных.

Таблица 11.2. Быстрая сортировка

Программа	Объем кода (строк языка C)	Время, нс
Библиотечная функция языка C qsort	3	$137n\log_2 n$
Быстрая сортировка 1	9	$60n\log_2 n$
Быстрая сортировка 2	9	$56n\log_2 n$
Быстрая сортировка 3	14	$44n\log_2 n$
Быстрая сортировка 4	15+5	$36n\log_2 n$
Библиотечная функция C++ sort	1	$30n\log_2 n$

Функция qsort состоит из 15 строк быстрой сортировки и 5 строк сортировки вставкой. Для миллиона случайных чисел время выполнения лежит в диапазоне от 0,6 с для библиотечной функции sort языка C++ до 2,7 с для библиотечной функции qsort языка C. В главе 14 мы изучим алгоритм, гарантированно сортирующий n целых чисел за $O(n\log n)$ для любых входных данных (даже в худшем случае).

11.4. Принципы

Это упражнение дает нам несколько ценных уроков о сортировке в частности и программировании вообще.

Библиотечная функция qsort

Библиотечная функция qsort проста в использовании и работает достаточно быстро. Она работает медленнее, чем самодельные версии быстрой сортировки, только потому, что предназначена для применения к различным типам данных, поэтому сравнение элементов осуществляется через вызов внешней функции. Интерфейс функции sort языка C++ существенно проще: массив x сортируется вызовом sort(x, x+n). Кроме того, она достаточно эффективно реализована. Если системная

сортировка отвечает вашим требованиям, не задумывайтесь и отметайте идею написания своей собственной.

Сортировка вставкой

Сортировку вставкой легко написать, и она достаточно эффективна для небольших задач. Сортировка 10 000 целых чисел с помощью этого алгоритма требует на моем компьютере всего лишь треть секунды.

Случай больших n

Для больших n жизненно важным оказывается время выполнения алгоритма быстрой сортировки $O(n \log n)$. Методы разработки алгоритмов из главы 8 дали нам идею (принцип «разделяй и властвуй»), а верификация, обсуждавшаяся в главе 4, позволила нам реализовать эту идею в ясном и эффективном коде.

Хотя значительное увеличение производительности обычно достигается сменой алгоритма, методы оптимизации из главы 9 ускорили работу сортировки вставкой в 4 раза, а быстрой сортировки — в 2 раза.

11.5. Задачи

1. Сортировка, как и любое другое мощное средство, часто используется там, где ее не следует использовать, и не используется там, где без нее не обойтись. Объясните, как можно недооценить или переоценить важность сортировки при получении статистических данных (минимум, максимум, среднее, медиана и мода распределения) для некоторого массива вещественных чисел.
2. [R. Sedgewick] Ускорьте схему разбиения Ломуто (Lomuto), используя элемент `x[l]` в качестве маркера. Покажите, как эта функция позволяет избавиться от вызова `swap` после цикла.
3. Какими экспериментами можно найти оптимальное значение `cutoff` в некоторой системе?
4. Хотя в среднем быстрая сортировка требует $O(\log n)$ памяти на стеке, в худшем случае может потребоваться объем памяти $O(n)$. Объясните, откуда проистекает этот недостаток, и постарайтесь избавиться от него.
5. [M. D. McIlroy] Покажите, как можно использовать схему разбиения Ломуто для сортировки битовых строк переменной длины за время, пропорциональное сумме их длин.
6. Используйте методы, описанные в этой главе, для реализации других алгоритмов сортировки. Сортировка выбором помещает наименьший элемент в `x[0]`, затем наименьший из оставшихся в `x[1]` и так далее. Сортировка Шелла (с уменьшающимся инкрементом) работает аналогично сортировке вставкой, но элементы сдвигаются на `h` позиций, а не на одну. Начальное значение `h` затем уменьшается в процессе работы

7. Реализации программ сортировки из этой главы можно найти на веб-сайте книги. Измерьте время их работы в вашей системе и сведите результаты в таблицу, аналогичную табл. 11.2.
8. Набросайте руководство пользователю вашей системы по выбору метода сортировки. Метод должен учитывать важность времени выполнения, требуемой памяти, времени программиста (затрачиваемого на разработку и обслуживание программы), общность (что, если нужно отсортировать символьные строки, в которых записаны римские числа?), стабильность (элементы с одинаковыми ключами должны оставаться в исходном порядке), особенности входных данных и другие. В качестве теста можете испробовать это руководство на задаче из главы 1.
9. Напишите программу нахождения k-го минимума в массиве `x[0..n-1]` за время $O(n)$. Допускается перестановка элементов.
10. Проведите эксперименты и постройте диаграмму времени работы быстрой сортировки на различных входных данных.
11. Напишите функцию разбиения с «широким главным элементом», постусловие которой будет таким:

$<t$	$=t$	$>t$

Как можно включить такую функцию в программу быстрой сортировки?

12. Изучите методы сортировки, используемые в обычной жизни (сортировка почты, разменные автоматы и другие).
13. Программы быстрой сортировки из этой главы выбирают центральный элемент случайным образом. Изучите возможные варианты, включая выбор медианы по некоторому подмножеству исходного массива.
14. Программы быстрой сортировки, рассматриваемые в данной главе, описывают подмножество с помощью двух целочисленных индексов. Это необходимо в языках типа Java, в которых указатели на элементы массивов отсутствуют. В языках C и C++ можно отсортировать массив целых чисел, используя приведенный ниже формат вызова функции:

```
void qsort(int x[], int n)
```

для исходного и всех последующих рекурсивных вызовов. Измените алгоритмы этой главы, ориентируясь на использование данного интерфейса.

11.6. Дополнительная литература

С момента выхода первого издания этой книги Addison Wesley в 1973 году, самым популярным справочником на эту тему стала книга Д. Кнута «Искусство программирования для ЭВМ, том 3: сортировка и поиск». В ней подробно описываются все основные алгоритмы, проводится математический анализ времени их работы. В книге также приведены примеры реализации алгоритмов на ассемблере. Пред-

лагаемые упражнения и ссылки на другие книги содержат важные модификации исходных алгоритмов. К моменту выхода второго издания в 1998 году Кнут обновил и отредактировал книгу. Используемый им ассемблер MIX устарел, но принципы разработки, иллюстрируемые кодом, бессмертны.

Третье издание книги Боба Седжвика «Алгоритмы» (Bob Sedgwick, Algorithms) содержит описание более современных подходов к сортировке и поиску. Части 1–4 посвящены «Основам», «Структурам данных», «Сортировке» и «Поиску». Книга «Алгоритмы на C» была опубликована издательством Addison Wesley в 1997 году, «Алгоритмы на C++» (в соавторстве с Крисом ван Вайком) — в 1998 году, а «Алгоритмы на Java» (в соавторстве с Тимом Линдхолмом) вышла в 1999 году. Упор делается на реализацию алгоритмов на различных языках программирования (по вашему выбору), тогда как описание их производительности дается на интуитивном уровне.

Книги этих двух авторов являются основными руководствами по сортировке, поиску (глава 13) и «кучам» (глава 14).

Задача о выборке

Небольшие компьютерные программы часто оказываются полезными в образовательном и развлекательном планах. В этой главе рассказывается история создания маленькой программки, которая, помимо упомянутых качеств, оказалась еще и очень полезной для фирмы.

12.1. Задача

Дело было в начале 80-х. Компания только что приобрела первые персональные компьютеры. Установив и настроив основные программы, я рекомендовал сотрудникам искать возможности автоматизации офисных задач с помощью новых машин. Фирма занималась опросами общественного мнения, и одна из сотрудниц предложила автоматизировать задачу выбора случайных представителей из списка избирательных участков. Поскольку решение задачи вручную требовало часа работы с таблицей случайных чисел, сотрудница предложила следующую программу:

На вход подается список названий участков и целое число m. *На выходе должен получиться список из* m *случайно выбранных избирательных участков. В исходном списке обычно присутствует несколько сотен названий (каждое из которых представляет собой строку длиной не более десятка символов), а* m *обычно лежит в интервале* 20..40.

Так представляла себе программу одна из пользователей. Хотите ли вы высказать какие-нибудь предложения по формулировке задачи до того, как мы попытаем ее реализовать программно?

Я ответил сотруднице, что это замечательная идея. Задача отлично подходила для автоматизации. Затем я отметил, что набрать несколько сотен имен, возможно, проще, чем работать с большой таблицей случайных чисел, но это все равно утомительное занятие, в процессе которого легко наделать ошибок. Да и вообще глупо вводить кучу данных, если все равно бóльшая их часть использована не будет. Поэтому я предложил поставить задачу в альтернативной форме.

На вход программы подается два целых числа: m *и* n, *причем* m<n. *На выходе должен получаться список из* m *случайных целых чисел в диапазоне* 0..n-1[1], *причем никакое число не должно встретиться дважды. С точки зрения теории вероятностей, мы должны реализовать выбор без возврата, причем вероятность выбора всех элементов должна быть одинакова.*

Для m = 20 и n = 200 программа должна выдать список из 20 чисел, который может начинаться, к примеру, так: 4, 15, 17, ... Пользователь может выбрать 20 случайных названий из списка избирательных округов, в котором этих округов 200, отсчитывая их от начала списка и помечая 4, 15, 17-е и последующие. Числа на выходе должны быть отсортированы, потому что список на бумаге не пронумерован.

Эта спецификация была встречена с энтузиазмом со стороны потенциальных пользователей. После написания программы они получали бы возможность за несколько минут делать то, на что раньше требовался бы час.

Взгляните на задачу с другой стороны: как бы вы ее решили? Предположим, у вас есть функция bigrand(), возвращающая большое случайное целое число (много большее m и n) и функция randint(i,j), возвращающая случайное число из интервала i..j (с однородным распределением). Задача 1 посвящена реализации таких функций.

12.2. Одно из решений

Как только мы выяснили детали задачи, которую предстояло решить, я побежал за томом «Получисленных алгоритмов» Кнута (иметь дубликаты всех трех томов Кнута дома и на работе стоит затраченных денег). Поскольку за некоторое время до того я изучил эту книгу весьма внимательно, мне смутно припоминались алгоритмы для решения подобных задач. Потратив несколько минут на изучение нескольких подходящих алгоритмов решения задачи, я понял, что алгоритм S в разделе 3.4.2 был идеальным для решения моей задачи.

В алгоритме последовательно рассматривается каждое из чисел 0, 1, 2, ..., n-1, и каждое из них может быть выбрано, если оно случайным образом пройдет некоторую проверку. Последовательным перебором чисел мы гарантируем, что результат окажется отсортирован.

Чтобы понять критерий отбора, рассмотрим пример с m = 2 и n = 5. Мы выбираем первое число 0 с вероятностью 2/5. Программно это реализуется следующим оператором:

```
if (bigrand() % 5) < 2
```

К сожалению, мы не можем установить ту же вероятность для числа 1, потому что в этом случае мы можем и не получить 2 числа из пяти (а можем и получить). Поэтому мы несколько изменим ситуацию, установив для единицы вероятность выбора 1/4, если 0 был выбран, и 2/4, если 0 не был выбран. В общем, чтобы получить s элементов из r оставшихся, мы будем выбирать очередной элемент с вероятностью s/r. Вот программа на псевдокоде:

---

[1] На самом деле программа выводила числа в диапазоне 1..n. Я изменил этот диапазон для единообразия и для того, чтобы с помощью этой программы можно было порождать массивы языка C. Программисты могут начинать отсчет с нуля, но социологи начинают с единицы.

```
select = m
remaining = n
for i = [0, n)
  if (bigrand() % remaining) < select
    print i
    select--
  remaining--
```

Пока m <= n программа всегда выбирает ровно m чисел. Большее количество выбрано быть не может, поскольку вероятность выбора очередного элемента становится нулевой, как только m чисел уже набрано. Меньше их тоже оказаться не может, потому что когда отношение требуемых к оставшимся становится равным 1, все оставшиеся числа однозначно проходят тест. Оператор for гарантирует, что числа будут выводиться в нужном порядке. Моего описания вам должно быть достаточно, чтобы поверить в равновероятность выбора любого набора чисел. Кнут доказывает это с помощью теории вероятностей.

Второй том Кнута очень облегчил мне задачу. Даже вместе с заголовком, вводом, выводом, проверкой ограничений и тому подобное, конечный вариант программы занимал всего 13 строк на языке BASIC. Он был готов спустя полчаса после постановки задачи и использовался много лет без каких-либо проблем. Вот эта программа на языке C++:

Листинг 12.1 Программа 1

```
void genknuth(int m, int n)
{  for (int i = 0; i < n; i++)
    /* выбор m из оставшихся n - i */
    if ((bigrand() % (n-i)) < m) {
      cout << i << "\n";
      m--;
    }
}
```

Программе требовалось всего несколько десятков байт памяти, и она была быстрой как молния (с учетом задач фирмы). Однако этот код может показаться медленным, если n будет слишком велико. Получение нескольких случайных 32-разрядных целых чисел ($n = 2^{32}$) требует на моем компьютере около 12 минут. Задачка на предварительные оценки: сколько времени потребуется на получение одного 48- или 64-разрядного целого с помощью этого алгоритма?

12.3. Пространство разработки

Часть работы программиста состоит в том, чтобы решать сегодняшние задачи. Другая, возможно, более важная, состоит в подготовке к решению завтрашних задач. Иногда подготовка заключается в прохождении курсов или чтении книг вроде трудов Кнута. Но чаще программисты учатся самостоятельно. Попробуем сделать это сейчас — исследуем пространство возможных решений этой задачи.

Рассказав о задаче на семинаре в Вестпойнтском училище, я попросил предложить лучший подход (по сравнению с исходным — вводить 200 названий вручную). Один из курсантов предложил скопировать исходный список, разрезать его с помощью машинки для нарезки бумаги, перемешать кусочки в коробке и вытащить

наугад требуемое их количество. Этот кадет продемонстрировал пример «концептуального прорыва», описанного в книге Адамса, цитируемой в разделе 1.7[1].

Теперь мы займемся исключительно программой, позволяющей получить m упорядоченных случайных целых чисел из диапазона 0..n-1. Начнем мы с исследования программы 12.1. Идея алгоритма прямолинейна, программа короткая, памяти используется совсем мало, а время выполнения отлично подходит для данного приложения. Однако это время выполнения пропорционально n, что может быть недопустимо для каких-либо других приложений. Имеет смысл потратить несколько минут на поиск другого решения той же задачи. Прежде чем продолжить свое ознакомление с данной главой, набросайте как можно больше схем алгоритмов высокого уровня, не заботясь о деталях реализации.

Одно из возможных решений состоит в помещении случайных целых чисел в изначально пустой набор до тех пор, пока в нем есть место. На псевдокоде он может быть записан так:

```
инициализация набора S пустым множеством
size = 0
while size < m do
  t = bigrand() % n
  if t is not in S  /* t не принадлежит S */
    insert t into S  /* вставка элемента в набор */
    size++
вывод элементов S в порядке возрастания
```

В этом алгоритме нет выделенных элементов, поэтому результат его работы оказывается вполне случайным. Однако остается проблема реализации набора S. Подумайте о том, какая структура данных могла бы для этого использоваться.

Раньше мне пришлось бы поразмыслить о сравнительных характеристиках отсортированных связных списков, двоичных деревьях и других возможных структурах данных. Сегодня у меня есть возможность воспользоваться чужим трудом, вложенным в стандартную библиотеку шаблонов C++, и назвать набор набором[2] (set).

Листинг 12.2. Программа 2

```
void gensets(int m, int n)
{  set<int> S;
  while (S.size() < m)
    S.insert(bigrand() % n);
  set<int>::iterator i;
  for (i = S.begin(); i != S.end(); ++i)
    cout << *i << "\n";
}
```

Написав эту программу, я с удовольствием обнаружил, что длина псевдокода совпадает с длиной реальной программы. Эта версия формирует и выводит миллион упорядоченных 32-разрядных случайных целых примерно за 20 секунд. По-

[1] На с. 57 книги Адамса описывается три типа озарений. Ах-озарения относятся к оригинальным вещам, ага-озарения — к открытиям. Решение кадета автор книги назвал бы ха-ха-озарением. Такое озарение — это простой ответ на сложный вопрос, отчасти шуточный (как в решениях 1.10, 1.11 и 1.12).

[2] Многие программисты называют этот декларатор *множеством*. В данной книге употребляются оба перевода, в зависимости от того, какое значение более уместно в контексте. — *Примеч. ред.*

сколько на формирование и вывод миллиона неупорядоченных целых чисел требуется примерно 12,5 секунды, на операции с набором тратится примерно 7,5 секунды.

Спецификация C++ STL гарантирует, что любая операция вставки будет выполняться за время $O(\log m)$, а перебор элементов списка потребует времени $O(m)$, так что вся программа будет работать за время $O(m \log m)$, где m мало по сравнению с n. Однако эта структура данных требовательна к памяти: мой компьютер со 128 Мбайт ОЗУ начинает обращаться к диску, когда m достигает величины порядка 1 700 000. В следующем разделе рассматриваются возможные реализации набора.

Другой способ получения упорядоченного набора целых чисел заключается в перемешивании n-элементного массива с числами 0..n-1 и сортировке его первых m элементов, которые затем могут быть выведены. Алгоритм P из раздела 3.4.2 монографии Кнута предназначен для перемешивания массива x[0..n-1].

```
for i = [0, n)
  swap(i, randint(i, n-1))
```

Эшли Шепхерд (Ashley Shepherd) и Алекс Воронов (Alex Woronow) пришли к выводу, что в этой задаче нужно перемешать только первые m элементов массива. В итоге получаем следующую программу на C++:

Листинг 12.3 Программа 3

```
void genshuf(int m, int n)
{  int i,j;
  int *x = new int[n];
  for (i = 0; i < n; i++)
    x[i] = i;
  for (i = 0; i < m; i++) {
    j = randint(i, n-1);
    int t = x[i]; x[i] = x[j]; x[j] = t;
  }
  sort(x, x+m)
  for (i = 0; i < m; i++)
    cout << x[i] << "\n";
}
```

Данный алгоритм использует n слов памяти и работает за время $O(n + m \log m)$, но метод, примененный в задаче 9 из главы 1, уменьшает время выполнения до $O(m \log m)$. Алгоритм можно рассматривать в качестве альтернативы программе 2, в которой набор выбранных элементов хранится в x[0..i-1], а оставшиеся элементы помещаются в x[i..n-1]. Явно представляя невыбранные элементы, мы исключаем необходимость проверки, был ли уже выбран ранее данный элемент. К сожалению, этот метод работает за время $O(n)$ и требует такого же количества памяти, поэтому обычно он уступает алгоритму Кнута.

Мы рассмотрели несколько вариантов решения задачи, но описанные программы ни в коей мере не покрывают все пространство разработки. Предположим, что n = 1 000 000, а m = n – 10. Мы можем сформировать отсортированный набор из 10 элементов, а потом выдать пользователю все остальные элементы основного множества, которые в этот набор не вошли. Предположим, что n = 10 000 000, а $m = 2^{31}$. Можно сгенерировать 11 миллионов целых чисел, отсортировать их, удалить повторяющиеся элементы, а затем выбрать из них 10 миллионов и отсортировать их. В решении задачи 9 описан хитроумный алгоритм Боба Флойда, основанный на поиске.

12.4. Принципы

В этой главе я постарался продемонстрировать некоторые важные этапы процесса написания программ. Хотя ниже эти этапы перечислены в естественном порядке, разработка обычно идет более активно: программист переходит от одного вида деятельности к другому, повторяя каждый из них много раз, до тех пор, пока не находит приемлемое решение.

Понимание предложенной задачи

Поговорите с пользователем об истории возникновения задачи. Уже сама постановка задачи часто содержит идеи, ведущие к ее решению. Эти идеи следует иметь в виду наравне со всеми прочими, возникающими по ходу работы.

Постановка абстрактной задачи

Ясная и четкая постановка задачи помогает решить ее и затем успешно применять полученный результат к решению других задач.

Исследование пространства разработки

Очень многие программисты слишком рано переходят к «идеальному» решению своих задач. Они размышляют несколько минут, а потом программируют целый день, вместо того чтобы подумать час, а потом час программировать. Использование неформальных языков высокого уровня помогает описывать проекты программ; псевдокод определяет последовательность выполнения программы, а абстрактные типы данных представляют основные структуры. На этом этапе процесса разработки знания, почерпнутые из книг, могут оказаться бесценными.

Реализация одного из решений

Если вам повезет, исследование пространства разработки может показать, что одно из решений гораздо предпочтительнее, чем прочие. Во всех остальных случаях придется создавать прототипы нескольких решений и выбирать из них. Нужно стремиться реализовать код по возможности просто и прямолинейно, используя самые мощные доступные средства[1].

Оглядывайтесь назад

Превосходная книга Полья «Как это решить» (Polya, How to Solve It) подойдет любому программисту, стремящемуся к дальнейшему совершенствованию своих

[1] В задаче 6 описано упражнение, которое я оценивал по стилю программирования. Большинство решений было объемом в 1 страницу. За такие решения я ставил среднюю оценку. Два студента, которые в предыдущее лето участвовали в большом программном проекте, сдали отлично документированные программы на 5 страниц, содержавшие десяток функций со сложными заголовками. Они получили низкие оценки. Лучшая программа была всего из пяти строк, поэтому мне трудно было засчитать в шестьдесят раз большую. Когда эти студенты пожаловались, что они просто использовали стандартные средства инженерии программного обеспечения, мне следовало бы процитировать Пэмелу Зейв: «*Задача инженерии программного обеспечения в том, чтобы регулировать сложность, а не увеличивать ее*». Несколько минут, потраченных на поиск короткого решения, могли бы сберечь им часы, потраченные на документирование их программ.

способностей в решении задач. На странице 15 автор отмечает, что *«всегда остается что-то несделанное; при соответствующем анализе и достаточной проницательности можно улучшить любое решение, и в любом случае можно улучшить степень понимания любого решения»*. Его советы особенно полезны, когда оглядываешься назад, на только что решенную задачу.

12.5. Задачи

1. Библиотечная функция языка C rand() в типичной реализации возвращает 15 случайных битов. Используйте ее для реализации функции bigrand(), возвращающей по меньшей мере 30 случайных битов, и функции randint(l, u), возвращающей случайное целое в диапазоне [l, u].
2. В разделе 12.1 было отмечено, что вероятность выбора всех m-элементных наборов должна быть одинаковой. Это более сильное требование, чем просто требование выбора любого целого с вероятностью m/n. Опишите алгоритм, в котором вероятность выбора всех элементов одинакова, но при этом вероятности выбора разных наборов элементов различны.
3. Покажите, что если m < n/2, то ожидаемое количество проверок на принадлежность набору до нахождения отсутствующего элемента в алгоритме, использующем наборы, меньше двух.
4. Попытка подсчета количества проверок на принадлежность в программе с наборами порождает много новых интересных задач на комбинаторику и теорию вероятностей. Сколько проверок в среднем делается в зависимости от чисел m и n? Сколько их будет сделано, если m = n? В какой ситуации наиболее вероятно проведение более m проверок?
5. В этой главе приведено несколько алгоритмов решения одной задачи. Их реализации можно найти на сайте этой книги. Измерьте производительность этих алгоритмов в своей системе и сделайте выводы об условиях применимости того или иного алгоритма (ограничения на время выполнения, объем памяти и другие).
6. Я предлагал студентам задачу о формировании упорядоченных подмножеств в рамках курса теории алгоритмов, причем делал это два раза. Первый — когда они еще не изучали поиск и сортировку. Студенты должны были написать программу для m = 20 и n = 400. Главным критерием оценки была ясность и краткость кода — время его работы роли не играло. После курса сортировки и поиска им нужно было решить ту же задачу еще раз, но теперь m = 5 000 000, а n = 1 000 000 000, причем оценка зависела в первую очередь от времени работы программы.
7. [V. A. Vyssotsky] Алгоритмы генерации комбинаторных объектов чаще всего выгодно выражать через рекурсивные функции. Алгоритм Кнута может быть записан следующим образом:

```
void randselect(m, n)
    предусловие: 0 <= m <= n
    постусловие: m неповторяющихся чисел от 0 до n-1 выводятся
            в порядке убывания
```

```
if m > 0
  if (bigrand() % n) < m
    print n-1
    randselect(m-1, n-1)
  else
    randselect(m, n-1)
```

Эта программа выводит числа в порядке убывания. Как можно сделать, чтобы они выводились в порядке возрастания? Докажите правильность получившейся программы. Как можно использовать основную рекурсивную структуру программы для получения всех m-элементных подмножеств диапазона 0..n-1?

8. Как бы вы реализовали случайный выбор m целых чисел из диапазона 0..n-1, если нужно было бы их выводить в случайном порядке? Как бы вы получали отсортированный список, если бы было допустимо присутствие в нем повторяющихся чисел? Что, если бы нужно было одновременно получить случайный порядок элементов, и при этом допускалось бы их повторение?
9. [R. W. Floyd] Когда m близко к n, алгоритмы, основанные на использовании наборов, генерируют большое количество чисел, которые затем отбрасываются, поскольку уже имеются в наборе. Можете ли вы придумать алгоритм, использующий только m случайных чисел, даже в худшем случае?
10. Как можно выбрать один из n объектов случайным образом, если объекты предъявляются последовательно, но их количество n заранее неизвестно? Поставлю задачу конкретнее: нужно напечатать одну случайную строку текстового файла, прочитав его всего один раз, причем количество строк заранее неизвестно, а вероятность выбора должна быть одинакова для всех строк текста.
11. [M. I. Shamos] Лотерея проводится с помощью карты с шестнадцатью точками, под которыми случайным образом распределены числа 1..16. Игрок стирает точки, открывая скрытые под ними числа. Если открывается число 3, карта проигрывает. Если открываются числа 1 и 2 (в любом порядке), карта выигрывает. Опишите, каким образом вы могли бы с помощью компьютера вычислить вероятность выигрыша при случайном выборе точек. На решение задачи вам отводится 1 час компьютерного времени.
12. Моя первая версия одной из программ этой главы содержала ошибку, приводившую к ее досрочному завершению при m = 0. Для прочих значений m она выводила числа, казавшиеся случайными, но на деле таковыми не являвшиеся. Как бы вы проверили программу, выводящую некоторый набор случайных чисел, на случайность результата?

12.6. Дополнительная литература

Том 2 монографии Д. Кнута «Искусство программирования для ЭВМ» называется «Получисленные алгоритмы». Глава 3 (в первой половине книги) посвящена случайным числам, а глава 4 (во второй половине книги) — арифметике. Раздел 3.4.2 «Случайная выборка и перемешивание» имеет непосредственное отношение к предмету обсуждения данной главы. Если вам понадобится написать собственный генератор случайных чисел или функции для работы со сложной арифметикой, — эта книга окажется незаменимой.

Поиск

Задача поиска встречается во множестве приложений. Компилятор ищет имя переменной, чтобы определить ее тип и адрес. Программа проверки правописания должна найти слово в словаре, чтобы проверить его правильность. Программа, обслуживающая телефонный справочник, ищет в нем имя абонента, чтобы определить его номер. Сервер имен доменов Интернет ищет имя запрашиваемого узла в своей базе, определяя по этому имени IP-адрес. В этих примерах, как и во множестве других, приходится осуществлять поиск в наборе данных для получения информации, содержащейся в одном из элементов набора.

В этой главе подробно рассматривается одна из задач, связанных с поиском, а именно: каким образом следует хранить в памяти набор целых чисел без каких-либо дополнительных, связанных с этими числами, данных? Хотя эта проблема кажется не слишком значительной, ход ее решения во многом аналогичен решению задачи реализации произвольного типа данных. Мы начнем с точной формулировки задачи и с ее помощью исследуем наиболее общие способы представления множеств данных.

13.1. Интерфейс

Продолжим решать задачу из предыдущей главы: нам нужно было получить упорядоченную последовательность m случайных чисел из диапазона [0, макс.), причем выбор осуществляется без возврата (одинаковых чисел быть не может). Итак, задача состоит в том, чтобы реализовать следующий псевдокод:

Листинг 13.1. Псевдокод программы-генератора случайной выборки

```
initialize set S to empty
/* инициализация набора S пустым множеством */
size = 0
while size < m do
```

```
  t = bigrand() % maxval
  if t is not in S /* элемента t еще нет в S */
    insert t into S /* добавить элемент t в набор S */
    size++
вывод элементов S в порядке возрастания
```

Назовем нашу структуру данных IntSet (набор целых чисел). Объявим ее как класс C++ со следующими экспортируемыми членами:

Листинг 13.2. Определение структуры для хранения набора целых чисел

```
class IntSetImp {
public
  IntSetImp(int maxelements, int maxval);
  void insert(int t);
  int size();
void report(int *v);
};
```

Конструктор IntSetImp инициализирует структуру пустым множеством. У функции имеется два аргумента, задающих максимальное количество элементов в наборе и максимальное значение элемента. В зависимости от реализации эти параметры могут игнорироваться. Функция insert добавляет новое целое число в набор, если оно там отсутствовало. Функция size возвращает текущее количество элементов в наборе, а функция report выводит эти элементы в порядке возрастания в массив v.

Этот короткий интерфейс подходит исключительно для обучения программированию. Ему недостает множества компонентов, необходимых для реального использования этого интерфейса, например обработки ошибок и деструктора. Опытный программист на C++ воспользовался бы абстрактным классом с виртуальными функциями, а потом бы наследовал его в своих реализациях. Мы пойдем более простым (а иногда и более эффективным путем) и будем использовать имена типа IntSetArr для реализации массива, IntSetList для реализации списка и так далее. Для произвольной реализации будет использоваться имя IntSetImp.

Приведенный в листинге 13.3 код использует описанную в листинге 13.2 структуру для формирования упорядоченного набора случайных целых чисел.

Листинг 13.3. Использование класса IntSetImp

```
void gensets(int m, int maxval)
{  int *v = new int[m];
  IntSetImp S(m, maxval);
  while (S size() < m)
    S insert(bigrand() % maxval)
  S report(v);
  for (int i = 0; i < m; i++)
    cout << v[i] << "\n";
}
```

Поскольку функция insert не позволяет добавить в набор два одинаковых элемента, нам нет необходимости проверять наличие элемента в наборе до вызова этой функции.

Проще всего реализовать класс IntSet с помощью мощного и достаточно общего шаблона set из стандартной библиотеки шаблонов C++ STL.

Листинг 13.4. Реализация класса IntSetSTL

```
class IntSetSTL {
private:
  set<int> S;
public:
  IntSetSTL(int maxelements, int maxval) { }
  int size() {return S.size(); }
  void insert(int t) { S.insert(t); }
  void report(int *v)
  {  int j = 0;
    set<int>::iterator i;
    for (i = S.begin(); i != S.end(); ++i)
      v[j++] = *i;
  }
};
```

Аргументы конструктора попросту игнорируются. Функции IntSet, size и insert полностью соответствуют своим аналогам в STL. В функции report для последовательного упорядоченного вывода элементов набора используется стандартный итератор. Эта структура общего вида достаточно хороша, но не идеальна. Мы вскоре доберемся до реализаций, которые будут в данной задаче в пять раз более эффективны по производительности и используемой памяти.

13.2. Линейные структуры

Попробуем реализовать набор самостоятельно. Начнем мы с простейшей структуры — массива. Количество элементов набора будет храниться в переменной n нашего класса, а сами числа — в массиве x.

```
private:
  int n, *x;
```

Полный текст реализации всех классов приводится в приложении 5. Приведенная в листинге версия конструктора на псевдокоде выделяет память под массив (с одним добавочным элементом-маркером) и устанавливает количество элементов равным нулю.

Листинг 13.5. Конструктор класса IntSetArray

```
IntSetArray(maxelements, maxval)
  x = new int[1 + maxelements]
  n = 0
  x[0] = maxval
```

Поскольку нам необходимо возвращать элементы в порядке возрастания, мы будем все время поддерживать этот порядок их хранения (в некоторых других приложениях выгодней использовать неотсортированные массивы). В конце множества упорядоченных элементов мы будем хранить элемент-маркер со значением maxval, превышающим любое возможное значение элементов набора. Теперь проверку на достижение конца списка можно будет заменить на проверку наличия большего элемента (которая нам и так нужна). Это упростит и ускорит код функции вставки элемента, приведенный в листинге 13.6.

Листинг 13.6. Вставка элемента в набор, реализованный через массив

```
void insert(t)
  for (i = 0, x[i] < t, i++)
    ;
  if x[i] == t
    return
  for (j = n, j >= i; j--)
    x[j+1] = x[j]
  x[i] = t
  n++
```

В первом цикле просматриваются элементы массива меньше значения вставляемого элемента t. Если такой элемент уже есть, можно немедленно выходить из функции. В противном случае мы сдвигаем элементы, большие данного, вправо на одну позицию (эта процедура применяется и к маркеру), вставляем элемент t в освободившуюся позицию и увеличиваем n на единицу. Эта функция выполняется за время $O(n)$.

Функция size во всех наших реализациях будет выглядеть одинаково.

Листинг 13.7. Функция size

```
int size()
  return n.
```

Функция report копирует все элементы, за исключением маркера, в выходной массив за время $O(n)$.

Листинг 13.8. Функция report в реализации IntSetArray

```
void report(v)
  for i = [0, n)
    v[i] = x[i]
```

Массивы отлично подходят для реализации наборов, когда количество элементов известно заранее. Поскольку массив изначально упорядочен, для определения принадлежности элемента набору можно использовать двоичный поиск, работающий за время $O(\log n)$. Дополнительные сведения о времени работы этой программы вы найдете в конце раздела.

Если размер набора заранее неизвестен, для его представления лучше всего использовать связные списки (рис. 13.1). При этом не требуется тратить время на сдвиг элементов при добавлении нового.

Рис. 13.1. Представление множества в виде связного списка

Внутренние члены класса IntSetList будут следующими.

Листинг 13.9. Внутренние члены класса IntSetList

```
private:
  int n.
  struct node {
    int val.
    node *next.
```

```
  node(int v, node *p) { val = v, next = p, }
};
node *head, *sentinel,
```

Каждый узел связного списка содержит целочисленное поле и указатель на следующий узел списка. Конструктор node присваивает этим полям значения своих аргументов.

Мы будем заботиться о том, чтобы список постоянно был упорядоченным, по той же причине, по которой заботились об упорядоченности массива. Мы также добавим в наш список узел-маркер, значение которого будет превышать значение любого элемента списка. Конструктор нашего класса будет создавать такой узел и устанавливать на него указатель head.

Листинг 13.10. Конструктор класса IntSetList

```
IntSetList(maxelements, maxval)
  sentinel = head = new node(maxval, 0)
  n = 0
```

Функция report перебирает элементы списка, помещая значения его узлов в выходной массив.

Листинг 13.11. Функция report класса IntSetList

```
void report(int *v)
  j = 0
  for (p = head, p != sentinel, p = p -> next)
    v[j++] = p->val
```

Чтобы вставить элемент в упорядоченный связанный список, мы перебираем его элементы до тех пор, пока не находим уже имеющийся элемент с тем же значением (в этом случае мы немедленно возвращаемся из функции) либо находим элемент с большим значением, перед которым вставляем новый. К сожалению, разнообразие возможных ситуаций сильно усложняет код (см. решение задачи 4). Простейшее решение этой задачи, известное мне, представляет собой рекурсивную функцию, вызываемую следующим образом:

Листинг 13.12. Функция insert для класса IntSetList

```
void insert(t)
  head = rinsert(head, t)
```

Рекурсивная часть проста:

```
node *rinsert(p, t)
  if p->val < t
    p->next = rinsert(p->next, t)
  else if p->val > t
    p = new node(t, p)
    n++
  return p
```

Когда задача усложняется разнообразием возможных случаев, применение подобной функции часто дает возможность сильно упростить код.

Когда один из вариантов реализации наборов применяется для генерации m случайных целых чисел, каждая из m операций поиска выполняется в среднем за время, пропорциональное m. Таким образом, на заполнение всей структуры по-

требуется время $O(m^2)$. Я предположил, что версия, использующая список, будет работать несколько быстрее, чем версия с массивом, потому что в ней используется дополнительная память (под указатели), а это исключает необходимость сдвига верхних элементов. В табл. 13.1 приведены данные о времени работы разных реализаций программы при n = 1 000 000 и m, меняющемся от 10 000 до 40 000.

Таблица 13.1. Время работы версий программы генерации случайных чисел, с

Структура	m=10 000	m=20 000	m=40 000
Массив	0,6	2,6	11,1
Список (простая версия)	5,7	31,2	170,0
Список (без рекурсии)	1,8	12,6	73,8
Список (групповое выделение памяти)	1,2	5,7	25,4

Время работы программы, основанной на массивах, росло квадратично с ростом m, как я и предполагал, и постоянный множитель в этой зависимости имел достаточно разумное значение. Однако для первой реализации со списком время выполнения уже для m = 10 000 оказывалось на порядок больше, чем для массивов, и росло быстрее, чем $O(m^2)$. Что-то в моих рассуждениях было неверным.

Вначале мне показалось, что проблема в рекурсии. Помимо дополнительных затрат на рекурсивный вызов функции, глубина рекурсии функции rinsert определяется положением искомого элемента, то есть может быть записана как $O(n)$. После рекурсивного перебора элементов к концу списка значение передается обратно последовательным присваиванием его указателям. Когда я заменил рекурсивную функцию на итеративную, описанную в решении задачи 4, время выполнения сократилось примерно в три раза.

Затем я решил изменить метод выделения памяти на метод, описанный в задаче 5. Вместо того чтобы выделять память при каждой операции вставки, конструктор выделял ее целым блоком под m узлов, а функция insert использовала эту память по мере необходимости. Улучшение было вызвано двумя причинами.

1. Модель затрат времени выполнения из приложения 3 показывает, что выделение памяти потребляет на два порядка больше времени, чем большинство простых операций. Мы заменили m таких дорогостоящих операций одной.
2. Модель затрат памяти из того же приложения говорит нам, что, если узлы выделяются блоком, на каждый из них уходит по 8 байт (4 байта на целое и 4 на указатель); 40 000 узлов занимают 320 Кбайт памяти и отлично помещаются в кэше второго уровня моего компьютера. Если узлы выделять по отдельности, на каждый из них будет тратиться 48 байт памяти и полный объем в 1,92 Мбайт превысит объем кэша второго уровня.

В другой системе с более эффективной подпрограммой выделения памяти удаление рекурсии ускорило работу в пять раз, тогда как переход к выделению памяти блоком дал всего лишь 10% выигрыш по скорости. Кэширование и удаление рекурсии, как и большинство других методов оптимизации, иногда дают значительные результаты, а иногда практически никаких.

Алгоритм вставки элемента в массив ищет подходящее место для вставляемого элемента, а затем сдвигает все последующие, освобождая для него место. Алго-

ритм вставки элемента в список делает только первую часть работы, но не вторую: сдвигать элементы больше нет необходимости. Почему же на вдвое меньшую работу программа, использующая списки, затрачивает вдвое больше времени? Во-первых, эта программа использует вдвое больше памяти: для больших списков приходится считывать 8 байт и помещать их в кэш, тогда как реально используются только 4 байта. Во-вторых, в массивах обращение к элементам хорошо предсказуемо, тогда как при работе со списками ячейки памяти считываются в случайном порядке.

13.3. Двоичное дерево поиска

Теперь мы перейдем от линейных структур к структурам, обеспечивающим быстрый поиск и вставку элементов. На рис. 13.2 показано двоичное дерево поиска, содержащее элементы 31, 41, 59 и 26 (последовательно добавлявшиеся именно в этом порядке).

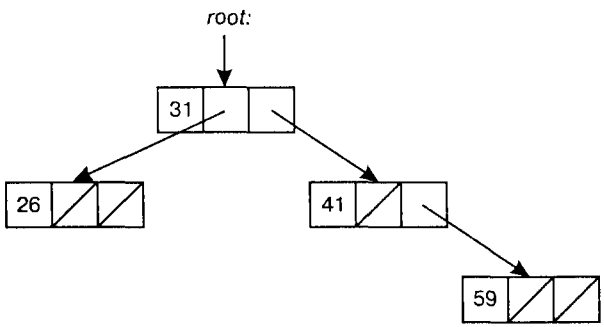


Рис. 13.2. Двоичное дерево

В классе IntSetBST мы определяем узлы дерева и его корень.

Листинг 13.13. Внутренняя структура класса IntSetBST

```
private
  int n, *v, vn;
  struct node {
    int val;
    node *left, *right;
    node(int i) { val = i; left = right = 0; }
  };
  node *root;
```

Дерево инициализируется пустым корнем. Действия с деревом осуществляются с помощью рекурсивных функций.

```
IntSetBST(int maxelements, int maxval) { root = 0; n = 0; }
void insert(int t) { root = rinsert(root, t); }
void report(int *x) { v = x; vn = 0; traverse(root); }
```

Функция вставки спускается вниз по дереву до тех пор, пока не будет найдено значение, равное данному (поиск в этом случае завершается), либо дерево закончится (тогда к нему добавляется новый узел).

Листинг 13.14. Рекурсивная функция добавления нового элемента к дереву

```
node *rinsert(p, t)
  if p == 0
    p = new node(t)
    n++
  else if t < p->val
    p->left = rinsert(p->left, t)
  else if t > p->val
    p->right = rinsert(p->right, t)
// Если p->val == t ничего делать не нужно
  return p
```

Поскольку в нашем приложении элементы вставляются в случайном порядке, нам не нужно беспокоиться о балансировке дерева, которая достаточно сложна. В задаче 1 показывается, что в других алгоритмах со случайными наборами дерево может получиться сильно разбалансированным.

При *симметричном обходе*[1] вершин дерева сначала обрабатывается левое поддерево, затем возвращается значение в узле, а потом обрабатывается правое поддерево.

Листинг 13.15. Функция симметричного обхода вершин дерева

```
void traverse(p)
  if p == 0
    return
  traverse(p->left)
  v[vn++] = p->val
  traverse(p->right)
```

Здесь для индексации следующего свободного элемента массива v используется переменная vn.

Таблица 13.2 показывает время работы для структуры set из библиотеки STL, которая была описана в разделе 13.1 (данные взяты с моей системы), для двоичных деревьев и для некоторых других структур, с которыми мы встретимся в следующем разделе. Максимальное значение зафиксировано: $n = 10^8$. Значение m увеличивается до тех пор, пока в системе не заканчивается оперативная память и не начинается активная работа с файлом подкачки.

Таблица 13.2. Время работы программы генерации случайной выборки для некоторых структур данных

Структура	m = 1 000 000		m = 5 000 000		m = 10 000 000	
	сек	Мбайт	сек	Мбайт	сек	Мбайт
STL	9,38	72				
BST	7,30	56				
BST*	3,71	16	25,26	79		
Корзины	2,36	60				

[1] В мою первую версию этой программы вкралась хитрая ошибка. Компилятор сообщал о наличии внутреннего несоответствия и завершал работу. Я отключил оптимизацию, ошибки исчезли, и я мысленно обругал авторов компилятора. Позже я понял, что при написании кода функции обхода я забыл включить проверку на нулевое значение p. Оптимизатор пытался преобразовать концевую рекурсию в цикл и завершал работу, поскольку не мог найти условия для выхода из цикла.

Структура	m = 1 000 000		m = 5 000 000		m = 10 000 000	
	сек	Мбайт	сек	Мбайт	сек	Мбайт
Корзины*	1,02	16	5,55	80		
Битовый вектор	3,72	16	5,70	32	8,36	52

Приведенные в таблице данные не включают затраты на вывод результатов (требующий приблизительно столько же времени, сколько работает библиотека STL). В простой реализации двоичных деревьев поиска исключена сложная схема балансировки, используемая STL (спецификация STL гарантирует хорошую производительность даже в худшем случае), и поэтому она работает несколько быстрее и использует меньший объем памяти. Для реализации STL обращения к диску начинались при m = 1 600 000, тогда как первый вариант BST доходил до m = 1 900 000. Сокращение BST* обозначает двоичное дерево поиска с некоторыми оптимизациями. Главная из них состоит в одновременном выделении памяти под все узлы единым блоком (как в задаче 5). Это значительно уменьшает затраты памяти и сокращает время выполнения примерно на одну треть. Помимо этого, рекурсия в этой программе заменяется итерацией (как в задаче 4) и используется узел-маркер, описанный в задаче 7, что приводит к дополнительному увеличению скорости на 25%.

13.4. Структуры для целых чисел

Перейдем к последним двум структурам, которые построены с учетом того, что нам необходимо хранить именно целые числа, а не данные каких-либо других типов. Битовые векторы знакомы нам с главы 1. В листинге 13.16 приведены внутренние данные и функции класса IntSetBitVec.

Листинг 13.16. Внутренние данные и функции класса IntSetBitVec

```
enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F },
int n, hi, *x,
void set(int i) { x[i>>SHIFT] |= (1<<(i & MASK)), }
void clr(int i) { x[i>>SHIFT] &= ~(1<<(i & MASK)), }
int test(int i) { return x[i>>SHIFT] & (1<<(i & MASK)), }
```

Конструктор в листинге 13.17 выделяет память под массив и сбрасывает все его биты в 0.

Листинг 13.17. Конструктор класса IntSetBitVec

```
IntSetBitVec(maxelements, maxval)
  hi = maxval
  x = new int[1 + hi/BITSPERWORD]
  for i = [0, hi)
    clr(i)
  n = 0
```

В задаче 8 вам предлагается ускорить это процесс путем перехода к пословным операциям. Аналогично можно ускорить и функцию report, приведенную в листинге 13.18.

Листинг 13.18. Функция report класса IntSetBitVec

```
void report(v)
  j = 0
  for i = [0, hi)
    if test(i)
      v[j++] = i
```

Наконец, функция insert (листинг 13.19) включает соответствующий бит и увеличивает число n, но только в том случае, если на момент ее вызова бит был сброшен.

Листинг 13.19. Функция insert класса IntSetBitVec

```
void insert(t)
  if test(t)
    return
  set(t)
  n++
```

Таблица 8.2 показывает, что, если n достаточно мало, чтобы битовый вектор целиком поместился в оперативной памяти, эта структура оказывается весьма эффективной (а задача 8 предлагает вам еще больше увеличить ее производительность). К сожалению, если $n = 2^{32}$, то битовый вектор потребует полгигабайта ОЗУ.

Последняя структура данных объединяет достоинства списков и битовых векторов. Целые числа раскидываются по набору *«корзин»* (bins). Предположим, что нужно выбрать 4 числа из диапазона 0..99. Поместим их в четыре корзины. Корзина 0 предназначена для чисел 0..24, корзина 1 — для чисел 25..49, корзина 2 — для чисел 50..74, а корзина 3 — для чисел 75..99.

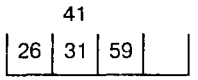


Рис. 13.3. Корзины для элементов

Корзины можно рассматривать как разновидность хэширования. Числа в каждой из корзин будут храниться в виде упорядоченного связанного списка. Поскольку распределение чисел однородно, в каждом из списков окажется в среднем по одному элементу.

В листинге 13.20 приведены внутренние данные класса IntSetBins.

Листинг 13.20. Внутренние данные класса IntSetBins

```
private
  int n, bins, maxval;
  struct node {
    int val;
    node *next;
    node(int v, node *p) { val = v, next = p; }
  };
  node **bin, *sentinel;
```

Конструктор в листинге 13.21 выделяет массив корзин и элемент-маркер с большим значением, а затем инициализирует все корзины указателями на этот маркер.

Листинг 13.21. Конструктор класса IntSetBins

```
IntSetBins(maxelements, pmaxval)
  bins = maxelements
  maxval = pmaxval
  bin = new node*[bins]
  sentinel = new node(maxval, 0)
  for i = [0, bins)
    bin[i] = sentinel
  n = 0
```

Функция insert должна помещать целое число t в соответствующую корзину. Очевидное отображение с помощью функции t * bins / maxval может привести к арифметическому переполнению (и сложностям при отладке, как я могу засвидетельствовать по собственному печальному опыту). Поэтому в нашей программе мы воспользуемся более безопасным отображением.

Листинг 13.22. Функция insert класса IntSetBins

```
void insert(t)
  i = t/ (1 + maxval/bins)
  bin[i] = rinsert(bin[i], t)
```

В данном случае функция rinsert такая же, как и для связанного списка. Аналогичным образом и функция report берется из программы со списками и применяется к каждой из корзин по очереди.

Листинг 13.23. Функция report класса IntSetBins

```
void report(v)
  j = 0
  for i = [0, bins)
    for (node *p = bin[i], p != sentinel; p = p->next)
      v[j++] = p->val
```

Таблица 8.2 свидетельствует, что реализация, использующая корзины, оказывается достаточно быстрой. Строка «Корзины*» содержит времена выполнения оптимизированной версии программы с корзинами, в которой память под все узлы выделяется одним блоком в процессе инициализации (как в задаче 5). Новая структура использует в четыре раза меньше памяти и работает вдвое быстрее исходной. Раскрытие рекурсии позволило ускорить программу еще на 10%.

13.5. Принципы

Мы бегло прошлись по пяти структурам данных, используемым для представления наборов. Средняя производительность этих структур, когда m мало по сравнению с n, описывается табл. 13.3, где b обозначает количество битов в слове.

Таблица 13.3. Средняя производительность структур для представления наборов

Представление набора	Инициализация	Вставка элемента	Вывод элементов	Полное время работы	Объем памяти в словах
Упорядоченный массив	1	m	m	$O(m^2)$	m
Упорядоченный список	1	m	m	$O(m^2)$	$2m$

продолжение

Таблица 13.3 (*продолжение*)

Представление набора	Инициализация	Вставка элемента	Вывод элементов	Полное время работы	Объем памяти в словах
Двоичное дерево	1	$\log m$	m	$O(m \log m)$	$3m$
Корзины	m	1	m	$O(m)$	$3m$
Битовый вектор	n	1	n	$O(m)$	n/b

Эта таблица отражает лишь поверхностные сведения о представлении наборов в подобных задачах. В решении задачи 10 предлагаются другие возможности. В разделе 15.1 главы 15 описаны структуры данных для поиска в наборах слов.

Несмотря на то что мы занимались только структурами данных, пригодными для представления наборов, мы узнали несколько принципов, применимых ко многим задачам программирования.

Важность библиотек

Стандартная библиотека шаблонов C++ STL предоставляет общее решение, простое в реализации и обладающее хорошими возможностями расширения и улучшения. Когда вы сталкиваетесь с задачей, в которой нужно использовать структуры данных, прежде всего следует искать общее средство решения такой задачи. В конкретном случае специализированные алгоритмы, учитывающие особенности задачи, дают возможность значительно увеличить производительность программы.

Важность памяти

В разделе 13.2 мы столкнулись с тем, что для оптимизированного списка вдвое меньшая по сравнению с массивами работа выполняется за вдвое большее время. Почему? Потому что в массиве на один элемент приходится вдвое меньше памяти, а обращение к элементам осуществляется последовательно. В разделе 13.3 мы обнаружили, что использование специальной процедуры выделения памяти сокращает занимаемый двоичным деревом объем втрое и ускоряет скорость работы вдвое. Время работы увеличивалось скачками, когда занимаемая память превышала некоторые границы: полмегабайта (объем кэша второго уровня) и 80 Мбайт (объем свободной памяти в ОЗУ) — все эти данные приведены для моего компьютера.

Методы оптимизации программ

Наиболее существенное увеличение быстродействия во всех случаях достигалось переходом к блочному выделению памяти. Это решение позволило избавиться от множества дорогостоящих вызовов. Замена рекурсии на итерацию ускорила работу связанных списков втрое, но для корзин эта же операция дала всего лишь 10% выигрыш. Использование маркеров в большинстве структур позволяет сделать код ясным и простым, и одновременно уменьшает время выполнения программы.

13.6. Задачи

1. В решении задачи 9 из предыдущей главы описан алгоритм получения упорядоченного набора целых чисел, придуманный Бобом Флойдом. Можете

ли вы реализовать его алгоритм с помощью структур IntSet, описанных в этой книге? Насколько удачно эти структуры работают с неслучайными распределениями, порождаемыми алгоритмом Флойда?

2. Как бы вы изменили упрощенный интерфейс класса IntSet, чтобы сделать его более надежным?
3. Добавьте к классам функцию find, позволяющую определить, присутствует ли некоторый элемент в списке. Можете ли вы реализовать эту функцию так, чтобы она работала быстрее, чем insert?
4. Перепишите функции вставки для списков, корзин и двоичных деревьев, заменив рекурсию итерацией. Затем измерьте разницу во времени выполнения.
5. В разделе 9.1 главы 9 и решении задачи 9 из той же главы описан способ Криса Ван Вайка, использованный им для уменьшения количества обращений к подсистеме выделения памяти. Он создавал собственную структуру, в которой хранил указатели на доступные узлы. Покажите, как можно применить эту идею к структурам IntSet, реализованным через списки, корзины и двоичные деревья.
6. Что даст вам измерение времени работы приведенного ниже фрагмента кода для различных реализаций класса IntSet?

```
IntSetImp S(m, n);
for (int i = 0; i < m; i++)
 S.insert(i);
```

7. В наших реализациях с массивами, списками и корзинами используются элементы-маркеры. Покажите, как этот метод можно применить к двоичным деревьям поиска.
8. Покажите, как можно ускорить операции инициализации и перечисления элементов для битовых векторов, используя параллельность одновременных операций с несколькими битами. С каким из типов данных программа работает быстрее всего: char, short, int, long или с каким-нибудь другим?
9. Покажите, как можно ускорить работу корзин, заменив дорогостоящий оператор деления на более дешевый логический сдвиг.
10. Какие еще структуры можно использовать для представления наборов целых чисел в задаче о генерации случайной выборки?
11. Постройте самую быструю функцию для получения упорядоченного набора случайных целых чисел без повторений. Чувствуйте себя свободным от любых ограничений на вид интерфейса представления наборов.

13.7. Дополнительная литература

Отличные учебники Кнута и Седжвика по теории алгоритмов описаны в разделе 11.6. Поиск является предметом исследования главы 6 (второй половины) книги Кнута «Сортировка и поиск» и четвертой (последней) части книги Седжвика «Алгоритмы».

13.8. Примеры поиска

Упрощенные структуры основной части этой главы снабдили нас знаниями, достаточными для перехода к изучению структур данных, применяющихся в промышленных приложениях. Это дополнение посвящено примечательной структуре, использованной Дугом Макилроем для хранения словаря в программе ispell, написанной им в 1978 году. Когда я писал первые варианты глав этой книги в 1980 году, для проверки правописания я пользовался именно программой Макилроя. Для второго издания я снова воспользовался программой spell и пришел к выводу, что она все еще очень полезна. Подробности о программе Макилроя можно узнать из его статьи «Разработка орфографического словаря» (IEEE Transactions and Communications COM-30, 1, Jan 1982, pp. 91–99). В моем толковом словаре «жемчужина» определяется как *«нечто избранное или драгоценное»* — эта программа вполне подходит под такое определение.

Первая задача, с которой столкнулся Макилрой, состояла в составлении списка слов. Он начал с пересечения полного словаря (для обеспечения правильности) со списком Брауновского университета объемом в миллион слов. Это было достойное начало, но работы было еще много.

Подход Макилроя лучше всего проявился в том, как он искал имена собственные, которые не включены в большинство словарей. Сначала он занялся человеческими именами: 1000 самых часто встречающихся фамилий из большого телефонного справочника, список мужских и женских имен, известные фамилии (например, Дейкстра и Никсон) и мифологические имена из индекса к одной из книг. Поразмыслив над «ошибками» вроде Xerox и Texaco, он решил добавить в словарь названия компаний из списка 500 наиболее известных. Издательские компании часто встречаются в библиографиях, поэтому он добавил и их. Затем Макилрой занялся географией: страны, их столицы, штаты, столицы штатов, названия ста самых крупных городов США и мира, океаны, планеты и звезды.

Он добавил к словарю названия наиболее распространенных животных и растений, термины из химии, анатомии и информатики. Но ему приходилось быть внимательным, чтобы не добавлять слишком много слов: в словарь не были включены специальные термины, которые могли бы считаться написанными с ошибкой обычными словами (как геологический термин cwm), а когда правильных вариантов написания было несколько, он включал только один из них (traveling, но не travelling[1]).

Одним из ключей к решению задачи был анализ результатов работы spell с реальными текстами. В течение некоторого времени программа автоматически отсылала ему копии результатов (на компромиссы между конфиденциальностью и эффективностью в те дни смотрели по-другому). Обнаружив ошибку, Макилрой решал ее наиболее общим способом. В результате получился отличный список из 75 000 слов. В него входит большинство слов, используемых мной в письменной речи, и при этом программа обнаруживает мои ошибки.

В программе использовался анализ аффиксов, позволяющий отделять приставки и суффиксы и оставлять одни корни. Это одновременно и необходимо и удобно. Необходимость связана с тем, что полного списка слов для английского языка быть не может; любая программа проверки орфографии должна делать предполо-

[1] Travelling — соответствует правилам орфографии, принятым в Великобритании, а traveling — в США. — *Примеч. ред.*

жения о происхождении слов наподобие misrepresented, либо она будет считать неправильными большое количество нормальных слов. Анализ аффиксов дал положительный побочный эффект, заключавшийся в уменьшении словаря.

Задачей анализа аффиксов является редукция слов наподобие misrepresented до sent, отделяя mis-, re-, pre- и -ed. Хотя слово represent и не означает «показывать заново», а present не значит «отправленный заранее», программа spell все равно использует подобные совпадения для уменьшения словаря. В таблицах программы содержится 40 правил добавления приставок и 30 правил добавления суффиксов. Список из 1300 исключений позволяет отбросить красивые, но неправильные догадки наподобие редукции слова entend (ошибочное написание intend) до en + tend. Итак, этот анализ позволяет уменьшить словарь из 75 000 слов до 30 000. Программа Макилроя обрабатывает каждое слово, отбрасывая приставки и суффиксы и проверяя наличие оставшейся части в словаре, до тех пор пока не обнаружится совпадение или не закончатся аффиксы (в последнем случае слово считается написанным с ошибкой).

Предварительные оценки показали, что весь словарь нужно постараться вместить в оперативную память. Это было особенно сложной задачей для Макилроя, который писал свою программу для PDP-11 с 64 Кбайт адресного пространства. О методах экономии памяти в аннотации к его статье говорится так: «Отбрасывание суффиксов и приставок уменьшает список на треть, хэширование позволяет избавиться от 60% оставшихся битов, а сжатие уменьшает объем еще вдвое». Таким образом ему удалось вместить список из 75 000 английских слов (и приблизительно такого же количества словоформ) в 26 000 16-битных машинных слов.

Макилрой использовал хэширование, чтобы представить каждое из 30 000 слов в 27 битах (чуть позже мы узнаем, почему именно в 27). Рассмотрим работу схемы на списке из пяти слов:

```
a list of five words
```

Первый метод хэширования состоит в использовании n-элементной хэш-таблицы, размер которой совпадает с количеством слов в списке, и функции хэширования, отображающей строку в целое число из диапазона [0, n). Пример функции хэширования для строк будет рассмотрен в разделе 15.1 главы 15. Поле таблицы с номером i указывает на связанный список, содержащий все строки, для которых функция хэширования возвращает значение i. Если пустые списки представить пустыми клетками и предположить, что h (a) = 2, h (list) = 1 и так далее, то таблица для нашего образца списка может выглядеть, например, следующим образом (рис. 13.4).

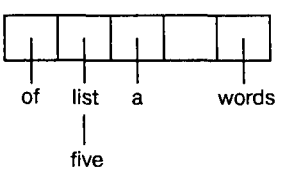


Рис. 13.4. Простейшая хэш-таблица

Чтобы найти слово w, нам нужно осуществить последовательный поиск в списке, на который указывает поле таблицы с номером h (w).

В следующей схеме используется таблица гораздо большего размера. При n = 23 в большинстве клеток, скорее всего, окажется не более одного элемента (рис. 13.5). В этом примере h (a) = 13 и h (list) = 5.

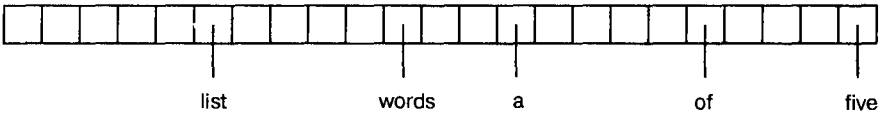


Рис. 13.5. Хэш-таблица для случая h (a) = 13 и h (list) = 5

В программе ispell n = 2^{27} (около 134 миллионов), и почти все непустые списки состоят из одного элемента.

Следующим ходом Макилрой рискнул: вместо связанного списка в каждое поле таблицы он поместил единственный бит. При этом объем используемой памяти значительно сокращается, но возникают ошибки. На рис. 13.6 используется та же функция хэширования, что и в предыдущем примере, а нулевым битам соответствуют пустые ячейки:

Рис. 13.6. Представление хэш-таблицы в виде битового массива

Чтобы найти слово w, программа обращается к биту h(w). Если этот бит сброшен, программа сообщает, что слова w в таблице нет. Если бит равен 1, программа делает предположение, что слово w в таблице есть. Иногда функция хэширования дает для ошибочного слова то же значение, что и для правильного, но вероятность такой ошибки 30 000/2^{27}, то есть около 1/4000. В среднем каждое из 4000 неправильных слов будет сочтено правильным. Макилрой предположил, что в типичном черновике содержится не более 20 ошибок, поэтому дефект будет проявляться не чаще, чем в одном из сотни документов, — поэтому он и выбрал число 27.

Представление таблицы строкой из n = 2^{27} битов потребовало бы более 16 миллионов байтов. Поэтому в программе хранятся только единичные биты. В приведенном ранее примере были бы сохранены значения 5, 10, 13, 18, 22. Слово w считается присутствующим в таблице, если в этой последовательности имеется значение h(w). Очевидное представление подобной последовательности потребовало бы 30 000 27-битных слов, но в компьютере Макилроя было всего лишь 32 000 16-битных, поэтому он отсортировал список и использовал код переменной длины для записи разностей между соседними значениями. Предполагая наличие фиктивного нулевого начального значения, мы получили бы новую последовательность: 5, 5, 3, 5, 4. Разности в программе Макилроя занимали в среднем 13,6 бит каждая. При этом оставалось несколько сотен лишних слов памяти, которые были отведены на указатели на некоторые точки в сжатом списке, что позволяло ускорить процесс поиска. Таким образом, получился словарь объемом 64 Кб, малым временем доступа и малой вероятностью ошибки.

Мы уже упомянули о двух достоинствах программы spell: она дает полезные результаты и словарь помещается в адресном пространстве. Помимо этого, программа была еще и быстрой. Даже на древних компьютерах, для которых она была впервые написана, проверка статьи в десять страниц требовала полминуты, а книгу вроде этой программа проверила бы за десять минут (что в те годы казалось безумно быстрым). Правописание одного слова можно было проверить за несколько секунд, поскольку маленький словарь можно было быстро считать с диска.

Кучи

Эта глава посвящена *«кучам»* — структурам данных, используемых для решения двух важных задач.

1. **Сортировка.** Сортировка массива размера n с помощью кучи выполняется за время, ни при каких условиях не превышающее $O(n \log n)$. Кроме того, этот метод сортировки использует лишь несколько байт дополнительной памяти.
2. **Очереди с приоритетом.** Для куч поддерживаются операции добавления нового и удаления наименьшего элемента. Каждая из этих операций выполняется за время $O(\log n)$.

В обеих задачах *кучи* облегчают программирование и сокращают объем вычислений.

Эта глава организована по принципу «снизу вверх»: мы начнем с подробностей и только потом перейдем к решению задач. В двух первых разделах главы описывается структура кучи и две функции, к ней применяемые. В двух следующих разделах мы покажем, как пользоваться кучами и упомянутыми функциями для решения описанных выше задач.

14.1. Структура данных

Куча представляет собой структуру данных, используемую для представления множеств элементов[1]. В наших примерах используются числа, но элементы кучи могут быть любого типа, для которого введено отношение порядка. На рис. 14.1 изображена куча из 12 элементов (элементами являются целые числа).

[1] В другом контексте слово «куча» означает большой сегмент памяти, из которого динамически выделяются объекты переменного размера. Однако в нашей главе мы не будем использовать его в этом контексте.

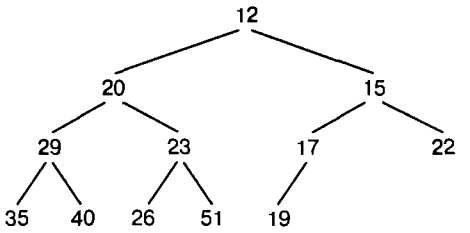


Рис. 14.1. Куча из 12 элементов

Это двоичное дерево является кучей благодаря двум своим свойствам.

Первое свойство мы назовем *«порядком»*: значение любого узла не превышает значения любого из его дочерних узлов. Отсюда следует, что наименьший элемент набора находится на вершине дерева (в нашем примере это число 12). Однако о соотношении левого и правого дочерних узлов ничего не говорится.

Второе свойство кучи называется *«формой»*. Идея «формы» лучше всего передается рис. 14.2.

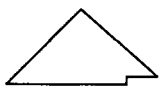


Рис. 14.2. Форма кучи

Другими словами, двоичное дерево, обладающее свойством «формы», заканчивается не более чем на двух уровнях (то есть заключительные узлы расположены не более чем на двух уровнях), причем находящиеся на нижнем уровне узлы сгруппированы слева. В дереве не должно быть незанятых узлов. Если в нем содержится n узлов, ни один из них не может отстоять более чем на $\log_2 n$ от корня. Вскоре мы увидим, как эти два свойства оказываются достаточно жесткими, чтобы мы могли найти наименьший элемент набора, но при этом допускают реорганизацию структуры при добавлении или удалении элемента.

Перейдем от абстрактных свойств куч к их реализации. Наиболее общее представление двоичных деревьев использует записи и указатели. Мы воспользуемся представлением, пригодным только для двоичных деревьев, обладающих свойством формы. Дерево, обладающее свойством формы и состоящее из 12 элементов, может быть представлено в 12-элементном массиве так, как это показано на рис. 14.3.

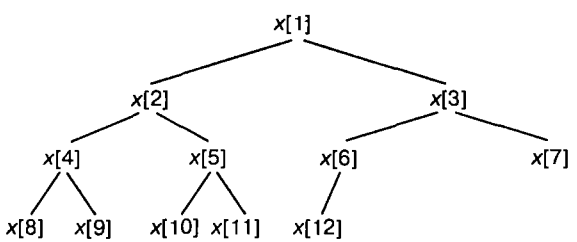


Рис. 14.3. Представление кучи с помощью массива

Обратите внимание, что для представления куч массивы нумеруются с единицы. Избежать проблем в языке C проще всего, объявив массив x[n+1] и забыв о нулевом элементе. В этом неявном представлении двоичного дерева его корень хранится в элементе x[1], узлы первого уровня — в элементах x[2] и x[3], и так далее. Типичные функции для такого дерева определяются следующим образом:

```
root = 1
value(i) = x[i]
leftchild(i) = 2*i
rightchild(i) = 2*i + 1
parent(i) = i/2
null(i) = (i < 1) или (i > n)
```

Неявное дерево из n элементов обязательно будет обладать свойством формы: отсутствие элементов в нем просто не предусмотрено.

На рис. 14.4 показана куча из 12 элементов и ее реализация в 12-элементном массиве.

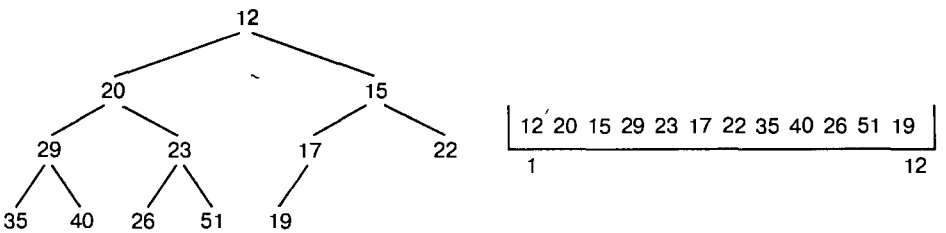


Рис. 14.4. Куча и ее реализация в массиве

Поскольку свойство формы гарантируется представлением, с этого места основное внимание будет уделено другому свойству, поэтому слово «куча» будет означать, что значение любого узла не меньше, чем значение его родителя. Иначе говоря, массив x[1..n] обладает свойством кучи, если

$$\forall_{2 \le i \le n} x[i/2] \le x[i].$$

Не забудьте, что целочисленный оператор деления (/) осуществляет округление с недостатком, поэтому 4/2 = 2 и 5/2 = 2. В следующем разделе нам придется говорить о подмножестве массива x[l..u], обладающем свойством кучи (свойством формы он обладает изначально). Поэтому математически мы можем определить кучу heap (l, u) следующим образом:

$$\forall_{2l \le i \le u} x[i/2] \le x[i].$$

14.2. Две важные функции

В этом разделе мы рассмотрим две функции, восстанавливающие массивы, свойство «кучности» которых было нарушено на одном из концов. Обе функции являются эффективными: для реорганизации кучи из n элементов обе они требуют примерно $\log n$ шагов. Учитывая построение главы, функции будут определены в этом разделе, а использованы в следующем.

Добавление произвольного элемента в поле x[n] при условии, что x[1..n-1] является кучей, не обязательно даст кучу heap (1, n). Восстановление свойства кучи (то есть распространение его на новый элемент) обеспечивается функцией siftup (просеивание вверх). Имя функции описывает стратегию, лежащую в ее основе: новый элемент просеивается и всплывает вверх до полагающегося ему места, обмениваясь позициями с вышестоящими элементами. В этом разделе используется типичное определение кучи, в котором новый элемент движется вверх: x[1] находится наверху кучи, поэтому x[n] находится внизу. Процесс иллюстрируется рис. 14.5, на котором (слева направо) показан процесс подъема нового элемента 13 вверх по дереву, пока он не оказывается на подобающем ему месте в качестве узла первого уровня.

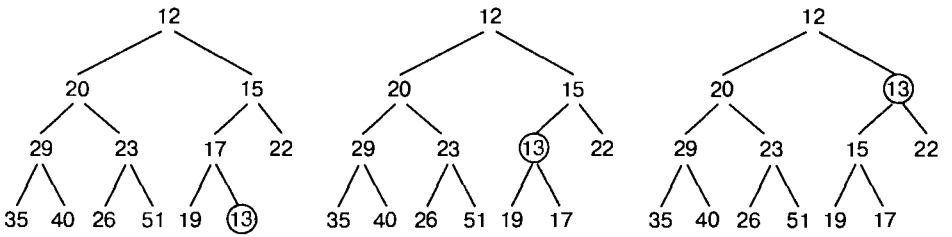


Рис. 14.5. Процесс подъема нового элемента в куче

Процесс продолжается до тех пор, пока узел с новым элементом не окажется больше либо равным значению родительского узла (как на рис. 14.5) либо пока он не поднимется на самую вершину кучи. Если процесс начинается при условии, что истинно высказывание heap(1, n-1), то к его завершению становится истинным высказывание heap(1, n).

Получив представление о работе функции, займемся ее кодированием. Процесс просеивания указывает на необходимость организовать цикл, поэтому мы сформулируем инвариант цикла. Из рис. 14.5 видно, что свойство кучи выполняется везде в дереве, за исключением узла с новым элементом и его родителя. Если i — индекс нового узла, то мы можем использовать следующий инвариант:

```
loop
   /* инвариант  heap(1, n) за исключением, возможно, узла i и его роди-
теля */
```

Поскольку изначально истинно утверждение heap(1, n-1), мы можем инициализировать цикл присваиванием i = n.

В цикл нужно вставить проверку необходимости продолжения работы. Работа считается завершенной, если обрабатываемый элемент достиг вершины кучи или если он больше либо равен значению родительского узла. Инвариант утверждает, что свойство кучи сохраняется для всех элементов, за исключением, быть может, данного элемента и его родителя. Если верно условие i == 1, тогда у узла i нет родителя и свойство кучи выполняется везде, поэтому цикл можно завершить. Если у i есть родитель, мы присваиваем переменной p его индекс: p = i/2. Если x[p]<=x[i], тогда свойство кучи выполняется везде и цикл можно завершить.

Если, напротив, элемент i и его родитель находятся в неправильном соотношении, тогда мы меняем их местами (элементы x[i] и x[p]). Этот шаг показан на рис. 14.6.

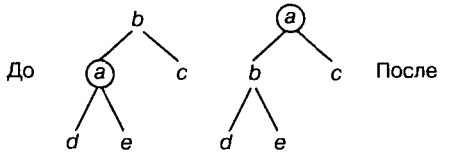


Рис. 14.6. Перестановка узлов, стоящих в неправильном порядке

После перестановки все пять элементов оказываются в правильном порядке: b < d и b < e, поскольку изначально элемент b был расположен выше[1]. Далее, a < b, поскольку условие x[p] <= x[i] не выполнилось, и a < c, поскольку a < b, а b < c. Таким образом, свойство кучи оказывается выполненным везде в массиве, за исключением, быть может, элемента p и его родителя, поэтому мы восстанавливаем инвариант, выполняя присваивание i = p.

Собрав все это вместе, получим функцию, представленную в листинге 14.1, которая выполняется за время, пропорциональное $\log n$, поскольку именно столько уровней имеется в куче.

Листинг 14.1. Функция просеивания нового элемента вверх

```
void siftup(n)
        предусловие: n > 0 && heap(1, n-1)
        постусловие: heap(1, n)
   loop
        /* инвариант  heap(1, n) за исключением, быть может, элемента i
и его родителя */
        if i == 1
             break
        p = i/2
        if x[p] <= x[i]
             break
        swap(p, i)
        i = p
```

Как и в главе 4, предусловия и постусловия характеризуют функцию. Если предусловие истинно до вызова функции, то постусловие окажется истинным после возврата из нее.

От просеивания вверх перейдем к просеиванию вниз. Если x[1..n] — куча, а мы присваиваем элементу x[1] новое значение, то в любом случае остается верным утверждение heap(2, n). Функция siftdown предназначена для расширения этого утверждения до heap(1, n). Это осуществляется путем просеивания элемента x[1] вниз по массиву, до тех пор, пока у него не закончатся дочерние элементы или пока элемент не окажется не превышающим значений всех его дочерних элементов. На рис. 14.7 элемент 18 просеивается вниз до тех пор, пока он не станет меньшим его единственного дочернего элемента (19).

Когда элемент просеивается вверх (всплывает), он всегда движется к корню дерева. Когда элемент просеивается вниз (тонет), процесс может быть более сложным: стоящий не на своем месте элемент обменивается с меньшим из своих дочерних элементов.

---

[1] Это важное свойство не учитывается инвариантом. Д. Кнут отмечает, что инвариант следует усилить до: «Высказывание heap(1, n) остается истинным, если у i нет родительского элемента. В противном случае оно осталось бы истинным, если x[i] заменить x[p], где p — родительский элемент для i».

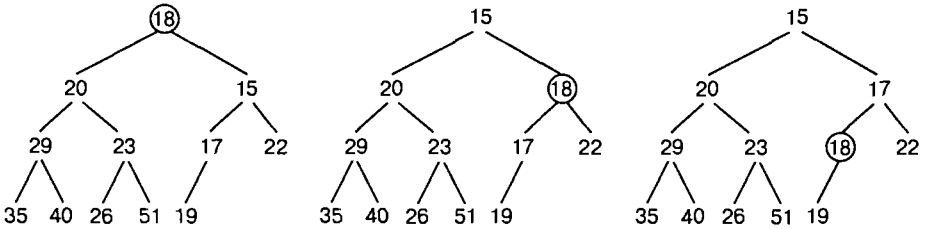


Рис. 14.7. Просеивание элемента вниз

Рисунок 14.7 иллюстрирует также инвариант цикла функции siftdown: свойство кучи остается верным везде, за исключением, быть может, обрабатываемого узла и его дочерних узлов.

```
loop
   /* инвариант: heap(1, n) истинно, за исключением, быть может, i и его
дочерних элементов (0, 1 или 2) */
```

Построение цикла аналогично функции siftup. Сначала нужно проверить, есть ли у i дочерние элементы. Если их нет, цикл завершается. Затем начинаются тонкости: если у i есть дочерние элементы, переменной c присваивается индекс наименьшего из них. Наконец, мы либо завершаем цикл, если x[i] <= x[c], либо движемся дальше к основанию кучи, обменивая местами x[i] и x[c] и выполняя присваивание i = c.

Листинг 14.2. Функция просеивания вниз (siftdown)

```
void siftdown(n)
        предусловие: heap(2, n) && n >= 0
        постусловие: heap(1, n)
   i = 1
   loop
        /* инвариант: heap(1, n) верно, за исключением, быть может, i и
его дочерних элементов (0, 1 или 2) */
        c = 2*i
        if c > n
             break
        /* c - левый дочерний элемент i */
        if c+1 <= n
             /* c+1 - правый дочерний элемент i */
             if x[c+1] < x[c]
                  c++
        /* c - наименьший из дочерних элементов */
        if x[i] <= x[c]
             break
        swap(c, i)
        i = c
```

Анализ возможных случаев, аналогичный проведенному для функции siftup, показывает, что операция обмена оставляет свойство кучи верным для всех элементов, кроме, быть может, c и его дочерних элементов. Как и siftup, эта функция работает за время, пропорциональное $\log n$, поскольку на каждом из уровней кучи выполняется постоянное количество операций.

14.3. Очереди с приоритетом

У любой структуры данных есть две стороны. Если смотреть снаружи, спецификация определяет ее функционирование — очередь сохраняет порядок элементов при

операциях вставки и извлечения (insert, extract). Реализация определяет то, как это делается — в основе может быть массив или связанный список. Мы начнем изучение очередей с приоритетом с описания их абстрактных свойств, а затем перейдем к реализациям.

Очередь с приоритетом изначально представляет собой пустой набор[1] элементов, который мы обозначим буквой S. Функция insert добавляет в этот набор новый элемент. Ее можно определить с помощью предусловий и постусловий.

```
void insert(t)
        предусловие: |S| < максимального размера
        постусловие. текущее состояние S = предыдущее состояние S.
объединенное с элементом t
```

Функция извлечения extractmin удаляет наименьший элемент из набора и возвращает его значение в своем единственном параметре t.

```
int extractmin()
        предусловие  |S| > 0
        постусловие  исходное состояние S = текущее состояние S. объеди-
ненное с результатом && результат = min(исходное состояние S)
```

Эта функция, конечно же, может быть изменена так, чтобы возвращать и максимальный элемент или какой-либо другой крайний для данной операции порядка.

Мы можем описать класс C++, решающий эту задачу, с помощью шаблона, в котором тип T элемента очереди может быть произвольным.

Листинг 14.3. Шаблон класса очереди с приоритетом

```
template <class T>
class priqueue {
public
   priqueue (int maxsize).  // инициализация пустым множеством
   void insert(T t).        // добавление t в очередь S
   T extractmin ().         // возвращение наименьшего элемента из S
}.
```

Очереди с приоритетом оказываются полезными во многих приложениях. В операционной системе такая структура может использоваться для обработки множества задач. Они помещаются в очередь в произвольном порядке, а следующая задача извлекается из набора:

```
priqueue <Task> queue.
```

В дискретной модели элементами являются моменты событий. В цикле имитации очередное событие извлекается из очереди (а новые могут туда добавляться):

```
priqueue<Event> eventqueue.
```

В обоих приложениях базовый класс очереди с приоритетом должен быть расширен дополнительной информацией. В нашем обсуждении мы опустим детали реализации, но в создаваемые вами классы C++ они должны быть включены.

Последовательные структуры, такие как массивы или связанные списки, в первую очередь приходят в голову при выборе способа реализации очередей с приоритетом. Если последовательность отсортирована, извлечь наименьший элемент несложно, но трудно добавить новый. Для неупорядоченной структуры ситуация будет обратной. В табл. 14.1 отражены данные о производительности различных структур на n-элементном наборе.

---

[1] Поскольку в таком наборе может содержаться несколько копий одного элемента, более точным было бы назвать его «мультинабором» или «мультимножеством» (multiset, bag)

Таблица 14.1. Производительность различных реализаций очередей с приоритетом

Структура	Вставка	Извлечение	n вставок и извлечений
Упорядоченная последовательность	$O(n)$	$O(1)$	$O(n^2)$
Кучи	$O(\log n)$	$O(\log n)$	$O(n \log n)$
Неупорядоченная последовательность	$O(1)$	$O(n)$	$O(n^2)$

Хотя положение элемента может быть определено с помощью двоичного поиска за время $O(n)$, перемещение элементов для освобождения места для нового требует $O(n)$ операций. Если вы забыли о различии между алгоритмами, выполняющимися за время $O(n^2)$ и $O(n \log n)$, загляните в раздел 8.5 главы 8: когда n=1 000 000, первая программа выполнится за три часа, а вторая — за одну секунду.

Реализация очередей с приоритетом с помощью куч — это середина между двумя крайностями. Набор из n элементов представляется с помощью массива x[1..n], обладающего свойством кучи, причем x объявляется в языках C или C++ как x[maxsize+1] (элемент x[0] не используется). Инициализация пустым множеством осуществляется с помощью присваивания n = 0. Добавляя новый элемент, мы увеличиваем n и помещаем этот элемент в x[n]. При этом возникает ситуация, для исправления которой и была придумана функция siftup: утверждение heap(1, n-1) оказывается истинным, а heap(1, n) — в общем случае ложным. Код, осуществляющий добавление нового элемента, приведен на листинге 14.4.

Листинг 14.4. Добавление нового элемента в очередь с приоритетом

```
void insert(t)
   if n >= maxsize
         /* вывести сообщение об ошибке */
   n++
   x[n] = t
   /* heap(1, n-1) */
   siftup(n)
   heap(1, n)
```

Функция extractmin находит наименьший элемент набора, удаляет его, а затем производит реструктурирование массива для восстановления свойства кучи. Поскольку массив является кучей, его наименьший элемент — x[1]. Оставшиеся n-1 элементов набора находятся в подмассиве x[2..n], обладающем свойством кучи. Истинность высказывания heap(1, n) восстанавливается в 2 этапа. Сначала x[n] перемещается в x[1] и уменьшается n, после чего элементы набора снова находятся в подмассиве x[1..n], а высказывание heap(2, n) оказывается истинным. На втором этапе мы вызываем функцию siftdown, которая и восстанавливает истинность heap(1, n). Код функции extractmin получился простым. Он приведен в листинге 14.5.

Листинг 14.5. Функция извлечения наименьшего элемента из очереди с приоритетом

```
int extractmin()
   if n < 1
         /* вывести сообщение об ошибке */
   t = x[1]
   x[1] = x[n--]
```

```
/* heap(2, n) */
siftdown(n)
/* heap(1, n) */
return t
```

Функции insert и extractmin на куче размерностью n выполняются за время $O(n)$.

В листинге 14.6 приведен полный код реализации очередей с приоритетом на C++.

Листинг 14.6. Реализация очередей с приоритетом на C++

```
template<class T>
class priqueue {
private
   int n, maxsize;
   T *x;
   void swap(int i, int j)
   {     T t = x[i]; x[i] = x[j]; x[j] = t; }
public
   priqueue(int m)
   {     maxsize = m;
         x = new T[maxsize+1];
         n = 0;
   }
   void insert(T t)
   {     int i, p;
         x[++n] = t;
         for (i = n; i > 1 && x[p=i/2] > x[i]; i=p)
               swap(p, i);
   }
   T extractmin()
   {     int i, c;
         T t = x[1];
         x[1] = x[n--];
         for (i = 1; (c = 2*i) <= n; i = c) {
               if (c+1 <= n && x[c+1] < x[c])
                     c++;
               if (x[i] <= x[c])
                     break;
               swap(c, i);
         }
         return t;
   }
};
```

Наш простейший интерфейс не содержит средств проверки ошибок и деструктора, но в нем отлично выражены алгоритмические основы данной реализации очередей с приоритетом. В отличие от нашего развернутого псевдокода, этот код написан максимально сжато.

14.4. Алгоритм сортировки

Очереди с приоритетом подсказывают простой алгоритм сортировки вектора: сначала элементы один за другим помещаются в очередь, а затем извлекаются оттуда. Такой алгоритм легко закодировать на C++, используя класс priqueue (листинг 14.7).

Листинг 14.7. Сортировка с помощью очереди с приоритетом

```
template<class T>
void pqsort(T v[], int n)
{  priqueue<T> pq(n);
   int i;
   for (i = 0; i < n; i++)
         pq.insert(v[i]);
   for (i = 0; i < n; i++)
         v[i] = pq.extractmin();
}
```

Операции вставки и извлечения элементов общим количеством $2n$ в худшем случае выполняются за время $O(n \log n)$, что превосходит характеристику алгоритма быстрой сортировки из главы 11 для худшего случая — $O(n^2)$. К сожалению, дополнительный массив x[0..n], в котором размещается куча, занимает n+1 слов памяти.

Перейдем к сортировке с помощью кучи (heapsort), которая по некоторым параметрам превосходит только что описанный алгоритм. Эта сортировка описывается более коротким алгоритмом, требует меньше памяти (поскольку не задействует дополнительный массив) и выполняется быстрее. При описании этого алгоритма мы будем предполагать, что функции siftup и siftdown были изменены так, чтобы на вершине кучи оказывался наибольший элемент. Этого проще всего достичь, поменяв знаки < и > на противоположные.

Наш первый алгоритм работал с двумя массивами, один из которых содержал сортируемые элементы, а второй использовался для представления очереди. Сортировка с помощью кучи экономит память, работая с одним массивом. Один-единственный массив x в этой реализации представляет две абстрактные структуры: кучу с левого конца и изначально неупорядоченную последовательность элементов с правого, причем к концу работы эта последовательность оказывается отсортирована. На рис. 14.8 показано, как меняется состояние массива с течением времени. Массив отложен по горизонтальной оси, а время — по вертикальной.

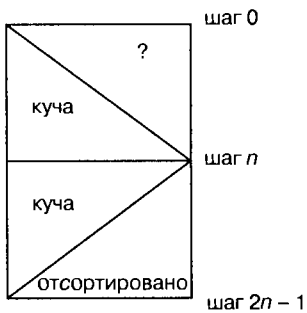


Рис. 14.8. Сортировка с помощью кучи (heapsort)

Алгоритм сортировки с помощью кучи работает в два этапа. На первых n шагах массив переводится в кучу, а на следующих n шагах элементы извлекаются из кучи в порядке убывания и образуют упорядоченную последовательность элементов, увеличивающихся слева направо.

Итак, на первом этапе формируется куча. На протяжении этого процесса инвариант может быть изображен следующим образом (рис. 14.9).

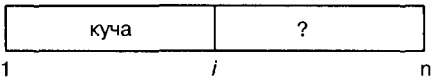


Рис. 14.9. Инвариант на первом этапе

Программа в листинге 14.8 делает истинным высказывание heap(1, n), просеивая элементы к началу массива.

Листинг 14.8. Первый этап сортировки heapsort: формирование кучи

```
for i = [2, n]
   /* инвариант: heap(1, i-1) */
   siftup(i)
   /* heap(1, i) */
```

На втором этапе куча используется для формирования упорядоченной последовательности. Инвариант этого этапа изображен на рис. 14.10.

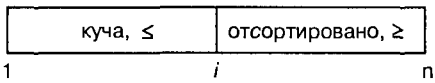


Рис. 14.10. Инвариант на втором этапе

В цикле инвариант сохраняется с помощью двух операций. Поскольку элемент x[1] является наибольшим среди первых i элементов, перестановка его с i-м элементом добавляет к упорядоченной последовательности еще один элемент. Эта перестановка нарушает свойство кучи, которое восстанавливается просеиванием нового верхнего элемента вниз. Код, выполняемый на втором этапе, приведен в листинге 14.9.

Листинг 14.9. Второй этап сортировки heapsort: формирование упорядоченной последовательности

```
for (i = n; i >= 2; i--)
   /* heap(1, i) && sorted(i+1, n) && x[1..i] <= x[i+1..n] */
   swap(1, i)
   /* heap(2, i-1) && sorted(i, n) && x[1..i-1] <= x[i..n] */
   siftdown(i-1)
   /* heap(1, i-1) && sorted(i, n) && x[1..i-1] <= x[i..n] */
```

Теперь, с готовыми функциями, мы можем записать весь алгоритм сортировки heapsort в пять строк (листинг 14.10).

Листинг 14.10. Алгоритм сортировки с помощью кучи (heapsort)

```
for i = [2, n]
   siftup(i)
for (i = n, i >= 2; i--)
   swap(1, i)
   siftdown(i-1)
```

Поскольку в алгоритме выполняется по n-1 вызовов siftup и siftdown, каждый из которых работает не медленнее, чем за $O(\log n)$, программа целиком работает за время $O(n \log n)$ даже в худшем случае.

В решениях задач 2 и 3 описаны методы, позволяющие ускорить (и упростить) алгоритм сортировки с помощью кучи (heapsort). Хотя этот алгоритм гарантирует хорошую скорость в худшем случае — $O(n \log n)$ — самый быстрый вариант его

реализации работает обычно медленнее, чем простой алгоритм быстрой сортировки из раздела 11.2 главы 11.

14.5. Принципы

Эффективность

Свойство формы гарантирует, что все узлы кучи находятся не глубже уровня $\log_2 n$. Функции siftup и siftdown оказываются эффективными потому, что дерево хорошо сбалансировано. Сортировка с помощью кучи исключает необходимость введения добавочного массива, совмещая две абстрактные структуры (кучу и последовательность) в одном массиве.

Правильность

Чтобы правильно написать цикл, мы начинаем с точной формулировки его инварианта. При выполнении операторов в теле цикла инвариант сохраняется. Свойства формы и порядка представляют собой новую форму инварианта — они являются инвариантными свойствами структуры кучи. При написании функции, работающей с кучей, можно предполагать наличие этих свойств в начале ее работы, причем нужно следить за тем, чтобы свойства эти сохранились к моменту завершения функции.

Абстракция

Хорошие системные инженеры всегда разделяют внешний вид компоненты (абстракцию, видную пользователю) и ее внутреннее устройство (содержимое черного ящика). В этой главе черные ящики заполняются двумя способами: с помощью абстрагирования процедур и типов данных.

Абстрагирование процедур

Функция сортировки может быть использована для сортировки массива без знания ее внутреннего устройства. При этом сортировка рассматривается как одна операция. Функции siftup и siftdown обеспечивают аналогичный уровень абстракции: при построении очередей с приоритетом и программы heapsort мы не заботились о внутреннем устройстве этих функций; нам достаточно было знать, *что* они делают (реорганизуют массив с нарушенным на одном из концов свойством кучи). Правильный подход к разработке дал нам возможность определить внутренне содержимое черных ящиков однажды, а затем собрать из них две разные программы.

Абстрактные типы данных

Назначение типа данных определяется его методами и описанием этих методов, а соответствие типа данных своему назначению определяется реализацией компонентов. Мы можем воспользоваться классом priqueue из этой главы или классом IntSet из предыдущей главы, используя только их описание (спецификацию). Конечно, их внутренняя реализация может сказаться на производительности программы.

14.6. Задачи

1. Реализуйте очереди с приоритетом с помощью куч так, чтобы достичь максимальной производительности. При каких значениях n они работают быстрее, чем последовательные структуры?
2. Измените функцию siftdown так, чтобы она соответствовала следующей спецификации:

```
void siftdown(l, u)
      предусловие  heap(l+1, u)
      постусловие· heap(l, u)
```

   Каким будет время работы такого кода? Покажите, как можно использовать этот алгоритм для формирования кучи за время $O(n)$ и получить, таким образом, более быстрый вариант сортировки heapsort, который к тому же оказывается короче исходного.
3. Реализуйте сортировку Heapsort так, чтобы производительность была максимальной. Насколько быстро работает эта программа по сравнению с другими из таблицы в разделе 11.3 главы 11?
4. Используйте механизм куч при реализации очередей с приоритетом для решения приведенных ниже задач (подумайте, как изменится ваш ответ, если станет известно, что входные данные уже упорядочены):
   а) составить таблицу кодов Хаффмана (описаны в большинстве учебников по теории информации и структурам данных);
   б) вычислить сумму большого количества чисел с плавающей точкой;
   в) найти миллион наибольших чисел из миллиарда, хранящихся в файле;
   г) выполнить слияние большого количества небольших упорядоченных файлов в один упорядоченный файл (задача возникает при реализации программы сортировки слиянием, работающей с дисками, — раздел 1.3 главы 1).
5. Задача об упаковке корзин состоит в раскладывании n грузов, каждый из которых лежит в диапазоне 0..1, по минимальному количеству корзин, вместимость которых различна и определена заранее. Первый эвристический метод решения этой задачи состоит в последовательном рассмотрении предлагаемых грузов и помещении их в первую подходящую по объему корзину (корзины рассматриваются в порядке возрастания вместимости). Девид Джонсон (David Johnson) показал, что этот эвристический алгоритм может быть реализован за время $O(n \log n)$. Покажите, как это можно сделать.
6. Типичная реализация файлов с последовательным доступом на диске организуется с помощью указателей, хранящихся внутри блоков и указывающих на следующий за данным блок файла. Для записи блока в этом методе требуется постоянное время (при первой записи файла); считывание первого и i-го блоков (после того, как считан i-1 блок) также требует постоянного времени. Считывание i-го блока выполняется за время, пропорциональное i. Когда Эд Маккрейт (Ed McCreight) разрабатывал контроллер жесткого

диска в исследовательском центре Ксерокс Пало Альто, он выяснил, что добавление еще одного указателя позволяет сохранить все прочие свойства, но при этом считывать i-й блок файла за время, пропорциональное логарифму i. Как бы вы реализовали эту идею? Объясните, что общего имеет алгоритм считывания i-го блока с программой возведения числа в степень i за $\log i$ операций в задаче 9 из главы 4.

7. На некоторых компьютерах самой дорогостоящей операцией в программе двоичного поиска является деление пополам, используемое для определения середины диапазона. Покажите, как можно заменить эту операцию операцией умножения в том случае, если массив правильно организован. Придумайте алгоритмы построения такого массива и поиска в нем.
8. Как можно реализовать очередь с приоритетом для чисел из диапазона [0, k) при условии, что средний размер очереди много больше k?
9. Покажите, что логарифмы времен выполнения insert и extractmin в реализации очередей с приоритетом с помощью куч отличаются от оптимального алгоритма для этих задач не более, чем на постоянный множитель.
10. Принцип, лежащий в основе куч, знаком всем спортивным болельщикам. Пусть Брайан победил Ала, а Линн победил Петера, а потом Линн победил Брайана в матче за звание чемпиона. Результаты обычно изображаются следующим образом (рис. 14.11).

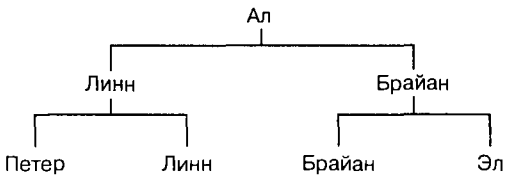


Рис. 14.11. Результаты полуфинальных матчей и финала

Такие деревья результатов часто рисуют для тенниса, баскетбола, футбола и бейсбола. Если предположить, что результаты матчей всегда справедливы и разумны (то есть побеждает реально сильнейший, что не всегда верно в атлетике), какова будет вероятность, что второй по силе игрок выйдет в финал? Придумайте алгоритм равномерного распределения игроков в соответствии с их рейтингом до чемпионата.

11. Как реализованы кучи, очереди с приоритетом и сортировка heapsort в стандартной библиотеке шаблонов C++ STL?

14.7. Дополнительная литература

Отличные учебники по теории алгоритмов, написанные Кнутом и Седжвиком, уже были упомянуты в разделе 11.6 главы 11. Кучи и сортировка heapsort являются предметом исследования Кнута в разделе 5.2.3 его книги «Сортировка и поиск». Очереди с приоритетом и сортировка heapsort описаны в главе 9 книги Седжвика «Алгоритмы».

Жемчужная строка

Нас окружают строки. Битовые строки составляют целые и дробные числа. Строки цифр образуют телефонные номера, а строки символов — слова. Из длинных строк символов получаются web-страницы, а из очень длинных — книги. Невообразимо длинные строки из букв[1] А, Ц, Г и Т хранятся в базах данных генетиков и в клетках читателей книги.

Разнообразные программы выполняют великое множество операций со строками. Строки можно сортировать, считать, искать и анализировать на наличие повторов. В настоящей главе эти темы представлены несколькими классическими задачами.

15.1. Слова

Первая задача: сформировать список слов, содержащихся в документе. Если такой программе «скормить» несколько сотен книг — из получившегося в результате списка слов можно будет составить неплохой словарь. Но что такое слово? Мы используем тривиальное определение: слово — это последовательность символов, заключенная в пробелы. Однако при этом web-страницы будут содержать множество слов наподобие <html>, <body> и . В задаче 1 вам предлагается подумать на эту тему.

Наша первая программа на C++ будет использовать наборы и строки из стандартной библиотеки шаблонов. Эта программа (листинг 15.1) — слегка измененная версия программы из решения задачи 1 к главе 1.

Листинг 15.1. Формирование списка используемых в документе слов

```
int main (void)
{  set<string> S.
```

[1] Первые буквы названий нуклеотидов: Аденин, Гуанин, Цитозин и Тимин, образующих цепочку молекулы ДНК. — *Примеч. ред.*

```
  set<string>  iterator j,
  string t,
  while (cin >> t)
    S.insert(t),
  for (j = S begin(), j != S end(), ++j)
    cout << *j << "\n",
  return 0;
}
```

Цикл while считывает входные данные и помещает слова в набор S (по спецификации STL набор не может содержать одинаковых элементов). В цикле for мы перебираем элементы набора и выводим их в лексикографическом порядке. Эта программа элегантна и достаточно эффективна (подробнее об эффективности мы поговорим позже).

В следующей задаче нам нужно будет подсчитать количество повторений слова в документе (составить частотный словарь). В табл. 15.1 приведены самые часто встречающиеся слова в Библии короля Иакова[1], отсортированные в порядке убывания частоты.

Таблица 15.1. Частотный словарь Библии

the	62053	shall	9756	they	6890
and	38546	he	9506	be	6672
of	34375	unto	8929	is	6595
to	13352	I	8699	with	5949
And	12734	his	8352	not	5840
that	12428	a	7940	all	5238
in	12154	for	7139	thou	4629

Почти 8% из 789 616 слов книги — это слово *the*. По нашему определению «слова», слова *And* и *and* считаются одним и тем же словом.

Эти данные были получены с помощью приведенной в листинге 15.2 программы на C++, также использующей стандартную библиотеку шаблонов, а именно функцию map — для сопоставления счетчика каждой строке.

Листинг 15.2. Программа формирования частотного словаря

```
int main(void)
{  map<string, int> M,
  map<string, int>  iterator j,
  string t,
  while (cin >> t)
    M[t]++,
  for (j = M begin(), j != M end(), ++j)
    cout << j->first << " " << j->second << "\n"
  return 0,
}
```

В цикле while все слова из t помещаются в таблицу M. При этом одновременно увеличиваются соответствующие счетчики, изначально инициализируемые нулями. В цикле for осуществляется перебор слов и вывод их (first) вместе со счетчиками (second).

[1] В англоязычном христианском мире наибольшей популярностью пользуется перевод Библии времен короля Иакова (Библия Иакова), изданный впервые в 1611 году. — *Примеч. ред.*

Эта программа на C++ проста, ясна и эффективна. На моем компьютере она обрабатывает текст Библии за 7,6 секунды. Примерно 2,4 секунды уходит на считывание файла, 4,9 секунды на вставку слов и 0,3 секунды на вывод результатов.

Время работы может быть уменьшено с помощью самостоятельно построенной хэш-таблицы. В узле таблицы будет находиться указатель на слово, счетчик повторений слова и указатель на следующий узел. На рис. 15.1 изображена такая таблица после помещения в нее строк in, the и in, в той маловероятной ситуации, когда функция хэширования даст для всех трех строк одно и то же значение 1:

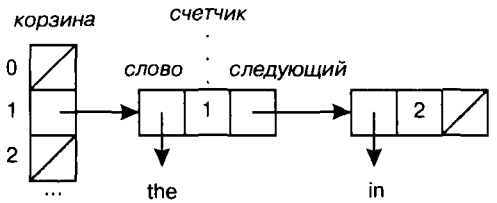


Рис. 15.1. Таблица хэширования после добавления в нее трех элементов

Таблицу эту мы реализуем на языке C с помощью структуры, приведенной в листинге 15.3.

Листинг 15.3. Реализация таблицы хэширования

```
typedef struct node *nodeptr;
typedef struct node {
  char *word;
  int count;
  nodeptr next;
} node;
```

Даже если воспользоваться нашим широким определением «слова», в Библии будет всего 29 131 такое слово. Мы воспользуемся старым правилом для определения размера таблицы: нужно выбрать ближайшее к предполагаемому количеству элементов простое число. Кроме того, мы используем популярное значение множителя — 31.

```
#define NHASH 29989
#define MULT 31
nodeptr bin[NHASH];
```

Наша функция хэширования будет отображать строку во множество натуральных чисел, не превышающих NHASH:

Листинг 15.4. Хэш-функция

```
unsigned int hash(char *p)
  unsigned int h = 0;
  for ( ; *p; p++)
    h = MULT * h + *p;
  return h % NHASH;
```

Использование беззнаковых целых чисел гарантирует положительность h.

Функция main инициализирует все корзины нулевыми указателями, считывает слова и увеличивает счетчик каждого из слов, после чего перебирает содержи-

мое таблицы хэширования, выводя слова (в произвольном порядке) и количество их повторений.

Листинг 15.5. Функция main программы составления частотного словаря

```
int main(void)
  for i = [0, NHASH)
    bin[i] = NULL
  while scanf("%s", buf) != EOF
    incword(buf)
  for i = [0, NHASH)
    for (p = bin[i]; p != NULL; p = p->next)
      print p->word, p->count
  return 0
```

Основная часть работы осуществляется функцией incword (листинг 15.6), увеличивающей значение счетчика данного слова (или инициализирующей его, если такое слово еще не встречалось).

Листинг 15.6. Функция увеличения счетчика для данного слова

```
void incword(char *s)
  h = hash(s)
  for (p = bin[h], p != NULL, p = p->next)
    if strcmp(s, p->word) == 0
      (p->count)++
      return
  p = malloc(sizeof(hashnode))
  p->count = 1
  p->word = malloc(strlen(s)+1)
  strcpy(p->word, s)
  p->next = bin[h]
  bin[h] = p
```

В цикле for просматриваются все узлы с одинаковым значением хэш-функции. Если искомое слово найдено, его счетчик увеличивается и происходит возврат из функции. Если слово в списке отсутствует, функция создает новый узел, выделяет память под него и копирует туда строку (опытные программисты на C воспользовались бы функцией strdup), после чего вставляет узел в начало списка.

Этой программе на C требуется 2,4 секунды для считывания входных данных (как и программе на C++), но набор статистики осуществляется всего за 0,5 с (против 4,9 для программы на C++), а результаты выводятся за 0,06 с (было 0,3). Итак, полное время выполнения новой программы при обработке Библии — 3 секунды ровно (было 7,6), а чистое время обработки — 0,55 с (было 5,2 с). Наша самодельная хэш-таблица, реализованная в 30 строках кода на языке C, оказалась на порядок быстрее, чем таблицы из стандартной библиотеки шаблонов C++ STL.

На этом примере я продемонстрировал два представления набора слов. Сбалансированные деревья поиска работают со строками как с неделимыми объектами. Эти структуры используются в большинстве реализаций наборов и таблиц в STL (set, map). Элементы в таких структурах всегда упорядочены, поэтому операции поиска предшествующего элемента и последовательного вывода выполняются достаточно быстро. Хэширование требует вычисления значения функции хэширования на основании внутреннего содержимого объекта (строки). Строки разбрасываются по большой таблице. В среднем этот метод работает очень быстро,

но в худшем случае он сильно проигрывает сбалансированным деревьям с гарантированным временем выполнения. Кроме того, такая таблица не поддерживает операции, использующие свойство порядка.

15.2. Фразы

Слова — основа любого документа. Множество важных задач можно решить поиском слов. Однако иногда приходится искать фразы наподобие «поиск подстроки» или «неявные структуры данных» в длинных строках (в собственных документах, файлах справки, web-страницах или вообще во всей сети).

Как бы вы стали искать в большом тексте «фразу из нескольких слов»? Если вы никогда не видели этого текста, вам пришлось бы начать с начала и просмотреть его целиком. В большинстве учебников по теории алгоритмов описываются различные подходы к решению задачи «поиска подстроки».

Предположим, что у вас есть возможность заранее подготовить текст перед тем, как осуществлять поиск. Можно сделать хэш-таблицу или дерево поиска для индексации всех слов документа и хранить в ней список всех вхождений всех слов документа. Такой обратный индекс позволит программе быстро найти в тексте любое слово. Фразы можно разделять на отдельные слова, но это сложно реализовать и такая программа может оказаться медленной. Однако в некоторых поисковых системах в Интернете для индексации web-страниц используется именно этот подход.

Теперь мы возьмем мощную структуру данных и используем ее в небольшой задаче. Нужно найти самую длинную повторяющуюся подстроку символов в большом текстовом файле. Например, самая длинная повторяющаяся подстрока в фразе «Ask not what your country can do for you, but what you can do for your country» — «can do for you», тогда как подстрока «your country» стоит на втором месте. Как бы вы написали программу, решающую эту задачу?

Здесь уместно вспомнить задачу об анаграммах из раздела 2.4 главы 2. Если входная строка хранится в массиве `c [0..n-1]`, мы можем начать с последовательного сравнения всех возможных пар подстрок, написав что-нибудь подобное:

```
maxlen = -1
for i = [0, n)
  for j = (i, n)
    if (thislen = comlen(&c[i], &c[j])) > maxlen
      maxlen = thislen
      maxi = i
      maxj = j
```

Функция `comlen` (листинг 15.7) возвращает длину одинаковой части двух строк (начиная с первых символов), передаваемых ей в качестве аргументов.

Листинг 15.7. Функция `comlen` — вычисление длины общей части двух строк

```
int comlen(char *p, char *q)
  i = 0
  while *p && (*p++ == *q++)
    i++
  return i
```

Поскольку этот алгоритм проверяет все возможные пары подстрок, он является квадратичным. Мы смогли бы ускорить его с помощью хэш-таблицы для поиска слов в фразах, но вместо этого мы попробуем новый подход.

Наша новая программа будет обрабатывать не более MAXN символов, хранящихся в массиве c:

```
#define MAXN 5000000
char c[MAXN], *a[MAXN];
```

Мы воспользуемся простой структурой данных, известной как «массив остатков». Она используется по меньшей мере с начала 70-х, хотя термин появился только в 90-х. Структура эта представляет собой массив a указателей на символы. По мере прочтения входного файла содержимое массива a инициализируется таким образом, чтобы соответствующие элементы указывали на символы входной строки:

```
while (ch = getchar()) != EOF
  a[n] = &c[n];
  c[n++] = ch;
c[n] = 0;
```

Последний элемент массива c содержит символ \0, завершающий все строки.

Элемент a[0] указывает на всю строку целиком. Следующий элемент указывает на остаток массива, начинающийся со второго символа, и так далее. Для входной строки «banana» элементам массива будут соответствовать следующие строки:

```
a[0]: banana
a[1]: anana
a[2]: nana
a[3]: ana
a[4]: na
a[5]: a
```

Теперь понятно, почему такой массив называется «массивом остатков».

Если длинная строка дважды входит в массив c, она окажется в двух остатках (то есть два элемента массива a будут начинаться с этой подстроки). Поэтому мы отсортируем массив a, чтобы одинаковые остатки оказались рядом (так же как анаграммы в разделе 2.4 главы 2). Массив для слова «banana» после сортировки станет таким:

```
a[0]: a
a[1]: ana
a[2]: anana
a[3]: banana
a[4]: na
a[5]: nana
```

Затем мы просканируем этот массив, сравнивая соседние элементы для выявления повторяющейся подстроки максимальной длины. В данном случае это будет слово «ana».

Массив остатков мы отсортируем с помощью функции qsort:

```
qsort(a, n, sizeof(char *), pstrcmp);
```

Функция сравнения pstrcmp добавляет один уровень косвенности к библиотечной функции strcmp. Для вычисления количества совпадающих букв при сканировании массива используется функция comlen.

Листинг 15.8. Алгоритм нахождения повторяющейся подстроки максимальной длины

```
for i = [0, n)
  if comlen(a[i], a[i+1]) > maxlen
    maxlen = comlen(a[i], a[i+1])
    maxi = i
  printf("%.*s\n", maxlen, a[maxi])
```

В операторе printf длина строки задается символом *, что позволяет напечатать ровно maxlen символов этой строки.

Я запустил получившуюся программу, чтобы найти самую длинную повторяющуюся подстроку в «Илиаде» Гомера (перевод Сэмюэля Батлера, 807 503 символа). Программа нашла приведенную ниже строку за 4,8 секунды:

whose sake so many of the Achaeans have died at Troy, far from their homes? Go about at once among the host, and speak fairly to them, man by man, that they draw not their ships into the sea

Первый раз эта фраза попадается в тексте, когда Юнона[1] говорит Минерве[2] о том, что нужно предотвратить уход греков (ахейцев) из-под стен Трои. Фраза повторяется, когда Минерва дословно пересказывает ее Улиссу[3]. На этом и нескольких других типичных текстовых файлах алгоритм работает за время $O(n \log n)$, поскольку в нем используется сортировка.

Массив остатков представляет все подстроки n символов исходного текста с помощью n дополнительных указателей. В задаче 7 вам предлагается использовать такой массив для решения задачи поиска подстроки. Мы же перейдем к более изящному способу применения таких массивов.

15.3. Порождение текста

Как можно создать случайный текст? Классический подход: дать волю несчастной обезьяне за ее старенькой печатной машинкой. Если вероятности нажатия на любую клавишу в нижнем регистре и на пробел одинаковы, то результат может выглядеть, например, следующим образом:

uzlpcbizdmddk nhsdzyyvfgxbgjjgbtsak rqvpgnsbyputvqqdtmgkltz ynqotqigexjumqphuj cfwn ll jiexpyqzgsdllgcoluphl sefsrvqqytjakmav bfusvirsjl wprwqt

Определение «английский» в применении к этому тексту выглядит как-то неубедительно.

Если подсчитать количество букв в играх вроде Scrabble или Boggle, выяснится, что оно различно для различных букв. Букв A, к примеру, гораздо больше, чем Z. Обезьяна могла бы напечатать более правдоподобный текст, подсчитав количество букв в некотором документе. Если буква A встретилась 300 раз, а буква B — 100, то обезьяна должна нажимать на клавишу A в среднем в три раза чаще. Это несколько приблизит нас к английскому языку:

saade ve mw hac n entt da k eethetocusosselalwo gx fgrsnoh,tvettaf aetnlbilo fc lhd okleutsndyeoshtbogo eet ib nheaoopefni ngent

---

[1] В русском переводе это была Гера. — *Примеч. ред.*

[2] Афине. — *Примеч. ред.*

[3] Одиссею. — *Примеч. ред.*

Большая часть вещей имеет смысл в контекстном рассмотрении. Предположим, что мы хотим случайным образом сформировать последовательность средних температур (по Фаренгейту) за все дни года. Последовательность из 365 случайных целых чисел между 0 и 100 никого не обманет. Мы могли бы сделать ее более убедительной, если бы сделали сегодняшнюю температуру случайной функцией, зависящей от вчерашней температуры. Если сегодня 85 градусов[1], завтра вряд ли будет 15.

То же верно и для английского языка. Если в тексте встретилась буква Q, следующая наверняка будет U. Генератор текста может дать более интересный результат, если будет это учитывать. Мы могли бы обработать некоторый текст и подсчитать, сколько раз каждая буква встречается после буквы A, сколько раз после B и так далее. Затем при составлении случайного текста мы бы учли эти вероятности при вычислении последующей буквы с учетом текущей. Текст «первого порядка» был получен именно таким образом.

1-й порядок: *t I amy, vin. id wht omanly heay atuss n macon aresethe hired boutwhe t, tl, ad tourest t plur I wit hengamind tarer-plarody thishand.*

2-й порядок: *Ther I the heingoind of-pleat, blur it dwere wing waske hat trooss. Yout lar on wassing, an sit." "Yould," "I that vide was nots ther.*

3-й порядок: *I has them the saw the secorrow. And wintails on my my ent, thinks, fore voyager lanated the been elsed helder was of him a very free bottlemarkable.*

4-й порядок: *His heard." "Exactly he very glad trouble, and by Hopkins! That it on of the who diffecentralia. He rushed likely?" "Blood night that.*

Идею можно распространить на последовательности символов произвольной длины. Текст второго порядка был получен формированием каждой буквы в зависимости от предыдущих двух (пара букв часто называется диграммой). За диграммой TH в английском обычно следуют гласные A, E, I, O, U и Y, реже — согласные R и W, и еще реже прочие буквы алфавита. Текст третьего порядка был получен путем формирования очередной буквы по трем предыдущим (триграмма). В тексте четвертого порядка почти все слова уже выглядят нормальными, и вас, думаю, не удивит, если я скажу, что он был получен из рассказа Конан-Дойля «The Adventure of Abbey Grange». Один из тех людей, кто читал черновик этой главы, обладал классическим образованием. Он отметил, что эта последовательность отрывков напомнила ему переход от староанглийского языка к языку Викторианской эпохи.

Читатели с математическим образованием могут увидеть здесь цепи Маркова. Каждое состояние представляется k-граммой, и вероятности перехода от одного состояния к другому не меняются, поэтому это можно назвать *«цепью Маркова с конечным числом состояний и стационарными вероятностями перехода»*.

Мы можем формировать случайный текст на уровне слов. Самый примитивный подход заключается в случайном выборе слов из словаря. Более «продвинутый» состоит в анализе некоторого документа, подсчете количества слов и корректировании вероятностей появления новых слов соответствующим образом (для решения такой задачи можно использовать программы из раздела 15.1). Более интересный текст можно получить с помощью цепей Маркова, учитывающих несколько предыдущих слов для получения следующего. Вот случайный текст, полученный из черновика первых 14 глав этой книги.

---

[1] По Фаренгейту. — *Примеч. ред.*

1-й порядок: *The table shows how many contexts; it uses two or equal to the sparse matrices were not chosen. In Section 13.1, for a more efficient that "the more time was published by calling recursive structure translates to build scaffolding to try to know of selected and testing and more robust and a binary search).*

2-й порядок: *The program is guided by verification ideas, and the second errs in the STL implementation (which guarantees good worst-case performance), and is especially rich in speedups due to Gordon Bell. Everything should be to use a macro: for n = 10,000, its run time; that point Martin picked up from his desk.*

3-й порядок: *A Quicksort would be quite efficient for the main-memory sorts and it requires only a few distinct values in this particular problem, we can write them all down in the program, and they were making progress towards a solution at a snail's pace.*

Текст первого порядка почти можно читать вслух, тогда как текст 3 порядка содержит длинные фразы из оригинала без искажений, но со случайными переходами от одной к другой. Для пародий лучше всего использовать тексты второго порядка.

Первый раз я столкнулся с генерацией английского текста на уровне букв и слов с различными порядками цепей в классической книге Шеннона «Математическая теория связи». Шеннон пишет: *«Чтобы получить, к примеру, [текст 1 порядка на уровне букв] следует открыть книгу на случайной странице и выбрать случайную букву с этой страницы, которую и следует выписать в текст. Затем нужно открыть книгу на другой случайной странице, найти там эту же букву и выписать следующую за ней. На следующей случайной странице следует найти вторую букву и выписать последующую, и так далее. Аналогичный процесс использовался для [получения текста 1-го и 2-го порядков на уровне букв и 0-го и 1-го порядков на уровне слов]. Интересно было бы построить дальнейшие приближения, но задача на следующих порядках оказывается слишком трудоемкой»*.

Компьютер позволяет автоматизировать выполнение этой трудоемкой задачи. Программа генерации цепей Маркова k-го порядка сможет поместить не более пяти мегабайт входного текста в массив inputchars:

```
int k = 2
char inputchars[5000000];
char *word[1000000];
int nword = 0;
```

Алгоритм Шеннона можно реализовать непосредственным поиском по всему тексту, хотя он и может оказаться неэффективным для больших объемов. Вместо этого мы задействуем массив word, аналогичный массиву остатков за исключением того, что его элементы всегда указывают на начала слов (типичная модификация). В переменной nword хранится количество слов. Файл считывается следующим кодом:

```
word[0] = inputchars
while scanf("%s", word[nword]) != EOF
  word[nword+1] = word[nword] + strlen(word[nword]) + 1
  nword++
```

Все слова добавляются к массиву inputchars (дополнительной памяти под них отводить не нужно), а завершаются слова символом \0, добавляемым функцией scanf автоматически.

После считывания исходного текста мы отсортируем массив word, чтобы все элементы, указывающие на последовательности с одинаковым началом, оказались рядом. Функция wordncmp из листинга 15.9 осуществляет сравнение последовательностей слов.

Листинг 15.9. Сравнение последовательностей слов

```
int wordncmp(char *p, char* q)
  n = k
  for ( ; *p == *q; p++, q++)
    if (*p == 0 && --n == 0)
      return 0;
  return *p - *q
```

Строки сравниваются до тех пор, пока их символы совпадают. При достижении символа \0 счетчик n уменьшается на 1, а после обнаружения k одинаковых слов происходит выход из функции. Если обнаруживаются различия, функция возвращает разность строк.

Считав текст, мы добавим к нему k символов \0, чтобы функция сравнения не дала сбоя в конце массива, выведем первые k слов документа (чтобы инициализировать наш генератор), а затем вызовем функцию сортировки.

Листинг 15.10. Вызов функции сортировки для массива остатков

```
for i = [0, k)
  word[nword][i] = 0
for i = [0, k)
  print word[i]
qsort(word, nword, sizeof(word[0]), sortcmp)
```

Функция sortcmp, как и раньше, добавляет «уровень косвенности» к указателям.

Наша эффективно использующая память структура содержит большой объем информации о содержащихся в тексте k-граммах. Если k == 1 и на вход подан текст «*of the people, by the people, for the people*», массив word может выглядеть следующим образом:

word[0]: by the
word[1]: for the
word[2]: of the
word[3]: people
word[4]: people, for
word[5]: people, by
word[6]: the people,
word[7]: the people
word[8]: the people,

Для простоты в этом списка показаны только первые k+1 слово из всей строки, на которую указывает элемент массива. Если нам нужно продолжить фразу, заканчивающуюся словом *the*, мы обратимся к массиву остатков и получим три возможных варианта: два «*people,*» и один «*people*».

Теперь мы можем генерировать абракадабру с помощью приведенного в листинге 15.11 псевдокода.

Листинг 15.11. Набросок программы порождения бессмысленного текста

```
фраза = первая фраза входного массива
цикл
  ищем фразу в массиве word[0..nword-1] с помощью двоичного поиска
  из всех фраз с первыми k одинаковыми словами выбираем одну, на которую
указывает p
  фраза = слово, следующее за p
  если k-е слово фразы имеет длину 0
    выход
  вывод k-го слова фразы
```

Цикл инициализируется присваиванием переменной фраза первых символов входного текста (вспомните, что эти символы уже выведены в выходной файл). Двоичный поиск (программа из раздела 9.3) позволяет найти первое вхождение фразы в массиве остатков (важно найти именное первое вхождение, а программа из раздела 9.3 ориентирована именно на это). В следующем цикле проверяются все равные фразы, а решение 12.10 используется для случайного выбора одой из них. Если k-е слово этой фразы имеет нулевую длину, эта фраза является последней в документе, поэтому цикл заканчивается.

В листинге 15.12 приведен полный текст псевдокода, реализующего эти идеи и вводящего ограничение на количество порождаемых слов.

Листинг 15.12. Бредогенератор

```
phrase = inputchars
for (wordsleft = 10000, wordsleft > 0, wordsleft--)
  l = -1
  u = nword
  while l+1 != u
    m = (l + u) / 2
    if wordncmp(word[m], phrase) < 0
      l = m
    else
      u = m
  for (i = 0; wordncmp(phrase, word[u+i]) == 0, i++)
    if rand() % (i+1) == 0
      p = word[u+i]
  phrase = skip(p, 1)
  if strlen(skip(phrase, k-1)) == 0
    break
  print skip(phrase, k-1)
```

Глава 3 книги Кернигана и Пайка «Практика программирования» (о которой рассказывается в разделе 5.9 главы 5) посвящена общим вопросам «разработки и реализации». В качестве примера авторы приводят программу порождения случайного текста с помощью цепей Маркова на уровне слов, поскольку «это типично для большинства программ: какие-то данные поступают на вход, какие-то получаются на выходе, а в процессе обработки выполняются какие-то хитрые действия». В их книге рассказывается история этой задачи и приводятся примеры реализации программ на языках C, Java, C++, Awk и Perl.

Программа, написанная нами, оказывается в выигрыше при сравнении с их программой на C. Код примерно вдвое короче; представление фразы с помощью указателя на k последовательных слов эффективно с точки зрения экономии памяти

и легко в реализации. При объеме входных данных около 1 Мбайт программы работают с приблизительно одинаковой скоростью. Поскольку Керниган и Пайк используют структуру данных, занимающую несколько больший объем, и выделяют память с помощью малоэффективного оператора malloc, наша программа расходует приблизительно на порядок меньше памяти (на моем компьютере). Если оптимизировать ее с учетом решения задачи 14 и заменить двоичный поиск и сортировку таблицей хэширования, программа из этого раздела станет работать вдвое быстрее (но при этом объем используемой памяти увеличится примерно на 50%).

15.4. Принципы

Задачи со строками

Каким образом ваш компилятор ищет имя переменной в таблице символов? Каким образом справочная система быстро просматривает содержимое компакт-диска по мере ввода символов строки запроса? Как поисковые машины ищут фразы в сети? В этих важных задачах используются методы, рассмотренные нами на примерах этой главы.

Структуры данных для хранения строк

Мы изучили несколько наиболее важных структур, используемых для представления строк в памяти.

Хэширование

Хэш-структура является весьма быстрой и ее легко реализовать.

Сбалансированные деревья

Эти структуры гарантируют хорошую производительность даже в худшем случае. Они включены в большинство реализаций стандартной библиотеки шаблонов C++ STL (структуры set, map).

Массивы остатков

Создайте массив указателей, проинициализируйте его так, чтобы каждый элемент указывал на свой символ строки, затем отсортируйте его и получится массив остатков. Этот массив позволяет находить схожие строки или искать слова и фразы с помощью двоичного поиска.

В разделе 13.8 главы 13 показан пример использования некоторых других структур для представления слов в словаре.

Библиотеки или «самодельные» компоненты?

Наборы, таблицы и строки библиотеки C++ STL весьма удобны в использовании, но их общий и мощный интерфейс приводит к тому, что специализированная функ-

ция хэширования работает быстрее. Другие компоненты библиотеки очень эффективны: наша функция хэширования вызывала strcmp, а для реализации массива остатков мы воспользовались qsort. Чтобы написать функции двоичного поиска и сравнения слов в программе порождения марковского текста, я подсмотрел, как реализованы библиотечные функции bsearch и strcmp.

15.5. Задачи

1. Повсюду в тексте главы мы использовали простое определение: слово — последовательность символов, заключенных в пробелы. Во множестве реальных форматов документов (HTML или RTF, к примеру) содержатся команды форматирования. Как можно обработать такие команды, не включив их в список слов? Может ли возникнуть необходимость выполнить какие-либо другие действия, чтобы в списке слов оказались действительно слова?
2. Как можно использовать библиотечные структуры set и map для решения задачи поиска из раздела 13.8 главы 13 на компьютере с большим объемом оперативной памяти? Каковы будут затраты памяти по сравнению с программой Макилроя?
3. Насколько можно ускорить программу хэширования из раздела 15.1, если в нее добавить специальный вызов malloc (для однократного выделения памяти)?
4. Если таблица велика и функция хэширования хорошо разбрасывает по ней данные, почти все списки в этой таблице оказываются короткими. Если одно из этих условий оказывается нарушенным, время поиска элемента в списке существенно возрастает. Если мы не находим некоторую строку в таблице хэширования из раздела 15.1, то она помещается в начало списка. Чтобы промоделировать ситуацию с небольшой таблицей, установите NHASH = 1 и поэкспериментируйте с этой и другими стратегиями формирования списка. Например, новый элемент можно дописывать не к началу, а к концу списка или перемещать элементы при их успешном обнаружении к его началу.
5. В разделе 15.1 частотный словарь выводился в порядке убывания частоты слов. Как можно изменить программу на C и C++, чтобы достигнуть этого? Как можно вывести только M первых по частоте повторения слов, если M — некоторая константа (например, 100 или 1000)?
6. Как найти подстроку, лучше всего совпадающую с данной, с помощью массива остатков? Как бы вы реализовали графический интерфейс пользователя для этой задачи?
7. Наша программа поиска повторяющихся строк работала очень быстро для «типичных» текстов, но для некоторых специально подобранных последовательностей она может работать очень медленно (медленнее, чем за $O(n^2)$). Измерьте скорость работы программы в такой ситуации. Может ли реально встретиться такой входной текст?

8. Как бы вы изменили программу поиска повторяющихся строк, чтобы она находила самую длинную строку, встречающуюся в тексте не менее M раз?
9. Если даны два текста, как найти самую длинную общую подстроку?
10. Покажите, как можно уменьшить количество указателей в программе поиска повторяющихся строк, если устанавливать их только на начала слов? Как это повлияет на выводимый программой текст?
11. Реализуйте программу для порождения марковского текста на уровне букв.
12. Как бы вы применили средства и методы из раздела 15.1 для порождения случайного текста порядка 0 (то есть не-марковского)?
13. Программа, порождающая марковский текст на уровне слов, есть на сайте этой книги. Исследуйте ее на своих текстах.
14. Как бы вы применили хэширование для увеличения скорости работы программы порождения марковского текста?
15. Цитата из Шеннона в разделе 15.3 относится к алгоритму, с помощью которого он порождал марковский текст. Реализуйте этот алгоритм с помощью программы. Она дает хорошее, но не совсем точное приближение к марковским частотам. Объясните, почему. Реализуйте программу, сканирующую всю строку целиком для порождения каждого слова (и следовательно, использующую реальные значения частот).
16. Как бы вы применили методы этой главы для подготовки списка слов для словаря? С этой задачей столкнулся Дуг Макилрой (раздел 13.8 главы 13). Как бы вы написали программу проверки орфографии *без* словаря? Как бы вы написали программу проверки грамматики без внесения в нее грамматических правил?
17. Исследуйте, как методики, относящиеся к анализу k-грамм, используются в приложениях типа систем распознавания речи, сжатия данных и им подобных.

15.6. Дополнительная литература

Большая часть книг, упомянутых в разделе 8.8, содержит информацию об эффективных алгоритмах и удобных структурах данных, применяемых для представления и обработки строк.

Эпилог к первому изданию

Интервью с автором показалось мне тогда наилучшим завершением первого издания книги. Оно все еще не потеряло своей актуальности, так что я вновь привожу его здесь.

Вопрос: *Спасибо, что согласились ответить на мои вопросы* (в дальнейшем вопросы выделены *курсивом*).

Ответ: Не за что: мое время — это и ваше время.

Все главы этой книги уже были напечатаны в качестве статей в Communications of the ACM. Почему вы решили собрать их вместе и выпустить книгу?

Существует несколько причин. Я исправил десятки ошибок, сделал сотни небольших улучшений и добавил несколько новых разделов. В книге на 50% больше задач, решений и иллюстраций. Кроме того, удобнее работать с книгой, чем со статьями, разбросанными по десятку журналов. Главная же причина заключается в том, что предметы обсуждения этих глав лучше проявляются, когда они собраны вместе. Целое оказывается большим, чем сумма его частей.

И какие же это предметы?

Самое главное заключается в том, что углубиться в размышления о программе может быть интересно и полезно. Наша работа — это нечто большее, чем просто систематическая разработка программы по техническим требованиям. Если эта книга поможет хотя бы одному программисту, утратившему иллюзии, снова полюбить свою работу, я буду считать цель достигнутой.

Какой-то сбивчивый ответ. Есть ли какие-то технические нити, связывающие главы воедино?

Часть вторая посвящена производительности, но производительность является общим вопросом всех глав. В нескольких главах интенсивно используются методы верификации программ. В приложении 1 приведен каталог алгоритмов, используемых в книге.

Судя по всему, большая часть глав посвящена процессу разработки. Можете ли вы дать какой-то общий совет на эту тему?

Я рад, что вы спросили. Я случайно подготовил список советов перед этим интервью. Вот десять предложений для программистов.

1. Решайте правильную задачу.
2. Исследуйте пространство решений.
3. Изучите данные.
4. Делайте предварительные оценки.
5. Учитывайте симметрию задачи.
6. Применяйте компоненты.
7. Создавайте прототипы.
8. Идите на необходимые компромиссы.
9. Не усложняйте сверх необходимого.
10. Стремитесь к изящности решения.

Эти советы мы приводим в контексте задач программирования, но они применимы к любой инженерной деятельности.

Здесь возникает вопрос, который давно меня беспокоил: легко упрощать маленькие программы в книге, но подходят ли эти методы для больших программных систем?

У меня есть три ответа: да, нет и возможно. Да, они подходят. В разделе 3.4 (первого издания) описан большой проект, который был упрощен до «всего лишь» 80 человеко-лет. Настолько же верен и ответ «нет»: если правильно упрощать, в большой системе просто не будет необходимости, поэтому к ней не придется применять эти методы. Хотя оба ответа достойны, истина, как всегда, лежит где-то посередине, и вот отсюда возникает ответ «возможно». Некоторые программы не могут не быть большими, и некоторые методы из этой книги могут к ним подойти. Операционная система Unix — это хороший пример мощного целого, построенного из простых и изящных частей.

Вот вы снова говорите о системе, созданной в Bell Labs. Не является ли эта книга слишком уж предвзятой?

Ну, возможно, немного. Я обращался к материалу из собственного опыта, и поэтому примеры относятся к тому, что меня окружало. Большая часть материала этих глав была предложена моими коллегами, и они заслужили похвалу (или упреки). Я многому научился от исследователей и разработчиков в Bell Labs. Мы работаем в атмосфере сотрудничества, стимулирующей взаимодействие между исследованиями и разработками. Поэтому то, что вы называете предвзятостью, на самом деле является проявлением восторженного отношения к компании.

Давайте спустимся на землю. Чего в этой книге не хватает?

Я надеялся включить в нее пример большой системы, состоящей из множества программ, но не смог найти такую систему, которую можно было бы описать на десяти страницах (типичный размер одной статьи). Я хотел бы добавить в нее несколько глав на тему «информатика для программистов» (наподобие глав о верификации программ или разработке алгоритмов) и «инженерные методы вычислений» (пример: предварительные оценки в главе 7).

Если вы так увлекаетесь «наукой» и «инженерией», почему в этих главах так мало теорем и таблиц и так много рассказов?

Очень просто: людям, которые берут интервью сами у себя, не следует критиковать стиль.

Эпилог ко второму изданию

Некоторые традиции существуют, потому что они этого заслуживают. Другие просто существуют.

С возвращением! Давно не виделись.

Четырнадцать лет.

Начнем с того же места, что и в прошлый раз. Почему вы решили выпустить новое издание?

Просто мне очень нравится эта книга. Ее было очень приятно писать, и читателям она тоже понравилась. Принципы выдержали проверку временем, но многие примеры из первого издания безнадежно устарели. В наше время читатель не может считать «большим» компьютер с 512 Кбайт ОЗУ.

И что же вы изменили в этом издании?

Все то, о чем я говорил в предисловии. Вы что, вообще к интервью не готовитесь?

Извините. Я вижу, что в предисловии вы отметили, что тексты программ книги доступны на web-сайте.

Писать программы было интереснее всего. Я реализовал большую часть программ для первого издания, но видел их тексты только я сам. Для этого издания я написал около 2500 строк на C и C++, которые теперь доступны всему миру.

Вы сказали, что код доступен всему миру? Я читал некоторые из ваших программ. Ужасный стиль! Короткие, даже микроскопические имена переменных, причудливые описания функций, глобальные переменные, которые должны были бы быть параметрами, и этот список можно продолжать бесконечно. Вам не стыдно думать о том, что другие программисты увидят ваши программы?

Мой стиль действительно может оказаться фатальным для больших программных проектов. Однако эта книга не является большим программным проектом. Ее даже нельзя назвать большой книгой. В решении задачи 1 из главы 5 говорится о сжатом стиле программирования и о том, почему я его выбрал. Если бы я хотел написать книгу на 1000 страниц, я бы выбрал более развернутый стиль программирования.

Если говорить о длинном коде, ваша программа sort.cpp измеряет быстродействие библиотечной функции языка C qsort, функции sort из C++ STL и нескольких самодельных быстрых сортировок. Вы можете все-таки сделать выбор? Следует ли программисту пользоваться библиотечными функциями, или писать свои?

Том Дафф дал лучший ответ на этот вопрос: «Крадите код всегда, когда это возможно». Библиотеки — это здорово. Применяйте их всегда, когда они выполняют свою задачу. Начните с системной библиотеки, затем ищите подходящие функции во внешних библиотеках. Однако в инженерной деятельности не всегда можно обойтись стандартными деталями. Когда библиотечные функции не справляются с задачей, пишите свои собственные. Надеюсь, что фрагменты псевдокода из этой книги (и реальный код с web-страницы) помогут начать эту нелегкую работу программистам, которым придется с этой книгой столкнуться. Мне кажется, что подход к тестированию и экспериментам поможет таким программистам оценить возможности множества алгоритмов и выбрать для своего приложения наиболее подходящий.

Кроме общедоступного кода и обновленных баек, что действительно нового есть в этой книге?

Я попытался рассказать об оптимизации с учетом наличия кэш-памяти и параллелизма на уровне инструкций. Вообще говоря, три новые главы отражают три основных изменения в новом издании: глава 5 описывает реальный код и тестовые программы, глава 13 подробно рассказывает о структурах данных, а в главе 15 выводятся сложные алгоритмы. Большая часть идей, собранных в этой книге, уже, так или иначе, появлялись на страницах других книг, но модель стоимости памяти из приложения 3 и алгоритм порождения марковского текста из раздела 15.3 впервые появились именно здесь. Новый алгоритм порождения марковского текста выигрывает в сравнении с классическим алгоритмом Кернигана и Пайка.

Ну вот, опять люди из Bell Labs. В прошлый раз вы были полны энтузиазма по поводу этого заведения, но тогда вы работали в нем всего несколько лет. За последние 14 лет в Bell Labs многое изменилось. Что вы теперь думаете об этой компании и об изменениях?

Когда я писал первые главы книги, лаборатории Bell Labs были частью фирмы Bell System. Когда первое издание было опубликовано, мы стали частью AT&T, а сейчас мы входим в состав Lucent Technologies. Компании, индустрия телекоммуникаций и поле применения компьютеров существенно изменились за эти годы. Лаборатория Bell Labs старалась идти в ногу с этими изменениями и часто была их инициатором. Я пришел в лабораторию, потому что мне нравится совмещать теорию и практику, создавать программы и писать книги. Маятник много раз качнулся в разные стороны за те годы, что я работаю в компании, но мое руководство всегда поощряло широкий диапазон видов деятельности.

Один из рецензентов первого издания написал: «Среда, в которой Бентли работает каждый день, — это нирвана для программиста. Он является членом технического персонала лаборатории Bell Labs в городе Мюррей Хилл, штат Нью Джерси, обладает доступом к самому современному программному и аппаратному обеспечению и стоит в очереди в буфет вместе с самыми известными в мире разработчиками и программистами». Bell Labs все еще остается именно таким местом.

Нирвана семь дней в неделю?

Нирвана почти каждый день, а во все остальные дни — просто чудесная жизнь.

Приложения

Каталог алгоритмов

Эта книга содержит большую часть материала курса теории алгоритмов для колледжа, но рассматривает этот материал с другой точки зрения. Упор делается в первую очередь на применение знаний и кодирование программ, а не на математический анализ. В этом приложении материал приведен в более стандартизованной форме. Ссылки на задачи приведены в форме M.N, где M — номер главы, N — номер задачи в главе.

Сортировка

- *Постановка задачи.* Выходная последовательность — это упорядоченная перестановка входной последовательности. Если на входе — дисковый файл, на выходе, как правило, другой файл; если на входе — массив, на выходе, как правило, тот же самый массив.
- *Приложения.* Приведенный ниже список всего лишь намекает на разнообразие приложений сортировки.
  - *Требования к выходной информации.* Некоторые пользователи требуют упорядочения выходной информации; см. раздел 1.1 и ваш телефонный справочник. Функции поиска, аналогичные двоичному, требуют предварительной сортировки.
  - *Группировка одинаковых элементов.* Программисты используют сортировку, чтобы сгруппировать вместе одинаковые элементы: программа поиска анаграмм из разделов 2.4 и 2.8 собирает вместе слова из одного класса анаграмм. Массивы остатков из разделов 15.2 и 15.3 собирают вместе похожие фразы. См. также задачи 2.6, 8.10 и 15.8.

- *Другие приложения.* Программа поиска анаграмм из разделов 2.4 и 2.8 использовала сортировку для получения канонического порядка следования букв слова, что позволяло получить сигнатуру класса анаграмм. В задаче 2.7 сортировка используется для реорганизации данных на ленте.

- *Функции общего назначения.* Приведенные ниже алгоритмы сортируют произвольную последовательность из n элементов.

  - *Сортировка вставкой (insertion sort).* Программа в разделе 11.1 работала за время $O(n^2)$ как в худшем случае, так и для случайных входных данных. Время выполнения нескольких ее версий приведено в таблице в этом разделе. В разделе 11.3 сортировка вставкой использовалась для упорядочения почти отсортированного массива за время $O(n)$. Это единственная стабильная сортировка в этой книге: элементы с равными ключами оказываются на выходе в том же порядке, в котором они были на входе.

  - *Быстрая сортировка (quicksort).* Простая быстрая сортировка в разделе 11.2 работает за время $O(n \log n)$ на массиве из n различных элементов. Она реализуется рекурсивно и использует логарифмически растущий объем памяти на стеке. В худшем случае она работает за время $O(n^2)$ и требует $O(n)$ памяти. Время $O(n^2)$ достигается на массиве одинаковых элементов; улучшенная версия быстрой сортировки из раздела 11.3 работает за время $O(n \log n)$ на любом массиве. Таблица в разделе 11.3 содержит эмпирические данные о времени работы нескольких реализаций быстрой сортировки. Стандартная библиотечная функция языка C qsort обычно реализуется на основе этого алгоритма. Она применяется в разделах 2.8, 15.2 и 15.3, а также в решении 1.1. Функция стандартной библиотеки шаблонов C++ STL sort часто использует этот алгоритм. Время ее работы также измерено в разделе 11.3.

  - *Сортировка с помощью кучи (heapsort).* Сортировка с помощью кучи из раздела 14.4 всегда работает за время $O(n \log n)$. Она не рекурсивна и использует небольшой постоянный объем дополнительной памяти. В решениях 14.1 и 14.2 описаны более быстрые версии сортировки heapsort.

  - *Другие алгоритмы сортировки.* Для сортировки файлов эффективен алгоритм сортировки слиянием (merge sort), который мы вкратце описали в разделе 1.3. Схема алгоритма слияния набросана в задаче 14.4.4. Решение 11.6 содержит псевдокод сортировки выбором и сортировки Шелла (selection sort, Shell sort).

  - *Время выполнения* нескольких алгоритмов сортировки описано в решении 1.3.

- *Специальные функции.* Эти функции позволяют писать короткие и эффективные программы для некоторых типов входных данных.

  - *Поразрядная сортировка.* Сортировка битовых строк из задачи 11.5, придуманная Макилроем, может быть обобщена для сортировки строк на других алфавитах (байтах, например).

- *Сортировка с битовым массивом.* Сортировка из раздела 1.4 использует тот факт, что числа берутся из небольшого диапазона, без повторений и дополнительных данных. Детали реализации и возможные расширения описаны в задачах 1.2, 1.3, 1.5 и 1.6.
- *Прочие сортировки.* Многопроходная сортировка в разделе 1.3 считывает входной файл много раз, экономя память за счет быстроты выполнения. В главах 12 и 13 порождаются упорядоченные наборы случайных чисел.

Поиск

- *Постановка задачи.* Функция поиска определяет, является ли заданный ей на входе элемент членом данного множества, и иногда выдает связанную с ним информацию.
- *Приложения.* Программа-справочник телефонных номеров Леска в задаче 2.6 ищет имя в базе данных, возвращая соответствующий ему телефонный номер. Шахматная программа Томпсона из раздела 10.8 искала текущее расположение фигур в базе данных, вычисляя оптимальный ход. Программа проверки правописания Макилроя из раздела 13.8 осуществляет поиск в словаре. Другие приложения поиска описаны в соответствующих функциях.
- *Функции общего назначения.* Приведенные ниже алгоритмы предназначены для поиска в произвольном n-элементном множестве.
  - *Последовательный поиск.* Простые и оптимизированные версии последовательного поиска для массивов приводятся в разделе 9.2. Последовательный поиск для массивов и связанных списков описан в разделе 13.2. Этот алгоритм используется для расстановки переносов (задача 3.5), сглаживания географических данных (раздел 9.2), представления разреженных матриц (раздел 10.2), порождения случайных наборов (раздел 13.2), сортировки сжатого словаря (раздел 13.8), упаковки корзин (задача 14.5) и поиска одинаковых фраз в тексте (раздел 15.3). Введение к главе 3 и задача 3.1 описывают два глупых варианта реализации последовательного поиска.
  - *Двоичный поиск.* Алгоритм поиска в отсортированном массиве за $\log_2 n$ сравнений описан в разделе 2.2, а код его разработан в разделе 4.2. В разделе 9.3 код был расширен для поиска первого вхождения элемента, а также оптимизирован. Приложения включают поиск записей в системе резервирования (раздел 2.2), строк с ошибками (раздел 2.2), анаграмм входного слова (задача 2.1), телефонных номеров (задача 2.6), положения точки между отрезками (задача 4.7), индекса записи в разреженном массиве (решение 10.2), случайных чисел (задача 13.3) и фраз (разде-

лы 15.2 и 15.3). В задачах 2.9 и 9.9 обсуждаются компромиссы между двоичным и последовательным поиском.

- *Хэширование.* В задаче 1.10 используется хэширование телефонных номеров. В задаче 9.10 хэшируется набор целых чисел; в разделе 13.4 набор корзин используется для хэширования набора целых чисел, и наконец в разделе 13.8 хэшируются слова в словаре. В разделе 15.1 хэширование используется для подсчета слов в документе.
- *Двоичные деревья поиска.* В разделе 13.3 используется (несбалансированное) двоичное дерево поиска для представления списка случайных целых. Сбалансированные деревья обычно используются при реализации шаблона set из стандартной библиотеки шаблонов C++ STL, которым мы пользовались в разделах 13.1 и 15.1 и решении 1.1.

- *Специальные функции.* Эти функции позволяют писать короткие и эффективные программы для некоторых типов входных данных.
  - *Индексация ключей.* Некоторые ключи могут быть использованы в качестве индекса в массиве значений. Корзины и битовые векторы в разделе 13.4 используют в качестве индексов числовые ключи. Используемые в качестве индексов ключи бывают следующих типов: телефонные номера (раздел 1.4), символы (решение 9.6), аргументы тригонометрических функций (задача 9.11), индексы разреженных массивов (раздел 10.2), значения счетчика программы (задача 10.8), номера шахматных комбинаций (раздел 10.8), случайные целые числа (раздел 13.4), значения хэш-функции для строки и целые значения в очередях с приоритетом.
  - *Прочие методы.* В разделе 8.1 показано уменьшение времени выполнения программы за счет помещения всех элементов в кэш. В разделе 10.1 описывается существенное упрощение программы поиска в таблице налогов после понимания контекста задачи.

Прочие алгоритмы на множествах

В этих задачах обрабатываются n-элементные множества, в которых допускается наличие повторяющихся элементов.

- *Очереди с приоритетом.* Очередь с приоритетом поддерживает операции добавления элемента и извлечения наименьшего элемента. В разделе 14.3 описаны две последовательных структуры, позволяющие реализовать очередь, и приводится пример класса на C++, эффективно реализующего очередь с приоритетом с помощью кучи. Приложения очередей описаны в задачах 14.4, 14.5 и 14.8.

- *Выборка.* В задаче 2.8 описана проблема выбора k-го минимума. Решение 11.9 предлагает эффективный алгоритм для этой задачи. Альтернативные алгоритмы упоминаются в задачах 2.8, 11.1 и 14.4.3.

Алгоритмы на строках

В разделах 2.4 и 2.8 мы искали в словаре группы анаграмм. В решении 9.6 предлагается несколько способов классификации символов. В разделе 15.1 вводилось определение слов и производился подсчет их количества в некотором файле; составлялся частотный словарь, сначала с помощью компонентов стандартной библиотеки шаблонов C++ STL, а затем с помощью самодельной таблицы хэширования. В разделе 15.2 массив остатков использовался для поиска самой длинной повторяющейся подстроки некоторого текста, а в разделе 15.3 используется другой массив остатков для порождения случайного текста по модели Маркова.

Алгоритмы с векторами и матрицами

Алгоритмы обмена местами двух подпоследовательностей вектора описаны в разделе 2.3 и задачах 2.3 и 2.4. В решении 2.3 эти алгоритмы реализованы в коде. В задаче 2.5 описан алгоритм обмена разделенных подпоследовательностей вектора. В задаче 2.7 сортировка применяется для транспонирования матрицы, записанной на магнитной ленте. Программы вычисления максимального значения в массиве описаны в задачах 4.9, 9.4 и 9.8. Алгоритмы экономии памяти при работе с векторами и матрицами описаны в разделах 10.3 и 14.4. Разреженные векторы и матрицы обсуждаются в разделах 3.1, 10.2 и 13.8. В задаче 1.9 описана схема инициализации разреженных векторов, которая позже применяется в разделе 11.3. Глава 8 содержит описания пяти алгоритмов нахождения подпоследовательности вектора с максимальной суммой элементов, и некоторые задачи этой главы также относятся к векторам и матрицам.

Случайные объекты

Функции порождения псевдослучайных целых чисел используются в книге повсюду; они реализуются в решении 12.1. Раздел 12.3 описывает алгоритм перемешивания элементов массива. Разделы 12.1–12.3 содержат описание нескольких алгоритмов случайной выборки (см. также задачи 12.7 и 12.9). В задаче 1.4 показан пример применения алгоритма. Решение 12.10 содержит алгоритм случайного выбора одного элемента из неопределенного их количества.

Численные алгоритмы

В решении 2.3 приведен алгоритм Евклида для нахождения наибольшего общего делителя двух целых чисел. В задаче 3.7 показана схема алгоритма оценки линейных рекуррентных соотношений с постоянными коэффициентами. Задача 4.9 содержит код эффективного алгоритма возведения числа в натуральную степень. В задаче 9.11 тригонометрические функции вычисляются с помощью таблицы. В решении 9.12 описана схема Горнера. Суммирование большого количества вещественных чисел описано в задачах 11.1 и 14.4.b.

Умеете ли вы делать оценки?

Предварительные вычисления из главы 7 всегда начинаются с некоторых основных характеристик объектов. Иногда эти числа можно найти в условии задачи (например, в технических требованиях), но обычно их приходится угадывать.

Приведенный ниже тест поможет вам оценить свои способности в качестве предсказателя. Для каждого из вопросов нужно указать верхнюю и нижнюю границы диапазона, в котором, по вашему мнению, с вероятностью 90% находится правильный ответ. Не пытайтесь сделать диапазон слишком широким или слишком узким. По моим оценкам, тест должен занять у вас от пяти до десяти минут. Пожалуйста, будьте добросовестны (и лучше, если вы сделаете копию этой страницы, и будете работать с ней, заботясь о следующем читателе).

1. [_____,_____] Население США на 1 января 2000 года (в миллионах человек).
2. [_____,_____] Год рождения Наполеона.
3. [_____,_____] Длина реки Миссисипи-Миссури в милях.
4. [_____,_____] Максимальный взлетный вес «Боинга-747» в фунтах.
5. [_____,_____] Время распространения радиосигнала от Земли до Луны в секундах.
6. [_____,_____] Географическая широта Лондона.
7. [_____,_____] Время одного оборота «Шаттла» вокруг Земли в минутах.
8. [_____,_____] Длина между башнями моста Golden Gate в футах.
9. [_____,_____] Количество подписей в Декларации Независимости.
10. [_____,_____] Количество костей в теле взрослого человека.

Когда вы закончите отвечать на вопросы, можете переходить к следующей странице, чтобы узнать свой результат и прочитать комментарий к нему.

Пожалуйста, ответьте на все вопросы, прежде чем переворачивать страницу.

Если вы еще не дали ответы на все вопросы самостоятельно, пожалуйста, вернитесь и сделайте это. Вот ответы, взятые из различных альманахов и аналогичных источников.

1. 1 января 2000 года население США составляло 272,5 миллиона человек.
2. Наполеон родился в 1769.
3. Длина реки Миссисипи-Миссури — 3710 миль.
4. Максимальный взлетный вес лайнера B747-400 — 875 000 фунтов.
5. Радиосигнал доходит от Земли до Луны за 1,29 с.
6. Лондон находится на широте около 51,5 градуса.
7. «Шаттл» облетает вокруг Земли примерно за 91 минуту.
8. Между башнями моста Golden Gate 4200 футов.
9. Декларацию подписало 56 человек.
10. В теле взрослого человека 206 костей.

Подсчитайте, какое количество диапазонов в ваших ответах содержали правильные значения. Поскольку вы использовали 90%-ный доверительный интервал, вы должны были угадать примерно 9 ответов из 10.

Если вы правильно ответили на все десять вопросов, у вас могут быть замечательные способности в области предсказания оценок. С другой стороны, вы, возможно, указывали слишком широкие интервалы. Думаю, вы сами выберете, что вам больше подходит.

Если вы правильно ответили на шесть вопросов или менее, вы, наверное, огорчены и смущены так же, как я, когда я впервые прошел подобный тест. Немного практики вам не повредит.

Если вы правильно ответили на семь или восемь вопросов, вы хорошо угадываете. В будущем помните о том, что нужно чуть-чуть расширять границы ваших 90%-ных доверительных интервалов.

Если вы угадали ответы на девять вопросов, то вы, возможно, отлично угадываете. Или вы указали бесконечные интервалы для девяти вопросов и нулевой для одного. Если это так, вам должно быть стыдно.

Модель стоимости времени и памяти

В разделе 7.2 описаны две небольшие программы, определявшие затраты памяти на базовые типы и затраты времени на простейшие операции. В этом приложении рассказывается, как из них можно вырастить полезные программы для получения оценок затрат времени и памяти. Полный исходный код обеих программ можно скачать с сайта книги.

Программа spacemod.cpp определяет объем памяти, занимаемый различными структурами языка C++. Первая часть программы состоит из последовательности операторов наподобие:

```
cout << "sizeof(char)=" <<sizeof(char),
cout << "sizeof(short)=" <<sizeof(short),
```

Это позволяет получить точные результаты для примитивных объектов:

```
sizeof(char)=1,  sizeof(short)=2   sizeof(int)=4,  sizeof(float)=4,
sizeof(struct*)=4,  sizeof(long)=4,  sizeof(double)=8
```

В этой программе определено с десяток структур, в которых используются простые соглашения об именах, иллюстрируемые приведенным ниже примером.

```
struct structc { char c, },
struct structic { int i, char c, },
struct structip { int i, structip *p; },
struct structdc { double d, char c, },
struct structc12 { char c[12], },
```

В программе используется макрос, возвращающий размер структуры и оценивающий количество байтов памяти, выделяемых оператором new.

```
struct        1    48    48 48 48 48 48 48 48 48 48
structic      8    48    48 48 48 48 48 48 48 48 48
structip      8    48    48 48 48 48 48 48 48 48 48
structdc     16    64    64 64 64 64 64 64 64 64 64
structcd     16    64    64 64 64 64 64 64 64 64 64
structcdc    24 -3744  4096 64 64 64 64 64 64 64 64
structiii    12    48    48 48 48 48 48 48 48 48 48
```

Первое число возвращается sizeof, а последующие десять представляют собой разности указателей, возвращаемых последовательными вызовами оператора new. Этот результат достаточно типичен. Большая часть чисел вполне адекватна, но

иногда подпрограмма выделения памяти переходит к новой области (выделяет участки не подряд).

Приведенный ниже макрос формирует одну строку отчета:

```
#define MEASURE(T, text) {  \
  cout << text << "\t";  \
  cout << sizeof(T) << "\t",  \
  int lastp = 0,  \
  for (int i = 0; i < 11, i++) {  \
    T *p = new T;  \
    int thisp = (int) p;  \
    if (lastp != 0)  \
      cout << " " << thisp - lastp,  \
    lastp = thisp,  \
  }  \
  cout << "\n",  \
}
```

Макрос вызывается с именем структуры в качестве первого аргумента и тем же именем в кавычках (для вывода на печать) в качестве второго аргумента.

```
MEASURE(structc,"structc").
```

В первом варианте этой программы использовался шаблон C++, в котором тип структуры играл роль параметра, однако из-за артефактов реализации измерения были не вполне адекватными.

В табл. 1 сведены воедино результаты работы программы на моем компьютере.

Таблица 1. Стоимость памяти

Структура	sizeof	new
int	4	48
structc	1	48
structic	8	48
structip	8	48
structdc	16	64
structcd	16	64
structcdc	24	64
structiii	12	48
structiic	12	48
structc12	12	48
structc13	13	64
structc28	28	64
structc29	29	80

Числа в левом столбце дают возможность оценить размер структуры, возвращаемый sizeof. Начинать следует с суммирования значений sizeof для типов. Это дает 8 байт для structip. Нужно также учитывать необходимость выравнивания, и поэтому, хотя компоненты структуры structcdc дают в сумме 10 байт, на самом деле она занимает 24 байта. Правый столбец дает нам представление о том, насколько неэкономично использует память оператор new. Любая структура, размер которой по sizeof составляет 12 байтов или меньше, при динамическом выделении занимает в памяти 28 байт. Структуры объемом 13–28 байт займут 64 байта памяти. В общем, размер выделенного блока будет кратен 16, и от 36 до 47 байтов будет потрачено зря. Это на удивление расточительно: в других системах для 8-байтовой записи хватало всего лишь 8 байт дополнительной информации.

В разделе 7.2 описана еще одна небольшая программа, позволяющая оценить стоимость выполнения одной конкретной операции на C. Мы можем обобщить ее, полу-

чив программу timemod.c, которая выводит стоимость операций языка C по времени. Программа-предшественница этой была написана Брайаном Кернеганом, Крисом ван Вайком и мной в 1991 году. Функция main состоит из последовательности строк-заголовков (T), за которыми следуют строки M, измеряющие стоимость операций:

```
T("Целочисленная арифметика")
M({}).
M(k++);
M(k = i + j).
M(k = i - j).
```

С помощью этих и аналогичных строк получается следующий результат:

```
Целочисленная арифметика (n=5000)
{}                250     261     250     250     251      10
k++               471     460     471     461     460      19
k = i + j         491     491     500     491     491      20
k = i - j         440     441     441     440     441      18
k = i * j         491     490     491     491     490      20
k = i / j        2414    2433    2424    2423    2414      97
k = i % j        2423    2414    2423    2414    2423      97
k = i & j         491     491     480     491     491      20
k = i | j         440     441     441     440     441      18
```

В первом столбце указана операция, которая выполняется в следующем цикле:

```
for i = [1, n]
  for j = [1, n]
    операция
```

Следующие пять колонок содержат время выполнения такого цикла в тиках (в этой системе один тик равен миллисекунде) для пяти его запусков (все эти времена адекватны; неадекватное время выполнения позволяет выявить ошибки). В последнем столбце указана средняя стоимость одной операции в наносекундах. Первая строка таблицы говорит нам о том, что для одного выполнения тела пустого цикла требуется 10 нс. Следующая строка показывает, что увеличение значения переменной k на единицу отнимает 9 нс. Все арифметические и логические операции требуют примерно одного и того же времени, за исключением операций деления и нахождения остатка, которые требуют времени на порядок больше.

Этот подход позволяет грубо оценить стоимость операций на конкретном компьютере, но результатам не следует придавать слишком большое значение. Все эксперименты проводились с отключенной оптимизацией. Когда я включал ее, оптимизирующий компилятор удалял циклы и все времена оказывались нулевыми.

Работу выполняет макрос M, схема которого может быть записана на псевдокоде так:

```
#define M(op)
  вывод op в виде строки
  timesum = 0
  for trial = [0, trials)
    start = clock()
    for i = [1, n]
      fi = i
      for j = [1, n]
        op
    t = clock() - start
    print t
    timesum += t
  print 1e9*timesum / (n*n*trials*CLOCKS_PER_SEC)
```

Полный код программы, измеряющей стоимость операций по времени, можно найти на сайте книги.

Давайте теперь изучим результаты работы программы на моей системе. Поскольку времена выполнения циклов близки друг к другу, мы их просто опустим и будем приводить только среднее время выполнения одной операции в наносекундах.

```
Арифметика с плавающей точкой (n = 5000)
fj=j.                    18
fj=j.    fk = fi + fj    26
fj=j.    fk = fi - fj    27
fj=j;    fk = fi * fj    24
fj=j.    fk = fi / fj    78
Операции с массивами
k = i + j            17
k = x[i] + j         18
k = i + x[j]         24
k = x[i] + x[j]      27
```

Операции с плавающей точкой начинаются с присваивания значения переменной j переменной fj. Эта операция занимает 8 нс. Во внешнем цикле значение переменной i присваивается переменной fi. Операции с плавающей точкой выполняются почти так же быстро, как и целочисленные. Операции с массивами почти не требуют дополнительных затрат времени. Следующие примеры дают нам информацию о стоимости операторов ветвления и сравнения.

```
Сравнения (n=5000)
if (i < j) k++       20
if (x[i]<x[j]) k++   25
Сравнения массивов и обмен элементов (n=5000)
k = (x[i]<x[k]) ? -1 1    34
k = intcmp(x+i, x+j)      52
swapmac(i, j)             41
swapfunc(i, j)            65
```

Реализация сравнения и перестановки в виде функций стоит в среднем на 20 нс дороже, чем реализация в явном виде. В разделе 9.2 сравнивается стоимость вычисления наибольшего из двух чисел с помощью функций, макросов и встроенного кода.

```
Функция, макрос и встроенный код для операции max (n=5000)
k = (i > j) ? i : j  26
k = maxmac(i, j)     26
k = maxfunc(i, j)    54
```

Функция rand оказалась на удивление дешевой (вспомните, однако, что функция bigrand вызывает ее дважды), извлечение квадратного корня стоит на порядок больше, чем основные арифметические операции (но всего вдвое дороже, чем операция деления), простые тригонометрические функции стоят еще в два раза больше, а сложные выполняются уже за микросекунды.

```
Математические функции (n=1000)
k=rand()          40
fk=j+fi           20
fk=sqrt(j+fi)    188
fk=sin(j+fi)     344
fk=sinh(j+fi)   2229
fk=asin(j+fi)    973
fk=cos(j+fi)     353
fk=tan(j+fi)     465
```

Поскольку эти операции работают медленно, значение n пришлось уменьшить. Выделение памяти стоит еще больше, поэтому мы уменьшаем n еще сильнее.

```
Выделение памяти (n=500)
free(malloc(16))     2484
free(malloc(100))    3044
```

Правила оптимизации программ

Моя книга 1982 года «Пишем эффективные программы» строилась на 27 правилах оптимизации кода. Эта книга уже не переиздается, поэтому я привожу правила здесь (с некоторыми небольшими изменениями), вместе с примерами их использования.

Жертвуем памятью ради скорости

- *Расширение структур данных.* Время выполнения определенных операций с данными может быть уменьшено, если структуру расширить добавлением дополнительной информации или изменить представление данных в этой структуре так, чтобы ускорить к ним доступ.
  - В разделе 9.2 Райт нужно было найти ближайшие соседние точки на поверхности Земли, причем координаты их определялись широтой и долготой, что требовало применения дорогостоящих тригонометрических вычислений. Эппель расширил ее структуру, добавив туда декартовы координаты, что позволило вычислять расстояния по Евклиду, а это существенно быстрее.
- *Вычисляйте результаты заранее и храните их.* Стоимость повторного вычисления дорогостоящей функции может быть уменьшена путем однократного ее вызова и сохранения результата. Последующие вызовы обрабатываются с помощью таблицы, а не самой функцией.
  - *Кумулятивные массивы* в разделе 8.2 и решении 8.11 позволяют заменить последовательность сложений двумя обращениями к таблице и вычитанием.
  - В решении 9.7 программа подсчета битов ускоряется путем обращения к байту или слову целиком за один раз.

  - В решении 10.6 сдвиг и логические операции заменены обращением к таблице.
- *Кэширование.* Данные, к которым чаще всего обращаешься, должны располагаться ближе всего.
  - В разделе 9.1 рассказывается о том, как ван Вайк кэшировал наиболее часто используемый тип данных, исключив дорогостоящие обращения к подсистеме выделения памяти. В решении 9.2 описаны подробности реализации одного из способов кэширования узлов.
  - В главе 13 кэшируются узлы списков, корзин и двоичных деревьев поиска.
  - Кэширование может работать неэффективно и замедлять программу, если обрабатываемые данные не удовлетворяют требованию локальности.
- *Лень в вычислениях.* Стратегия, при которой значения объектов вычисляются только по мере необходимости, позволяет избавиться от вычисления значений тех объектов, которые в действительности не нужны.

Жертвуем скоростью ради памяти

- *Упаковка.* Способы уплотненного представления данных позволяют уменьшить затраты памяти за счет быстроты обращения к этим данным.
  - Способы представления разреженных массивов из раздела 10.2 значительно уменьшают затраты памяти, но несколько увеличивают время доступа к элементам массивов.
  - Словарь Макилроя для программы проверки правописания из раздела 13.8 состоит из 75 000 слов, сжатых до 52 Кб.
  - Массивы Кернигана в разделе 10.3 и сортировка с помощью кучи в разделе 14.4 основаны на методе совместного использования (overlaying) структур, позволяющего уменьшить объем, занимаемый данными, путем совместного их хранения в одной области памяти.
  - Хотя иногда при упаковке и приходится жертвовать временем ради памяти, упакованные данные иногда обрабатываются быстрее, чем неупакованные.
- *Интерпретаторы.* Объем кода программы часто может быть уменьшен с помощью интерпретаторов, позволяющих компактно представить последовательности часто используемых операторов.
  - В разделе 3.2 интерпретатор используется для составления формальных писем, а в разделе 10.4 интерпретатор используется для простой графической программы.

Циклы

- *Выносите код из циклов.* Вместо того чтобы выполнять некоторую операцию в теле цикла каждый раз при его проходе, лучше вызвать ее один раз вне цикла.

  - В разделе 11.1 оператор присваивания значения переменной t выносится из основного цикла программы isort2.
- *Совмещение проверок.* Внутренний цикл должен содержать минимально возможное количество проверок, а лучше всего только одну. Программист должен стараться уменьшить количество условий завершения цикла их объединением.
  - *Маркерные элементы* являются наиболее часто встречающимся примером применения этого правила. Маркерный элемент помещается на границе данных в структуре, чтобы уменьшить стоимость проверки на достижение этой границы. В разделе 9.2 маркерный элемент используется при последовательном поиске в массиве. В главе 13 маркеры применяются для написания ясного и эффективного кода для массивов, связанных списков, корзин и двоичных деревьев поиска. В решении 14.4 маркер ставится на одном из концов кучи.
- *Раскрытие цикла.* Раскрытие цикла позволяет избавиться от затрат на изменение значения индексов, помогает избежать остановов конвейера, уменьшает количество ветвлений и увеличивает параллелизм на уровне инструкций.
  - Раскрытие цикла последовательного поиска в разделе 9.2 уменьшает время его выполнения примерно на 50%, а раскрытие цикла двоичного поиска в разделе 9.3 уменьшает время его выполнения на величину от 35 до 65%.
- *Раскрытие циклов передачи.* Если большая часть внутреннего цикла состоит в тривиальных операциях присваивания, их часто можно удалить из цикла, если перепланировать использование переменных. Чтобы исключить присваивание i = j, нужно в последующем коде ставить вместо переменной i переменную j.
- *Удаление безусловных переходов.* В быстром цикле не должно быть безусловных переходов. Безусловный переход в конце цикла может быть удален путем «поворота» этого цикла таким образом, чтобы в конце его оказалось условное ветвление. Эта операция обычно выполняется оптимизирующими компиляторами.
- *Слияние циклов.* Если два соседних цикла работают с одним и тем же набором элементов, то их тела следует объединить, чтобы осталась только одна управляющая конструкция (заголовок цикла).

Логические правила

- *Используйте алгебраическую эквивалентность.* Дорогостоящую логическую операцию можно заменить на эквивалентную ей алгебраическую, которая будет вычисляться быстрее.
- *Сокращение монотонных функций.* Если нужно проверить, превысила ли неубывающая монотонная функция нескольких переменных некоторый по-

рог, значения оставшихся переменных после достижения порога учитывать уже не нужно.

  - Усложненная версия этого правила позволяет завершать цикл после достижения его цели. Циклы поиска в главах 10, 13 и 15 завершают работу непосредственно после нахождения искомого элемента.

- *Изменение порядка проверок.* Логические проверки должны быть расположены так, чтобы более быстрые условия, которые чаще оказываются правильными, стояли перед более медленными условиями, которые реже оказываются правильными.
  - В решении 9.6 приведена схема проверок, порядок которых может быть изменен.
- *Предварительное вычисление логических функций.* Логическая функция на небольшом множестве исходных значений может быть заменена таблицей, представляющей это множество.
  - В решении 9.6 описана реализация стандартной библиотечной функции классификации символов на C с помощью таблицы.
- *Удаление булевских переменных.* Булевские переменные могут быть убраны из программы с помощью замены операции присваивания такой переменной некоторого значения оператором `if...else`, в котором одна ветвь относится к случаю, когда переменная истинна, а другая — к случаю, когда эта переменная оказывается ложной.

Составление процедур

- Устранение иерархий функций. Время выполнения элементов набора функций, нерекурсивно вызывающих друг друга, может быть сокращено раскрытием этих функций и связыванием передаваемых переменных.
  - Замена функции `max` соответствующим макросом в разделе 9.2 ускорила программу почти вдвое.
  - Раскрытие функции `swap` в разделе 11.1 ускорило программу почти втрое, но в разделе 11.3 это же действие дало гораздо меньший эффект.
- *Учитывайте частоту ситуаций.* Функции должны правильно обрабатывать все возможные ситуации и быть наиболее эффективными в наиболее часто возникающих ситуациях.
  - В разделе 9.1 подпрограмма выделения памяти ван Вайка корректно обрабатывала все типы узлов, но была особенно эффективна для некоторого наиболее часто встречавшегося типа.
  - В разделе 6.1 Эпель обрабатывал наиболее дорогостоящую ситуацию с близко расположенными объектами со специальным уменьшенным шагом по времени, что позволило во всех остальных случаях использовать более эффективный шаг большей величины.

- *Сопрограммы.* Многопроходный алгоритм может быть переделан в однопроходный с помощью сопрограмм.
  - В программе поиска анаграмм из раздела 2.8 используется канал, который может быть реализован как набор сопрограмм.
- *Трансформация рекурсивных функций.* Время выполнения рекурсивных функций может быть уменьшено применением следующих трансформаций:
  - замена рекурсии итерацией, как в программах со списками и двоичными деревьями в главе 13;
  - преобразование рекурсии в итерацию использованием явного стека (если функция содержит только один рекурсивный вызов себя самой, нет необходимости сохранять на стеке адрес возврата);
  - если в самом конце тела функции делается вызов этой же функции, его можно заменить на безусловный переход к началу функции. Это часто называется «удалением концевой рекурсии». Код в решении 11.9 является потенциальным кандидатом на такое преобразование. Такое ветвление часто может быть преобразовано в цикл, и оптимизирующие компиляторы обычно так и поступают;
  - часто оказывается более эффективным решать небольшие подзадачи внешними процедурами, а не выполнять рекурсию до задачи размером 0 или 1. Функция `qsort4` из раздела 11.3 использует значение `cutoff` около 50.
- *Параллелизм.* Программа должна быть написана так, чтобы максимальным образом использовать параллелизм аппаратуры, на которой она выполняется.

Составление выражений

- *Инициализация во время компиляции.* Максимальное количество переменных следует инициализировать до запуска программы.
- *Использование алгебраической эквивалентности.* Если вычисление выражения занимает слишком много времени, его следует заменить на более дешевый эквивалент.
  - В разделе 9.2 Эппель заменил дорогостоящие тригонометрические операции умножениями и сложениями и воспользовался монотонностью функции для того, чтобы избавиться от операции извлечения квадратного корня.
  - В разделе 9.2 дорогостоящий оператор деления с остатком во внутреннем цикле был заменен на более быстрый оператор if.
  - Часто можно умножать и делить на степени двойки с помощью операций битового сдвига в ту или иную сторону. В решении 13.9 деление заменено сдвигом. В решении 10.6 деление на 10 заменено сдвигом на 4.
  - В цикле, перебирающем элементы массива, следует по возможности заменять умножение сложением. Многие оптимизирующие компиляторы

сделают это за вас. Этот метод может быть обобщен на более широкий класс алгоритмов с инкрементом.

- *Удаление одинаковых выражений.* Если одно и то же выражение вычисляется дважды с одинаковыми значениями входящих в него переменных, следует сохранить результат первого вычисления и использовать его вместо того, чтобы вычислять второй раз.
  - Современные компиляторы обычно хорошо справляются с такой задачей самостоятельно, если выражение не содержит вызовов функций.
- *Парное вычисление.* Если два одинаковых выражения часто вычисляются подряд, их следует вынести во внешнюю функцию.
  - В разделе 13.1 первый вариант псевдокода использовал функции member и insert. В коде на C++ пара этих функций заменена на одну функцию insert, которая просто ничего не делает, если элемент уже имеется в наборе.
- *Используйте параллелизм на уровне слов.* При вычислении дорогостоящих выражений используйте всю ширину полосы пропускания данных того компьютера, на котором вы работаете.
  - В задаче 13.8 показано, как можно работать одновременно с несколькими битами битовых векторов с помощью типов char и int.
  - В решении 9.7 используется параллельный подсчет битов.

Классы C++

Ниже приведен полный листинг классов, использовавшихся в главе 13 для представления наборов целых чисел. Исходный код целиком можно скачать с сайта книги.

```
class IntSetSTL {
private:
  set S;
public:
  IntSetSTL(int maxelements, int maxval) { }
  int size() { return S.size(); }
  void insert(int t) { S.insert(t);}
  void report(int *v)
  {  int j = 0;
     set<int>::iterator i;
     for (i = S.begin(); i != S.end(); ++i)
       v[j++] = *i;
  }
};

class IntSetArr {
private:
  int n, *x;
public:
  IntSetArr(int maxelements, int maxval)
  {  x = new int[1 + maxelements];
     n=0;
     x[0] = maxval; /* sentinel at x[n] */
  }
  int size() { return n; }
  void insert(int t)
  {  int i, j;
     for (i = 0; x[i] < t; i++)
       ;
     if (x[i] == t)
```

```
      return.
    for (j = n. j >= i. j--)
      x[j+1] = x[j].
    x[i] = t.
    n++.
  }
  void report(int *v)
  {  for (int i = 0. i < n. i++)
      v[i] = x[i].
  }
}.

class IntSetList {
private
  int  n.
  struct node {
    int val.
    node *next.
    node(int i. node *p) { val = i. next = p. }
  }.
  node *head. *sentinel.
  node *rinsert(node *p. int t)
  {  if (p->val < t) {
      p->next = rinsert(p->next. t).
    } else if (p->val > t) {
      p = new node(t. p).
      n++.
    }
    return p.
  }
public
  IntSetList(int maxelements. int maxval)
  {  sentinel = head = new node(maxval. 0).
    n = 0.
  }
  int size() { return n. }
  void insert(int t) { head = rinsert(head. t). }
  void report(int *v)
  {  int j = 0.
    for (node *p = head. p != sentinel. p = p->next)
      v[j++] = p->val.
  }
}.

class IntSetBST {
private
  int  n. *v. vn.
  struct node {
    int val.
    node *left. *right;
    node(int v) { val = v. left = right = 0. }
  }.
  node *root.
  node *rinsert(node *p. int t)
  {  if (p == 0) {
      p = new node(t).
      n++.
```

```
    } else if (t < p->val) {
      p->left = rinsert(p->left, t);
    } else if (t > p->val) {
      p->right = rinsert(p->right, t);
    } // do nothing if p->val == t
    return p;
  }
  void traverse(node *p)
  {  if (p == 0)
       return;
    traverse(p->left);
    v[vn++] = p->val;
    traverse(p->right);
  }
public:
  IntSetBST(int maxelements, int maxval) { root = 0; n = 0; }
  int size() { return n; }
  void insert(int t) { root = rinsert(root, t); }
  void report(int *x) { v = x; vn = 0; traverse(root); }
};

class IntSetBitVec {
private:
  enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F };
  int  n, hi, *x;
  void set(int i) { x[i>>SHIFT] |= (1<<(i & MASK)); }
  void clr(int i) { x[i>>SHIFT] &=~(1<<(i & MASK)); }
int test(int i) {return x[i>>SHIFT] & (1<<(i& MASK)); }
public:
  IntSetBitVec(int maxelms, int maxval)
  {  hi = maxval;
    x = new int[1+hi/BITSPERWORD];
    for (int i = 0; i < hi; i++)
      clr(i);
    n = 0;
  }
  int size() { return n; }
  void insert(int t)
  {  if (test(t))
       return;
    set(t);
    n++;
  }
  void report(int *v)
  {  int j=0;
    for (int i = 0; i < hi; i++)
      if (test(i))
        v[j++] = i;
  }
};

class IntSetBins {
private:
  int  n, bins, maxval;
  struct node {
    int val;
    node *next;
```

```
    node(int v, node *p) { val = v; next = p; }
  };
  node **bin, *sentinel;
  node *rinsert(node *p, int t)
  {  if (p->val < t) {
       p->next = rinsert(p->next, t);
     } else if (p->val > t) {
       p = new node(t, p);
       n++;
     }
     return p;
  }
public:
  IntSetBins(int maxelements, int pmaxval)
  {  bins = maxelements;
     maxval = pmaxval;
     bin = new node*[bins];
     sentinel = new node(maxval, 0);
     for (int i = 0; i < bins; i++)
       bin[i] = sentinel;
     n = 0;
  }
  int size() { return n; }
  void insert(int t)
  {  int i = t / (1 + maxval/bins);  // CHECK !
     bin[i] = rinsert(bin[i], t);
  }
  void report(int *v)
  {  int j = 0;
     for (int i = 0; i < bins; i++)
       for (node *p = bin[i]; p != sentinel; p = p->next)
         v[j++] = p->val;
  }
};
```

Подсказки к некоторым задачам

Глава 1

4. Прочитайте главу 12.

5. Попробуйте двухпроходный алгоритм.

6, 8, 9. Индексируйте ключом.

10. Испытайте хэширование и не ограничивайте себя компьютерной системой.

11. Это задача для птиц.

12. Чем вы пишете, когда у вас нет авторучки?

Глава 2

1. Подумайте о сортировке, двоичном поиске и сигнатурах.

2. Ищите линейный алгоритм.

5. Используйте равенство $cba = (a'b'c')'$.

7. Высоцкий воспользовался системной утилитой и двумя короткими программами, которые он сам написал как раз для этого случая.

8. Подумайте о k наименьших элементах набора.

9. Стоимость s последовательных поисков пропорциональна s*n. Полная стоимость s двоичных поисков равна стоимости самих поисков и времени, затрачиваемого на сортировку таблицы. Не беспокойтесь о константах слишком сильно, почитайте задачу 9 из главы 9.

10. Как Архимед узнал, что корона короля не из чистого золота?

Глава 3

2. Храните коэффициенты рекурсии в одном массиве, а k предыдущих значений — в другом. Программа должна состоять из двух вложенных циклов.

4. Только одну функцию придется написать целиком, а другие две должны ее вызывать.

Глава 4

2. Работайте с точным инвариантом. Добавьте два лишних элемента в массив, чтобы можно было проинициализировать инвариант значениями x[-1]=-inf, x[n]=inf.

5. Если вы решите эту задачу, бегите на ближайший математический факультет и пишите кандидатскую диссертацию.

6. Ищите инвариант процесса и свяжите начальное состояние банки с ее конечным состоянием.

7. Перечитайте раздел 2.2.

9. Попробуйте приведенные ниже инварианты цикла, которые являются правильными до выполнения проверки в первом операторе while. Для векторного сложения инвариант будет такой:

$i \le n \,\&\&\, \forall_{1 \le j < i} a[j] = b[j] + c[j]$

а для последовательного поиска:

$i \le n \,\&\&\, \forall_{1 \le j < i} x[j] \ne t$

11. В решении задачи 14 из главы 11 приведено решение с рекурсивной функцией, передающей указатель на массив.

Глава 5

3. Ищите термины вроде «проверки мутаций».

5. Что можно выполнить за $O(\log n)$ и $O(n)$ лишних сравнений?

6. На web-сайте этой книги есть программа на языке Java, иллюстрирующая работу алгоритмов сортировки.

9. Формат вывода программы тестирования содержит символы табуляции, что подразумевает совместимость с большинством электронных таблиц. Я обычно сохраняю результаты серии последовательных экспериментов на одном листе таблицы вместе с графиками производительности и комментариями о том, почему я провел эксперименты и что я узнал из каждого из них.

Глава 6

1. Подсмотрите раздел 8.5 главы 8.

3. Измените модель стоимости, описанную в приложении 3, так, чтобы измерять стоимость операций с двойной точностью.

7. Автомобильные катастрофы предотвращаются с помощью некоторых мер, включающих тренировку водителей, жесткое ограничение скорости, ограничения по возрасту для употребления алкоголя, высокие штрафы за нахождение в пьяном виде за рулем, развитие системы общественного транспорта. Если катастрофы происходят, можно снизить ущерб, причиняемый пассажирам, особым устройством салона, ремнями безопасности и воздушными подушками. Если пассажиры полу-

чили повреждения, опасность для жизни пассажиров может быть снижена скорой медицинской помощью, быстрой эвакуацией на вертолете и своевременным хирургическим вмешательством.

Глава 7

5. Сначала я испытал функцию $(1+x/100)^{72/x}$, затем с помощью электронной таблицы построил график функции $(1+0,72/x)^x$. Чтобы доказать свойства «правила 72», вспомните, что $\lim(1+c/n)^n = e^n$, что $\ln 2 \approx 0,693$ и что асимптота — не всегда лучшая аппроксимация.

8. Изучите схемы и программы в задачах 7 к главе 2; 10, 12 и 13 к главе 8; 4 к главе 9; 10 к главе 10; 6 к главе 11; 7, 9 и 11 к главе 12; 3, 6 и 11 к главе 13; 4, 5, 7, 9 и 15 к главе 15, и в разделах 1.3, 2.2, 2.4, 2.8, 10.2, 12.3, 13.2, 13.3, 13.8, 14.3, 14.4, 15.1, 15.2 и 15.3.

Глава 8

4. Стройте кумулятивную сумму в случайном порядке.

7. Сложение с плавающей точкой не всегда коммутативно.

8. Помимо вычисления максимальной суммы области возвращайте информацию о максимальных векторах, заканчивающихся с каждой стороны массива.

10, 11, 12. Используйте кумулятивный массив.

13. Очевидный алгоритм работает за время $O(n^4)$. Попытайтесь написать кубический алгоритм.

Глава 9

3. Сложение увеличило `k` не более чем на `n-1`, поэтому мы знаем, что `k` меньше, чем `2*n`.

9. Чтобы двоичный поиск был сравним по быстроте с линейным даже для небольших `n`, нужно увеличить стоимость операции сравнения (см., к примеру, задачу 7 в главе 4).

Глава 10

1. Какой код для обращения к упакованным полям получился на выходе компилятора?

5. Перемешайте и сопоставьте функции и таблицы.

7. Нужно уменьшить объем данных, считая некоторые области памяти эквивалентными. Эти диапазоны могут быть либо блоками фиксированной длины (например, 64 байта), или функциональными границами.

Глава 11

2. Пусть индекс i изменяется от больших значений к малым, чтобы он приближался к известному значению t в x[l].

4. Когда вам нужно решить две подзадачи, какую из них вы будете решать первой, а какую оставите на стеке для решения во вторую очередь — большую или маленькую?

9. Измените программу быстрой сортировки так, чтобы она осуществляла рекуррентный вызов только для поддиапазона, содержащего k.

Глава 12

4. Идите к специалисту по статистике и произнесите фразы: «*задача сборщика билетов*» и «*парадокс дней рождений*».

11. В задаче сказано, что можно использовать компьютер, но не сказано, что делать это обязательно.

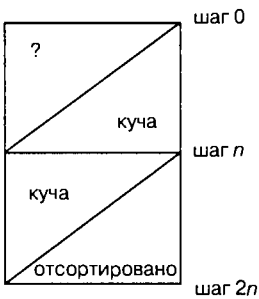
Глава 13

2. Проверять нужно попадание добавляемого элемента в нужный диапазон и наличие свободного места в структуре данных. Деструктор должен освобождать всю занимаемую память.

3. Используйте двоичный поиск, чтобы выяснить, находится ли элемент в упорядоченном массиве.

Глава 14

2. Примените сортировку с помощью кучи к следующей структуре:



3. См. задачу 2. Рассмотрите возможность вынесения операций из тела цикла.

6. В кучах используются неявные указатели с узла i на узел 2*i. Попробуйте реализовать то же самое для файлов.

7. Двоичный поиск в массиве x[0..6] использует неявное дерево, корнем которого является элемент x[3]. Как можно вместо этого использовать неявные деревья из раздела 14.1 главы 14?

9. Используйте нижнюю границу времени сортировки $O(n \log n)$. Если операции insert и extractmin работают быстрее, чем за время $O(\log n)$, вы сможете выполнить сортировку быстрее, чем за $O(n \log n)$. Покажите, как можно быстро сортировать массив с помощью этих операций.

Глава 15

15. Пусть мы порождаем текст Маркова первого порядка из документа с миллионом слов, содержащего слова x, y и z в одной фразе: «x y x z». В половине случаев за x должно следовать y, в другой половине — z. Что даст для этого примера алгоритм Шеннона?

16. Как можно использовать количество k-грамм в буквах или словах?

17. Некоторые коммерческие системы распознавания речи основаны на статистике триграмм.

Решения избранных задач

Решения к главе 1

1. Приведенная далее программа на C++ использует библиотечную функцию qsort для сортировки файла целых чисел:

```
int intcomp(int *x, int *y)
{  return *x - *y; }
int a[1000000];
int main(void)
{  int i, n=0;
  while (scanf("%d", &a[n]) != EOF)
    n++;
  qsort(a, n, sizeof(int), intcomp);
  for (i = 0; i < n; i++)
    printf("%d\n", a[i]);
  return 0;
}
```

Еще одна программа использует контейнерный класс set из стандартной библиотеки шаблонов для решения той же задачи.

```
int main(void)
{  set<int> S;
  int i;
  set<int>  iterator j;
  while (cin >> i)
    S insert(i);
  for (j = S begin(); j != S end(); ++j)
    cout << *j << "\n";
  return 0;
}
```

В решении задачи 3 обсуждается производительность обеих программ.

2. Приведенные ниже функции позволяют установить, проверить и сбросить значение бита.

```
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
```

```
#define N 10000000
int a[1+N/BITSPERWORD];

void set(int i) { a[i>>SHIFT] |= (1<<(i & MASK)); }
void clr(int i) { a[i>>SHIFT] &= ~(1<<(i & MASK)); }
int test(int i) {return a[i>>SHIFT] & (1<<(i & MASK)); }
```

3. Программа на языке C, приведенная ниже, реализует алгоритм сортировки, используя функции, определенные в решении задачи 2.

```
int main(void)
{ int i;
  for (i = 0; i < N; i++)
    clr(i);
  while (scanf("%d", &i) != EOF)
    set(i);
  for (i = 0; i < N; i++)
    if (test(i))
      printf("%d\n", i);
  return 0;
}
```

Я воспользовался программой из решения задачи 4, чтобы сформировать файл из миллиона различных целых чисел, каждое из которых не превышало десяти миллионов. В табл. 1 приведена стоимость сортировки такого файла с помощью системной утилиты sort, программ на C и C++ из решения задачи 1 и программы с битовым вектором.

Таблица 1. Время сортировки файла

	Системная сортировка	C++/STL	C/qsort	C/bitmap
Полное время, с	89	38	12,6	10,7
Время вычисления, с	79	28	2,4	0,5
Занимаемая память, Мбайт	0,8	70	4	1,25

В первой строке приведено полное время работы программы, а во второй строке из этого времени вычитается время ввода и вывода данных (10,2 с). Хотя программа на C++ использует в 50 раз больше памяти и времени на вычисления, чем специализированная программа на C, сам текст программы оказывается вдвое короче и его гораздо проще расширить для решения других задач.

4. Смотрите главу 12, в особенности задачу 8. В приведенном ниже коде предполагается, что функция randint(l, u) возвращает случайное целое число в диапазоне [l..u].

```
for i = [0, n)
  x[i] = i
for i = [0, k)
  swap(i, randint(i, n-1))
  print x[i]
```

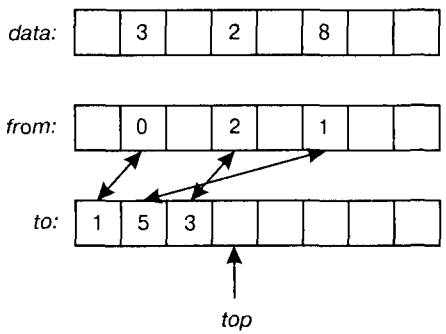
[1]Функция swap меняет местами два элемента массива x. Функция randint подробно обсуждается в разделе 12.1 главы 12.

5. Представление всех десяти миллионов чисел в битовом массиве потребует такого же количества битов, то есть 1,25 миллиона байтов. Если учесть, что телефон-

ные номера (в США) не начинаются с нуля и единицы, можно уменьшить этот объем до одного миллиона байтов. Можно использовать и двухпроходный алгоритм, который будет сортировать числа от 0 до 4 999 999 за первый проход, используя 5 000 000/8=652 000 слов памяти, а затем обработает числа от 5 000 000 до 9 999 999. k-проходный алгоритм сортирует n неповторяющихся натуральных чисел, не превосходящих n за время, пропорциональное k*n, и требует объема памяти, пропорционального n/k.

6. Если каждое число появляется не больше десяти раз, то количество его повторов можно сосчитать, имея четыре бита (половину байта). Используя решение задачи 5, можно отсортировать весь файл за один проход, используя 10 000 000/2 байт памяти или за k проходов, используя 10 000 000/(2×k) байт памяти.

9. Инициализация вектора data[0..n-1] может быть проведена с помощью сигнатуры, хранящейся в двух дополнительных n-элементных векторах from и to, и целого числа top. Если элемент data[i] проинициализирован, то from[i]<top и to[from[i]] = i. Поэтому from представляет собой простую сигнатуру, а to и top гарантируют, что случайное содержимое from до инициализации не будет воспринято как сигнатура. Свободные участки data сбрасываются так, как это показано на рисунке.



Переменная top изначально имеет нулевое значение. Первый доступ к элементу i осуществляется следующим кодом:

```
from[i] = top
to[top] = i
data[i] = 0
top++
```

Эта задача и ее решение взяты из упражнения 2.12 книги Ахо, Хопкрофта и Ульмана «Разработка и анализ компьютерных алгоритмов» (Aho, Hopcroft abd Ullman, Design and Analysis of Computer Algorithms, Addison Wesley, 1974). Она сочетает индексацию с хитрой схемой составления сигнатур. Программа может использоваться для матриц так же, как и для векторов.

10. Магазин размещал бумажные формы заказов в массив корзин 10×10, используя два последних знака телефонного номера заказчика в качестве индекса хэширования. При заказе форма помещалась в соответствующую телефонному номеру корзину. Когда покупатель прибывал за покупкой, продавец последовательно просматривал заказы в соответствующей корзине — это классическая схема *«открытого хэширования с разрешением коллизий с помощью последовательного поиска»*.

Последние две цифры телефонного номера распределены по покупателям практически случайным образом, поэтому дают отличную функцию хэширования, тогда как первые две цифры дали бы совершенно ужасную функцию — почему? В некоторых муниципалитетах используется аналогичная схема ведения записей в журналах.

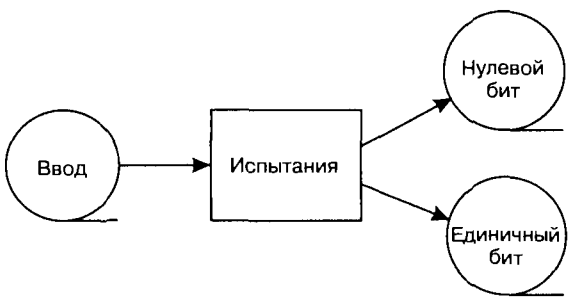
11. Компьютеры могли связываться с помощью УКВ-передатчика, но для распечатки чертежей в отделе тестирования потребовался бы плоттер большого формата, а такие плоттеры в то время стоили дорого. Поэтому чертежи выводились на заводе, затем фотографировались и пленка отсылалась в отдел тестирования почтовым голубем. На месте назначения ее распечатывали на фотоувеличителе. Голубь добирался до отдела за 45 минут: вдвое быстрее, чем машина, и обходился всего в несколько долларов в день. За 16 месяцев работы по проекту несколько сотен роликов пленки были пересланы с голубями и только два было утеряно (в той местности водились ястребы; никакая секретная информация с голубями не пересылалась). Сейчас плоттеры стоят дешево, поэтому и решение было бы другим: мы бы воспользовались передатчиком.

12. В соответствии с легендой, русские, разумеется, решили задачу с помощью карандаша. О том, что происходило на самом деле, вы можете прочитать по адресу: http://www.spacepen.com. Компания Fisher Space Pen была основана в 1948. Производимыми ею ручками пользовались и русские космонавты, и альпинисты в Гималаях, и исследователи подводного мира.

Решения к главе 2

Раздел 2.1

1. При рассмотрении этой программы двоичного поиска полезно помнить о 32 битах, представляющих каждое целое число. За первый проход мы считываем не более четырех миллиардов чисел и записываем все, начинающиеся с нуля, в один файл, а все остальные — в другой.



В одном из этих файлов содержится не более двух миллиардов чисел, и для этого файла мы и повторим процесс разбиения, но на этот раз по второму биту. Если в исходном файле содержалось n элементов, то на первом проходе будет считано n, на втором — n/2, на третьем — n/4 и так далее. Поэтому полное время работы программы пропорционально n. Если же попытаться отсортировать файл и затем просмотреть его, то на это потребуется время, пропорциональное $n \log n$. Эта задача

была предложена Эдом Рейнгольдом в качестве экзаменационной в университете штата Иллинойс.

2. См. раздел 2.3.

3. См. раздел 2.4.

Раздел 2.6

1. Чтобы найти все анаграммы некоторого слова, мы начинаем с вычисления его сигнатуры. Если предварительная обработка не предусматривается, придется считывать весь словарь, вычислять сигнатуру каждого из слов и затем сравнивать ее с сигнатурой исходного слова. Если разрешена предварительная обработка, можно решить задачу двоичным поиском по структуре, содержащей пары (сигнатура, слово), упорядоченные по сигнатуре. Муссер и Саини реализуют несколько программ поиска анаграмм в главах 12–15 книги «Учебное и справочное руководство по STL» (Musser, Saini, STL Tutorial and Reference Guide, Addison Wesley, 1996).

2. Двоичный поиск позволяет найти повторяющийся элемент рекурсивным поиском под-интервала, содержащего более половины чисел. В моем первом варианте решения не гарантировалось, что количество чисел на каждой итерации сокращается вдвое, поэтому в худшем случае время выполнения $\log_2 n$ проходов было пропорционально $n \log n$. Джиму Сейксу удалось сократить время выполнения до линейного, когда он учел тот факт, что можно отбрасывать повторяющиеся числа, если их слишком много. Если известно, что повторяющееся число должно быть в некотором диапазоне, включающем m чисел, то только m+1 число будет сохранено на рабочей ленте. Все лишние числа просто сбрасываются. Несмотря на то, что в его методе входные данные часто игнорируются, эта стратегия оказывается достаточно надежной: одно повторяющееся число будет найдено всегда.

3. Приведенный ниже код циклически сдвигает массив x[n] влево на rotdist.

```
for i = [0, gcd(rotdist, n))
  /* сдвиг i-х элементов блоков */
  t = x[i]
  j = i
  loop
    k = j + rotdist
    if k >= n
      k -= n
    if k == i
      break
    x[j] = x[k]
    j = k
  x[j]
```

Наибольший общий делитель rotdist и n равен количеству циклов перестановок (в терминах современной алгебры это число равно количеству классов смежности в группе перестановок, обусловленных циклическим сдвигом).

Следующая программа взята из раздела 18.1 книги Гриса «Наука программирования» (Gries, Science of Programming). В ней предполагается, что функция swap(a, b, m) меняет местами элементы x[a..a+m-1] и x[b..b+m-1].

```
if rotdist == 0 | | rotdist == n
return
i  = p = rotdist
j = n - p
while i != j
  /* инвариант
  x[0   p-i ] двигать не нужно
  x[p-i  p-1 ] = a (нужно поменять с b)
  x[p   p+j-1] = b (нужно поменять с a)
  x[p+j  n-1 ] двигать не нужно */
  if i > j
    swap(p-i, p, j)
    i -= j
  else
    swap(p-i , p+j-i, i)
    j -= i
swap(p-i , p, i)
```

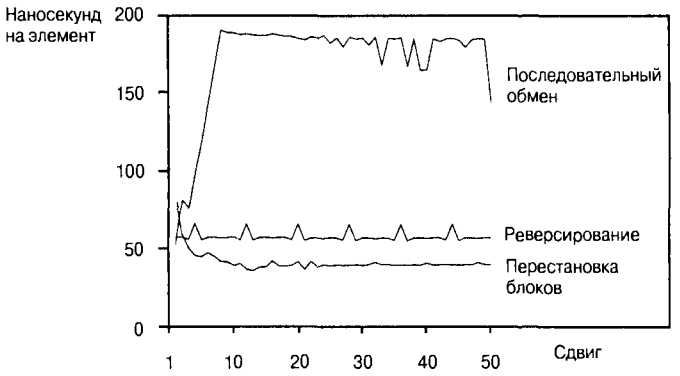
Понятие инварианта цикла описано в главе 4.

Этот код изоморфен приведенному далее (медленному, но верному) алгоритму Евклида вычисления наибольшего общего делителя i и j. Предполагается, что эти числа отличны от нуля.

```
int gcd(int i, int j)
  while i != j
    if i > j
      i -= j
    else
      j -= i
  return i
```

Грис и Миллс исследуют все три алгоритма циклического сдвига в статье «Обмен участков» (Gries, Mills, Swapping Sections, Cornell University Computer Science Technical Report 81–452).

4. Я запустил все три алгоритма на компьютере Pentium II 400 МГц, зафиксировав значение n = 1 000 000 и меняя величину сдвига rotdist от 1 до 50. На графике приведено среднее время выполнения программ.



Алгоритм реверсирования работает с постоянной скоростью 58 нс на элемент, но иногда его время работы подскакивает до 66 нс. Это происходит при сдвигах, равных 4, 12, 20 и так далее — последовательность чисел, дающих остаток 4 при делении на 8. Возможно, это связано с размером кэша, равным 32 байт. Алгоритм перестановки блоков является самым медленным для небольших сдвигов, вероятно, из-за затрат на вызов функции перестановки для одноэлементных блоков, но для сдвигов, больших 2, он оказывается самым быстрым, вероятно, из-за удобства кэширования. Алгоритм последовательного обмена является самым быстрым для малых сдвигов, но данные в этом алгоритме плохо кэшируются, поэтому время его работы возрастает почти до 200 нс при сдвиге, равном 8. Для больших сдвигов время работы этого алгоритма колеблется около 190 нс с отдельными спадами (при n=1000 время спадает до 105 нс, а затем снова возрастает до 190 нс). В середине 80-х аналогичный код использовался для работы со страницами памяти (сдвиг устанавливался равным обмену страницы).

6. Сигнатурой имени является его цифровой код (пример: «LESK*M*», «5375*6*»). Для поиска ложных совпадений следует сопоставить всем именам их цифровые сигнатуры, отсортировать их по сигнатурам (и по имени для одинаковых сигнатур), а затем последовательно считывать отсортированный файл и выводить сообщения об одинаковых сигнатурах, соответствующих различным именам. Чтобы определить имя по цифровой сигнатуре, мы используем структуру, в которой эти сигнатуры содержатся вместе с именами и другими данными. Можно, конечно, отсортировать эту структуру и затем искать в ней нужную сигнатуру с помощью двоичного поиска, но в реальной системе лучше всего будет воспользоваться хэшированием или базой данных.

7. Чтобы транспонировать матрицу, Виссоцки пронумеровал ее столбцы и строки, вызвал системную сортировку сначала по номерам столбцов, а затем по номерам строк, после чего удалил номера строк и столбцов с помощью третьей программы.

8. Ага!-озарение для этой задачи в том, что сумма элементов какого-то из k-элементных подмножеств не превышает t тогда и только тогда, когда сумма подмножества из k наименьших элементов не превышает этого числа. Это подмножество может быть найдено за время, пропорциональное $n \log n$, сортировкой исходного множества, или за время, пропорциональное n, с помощью алгоритма выбора (см. решение задачи 9 из главы 11). Когда Ульман предложил это задание в качестве упражнения своим студентам, те предложили всевозможные алгоритмы решения, среди которых были и алгоритмы с упомянутой скоростью работы, но кроме них были и такие, которые работали за $O(n \log k)$, $O(nk)$, $O(n^2)$ и $O(n^k)$. Можете ли вы придумать естественные алгоритмы решения этой задачи с таким временем работы?

10. Эдисон наполнил колбу водой, затем вылил эту воду в измерительную мензурку. Возможно, подсказка помогла вам вспомнить, что Архимед также пользовался водой для измерения объемов. В те времена было принято кричать не «Ага!», а «Эврика!», если вас посещало озарение.

Решения к главе 3

1. Каждая запись в таблице налогов содержит три значения: нижнюю границу диапазона, базовый тариф и тариф, по которому облагается разность между доходом

и нижней границы диапазона. Если таблицу дополнить последней (маркерной) записью с бесконечно большой нижней границей, программу последовательного поиска будет написать гораздо проще и работать она будет быстрее (см. раздел 9.2 главы 9). Можно также воспользоваться двоичным поиском. Эти методы применимы к любым кусочно-линейным функциям.

3. Изображенная ниже буква I

```
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX
   XXX
   XXX
   XXX
   XXX
   XXX
   XXX
XXXXXXXXX
XXXXXXXXX
XXXXXXXXX
```

может быть закодирована как

```
3 строк 9 X
6 строк 3 пробела 3 X 3 пробела
3 строк 9 X
```

или в более компактном виде:

```
3 9 X
6 3 п 3 X 3 п
3 9 X
```

4. Чтобы узнать количество дней между двумя конкретными датами, нужно определить номер дня в году для каждой из дат, вычесть раннюю из поздней, а затем добавить 365, помноженное на разность лет и прибавить количество високосных лет. Чтобы узнать день недели для конкретной даты, нужно вычислить количество дней между этой датой и любым воскресеньем, дата которого известна, а затем с помощью деления с остатком получить из этого количества день недели. Чтобы сформировать календарь на один месяц какого-либо года, нужно знать, сколько в этом месяце дней (особенно внимательным следует быть с февралем). Также нужно знать день недели, на который приходится первое число этого месяца. Дершовиц и Рейнгольд написали целую книгу о календарных вычислениях (Dershowitz and Reingold, Calendarical Calculations, Cambridge University Press, 1997).

5. Поскольку сравнения выполняются справа налево, может оказаться полезным хранить слова в обратном порядке (справа налево). Возможные представления последовательности суффиксов следующие: двумерный массив символов (обычно этот вариант слишком расточителен); одномерный массив символов, в котором суффиксы разделены специальным символом; массив символов с массивом указателей, по одному указателю на каждое слово.

6. Ахо, Керниган и Вейнбергер приводят программу порождения писем из 9 строк на странице 101 книги «Язык программирования Awk» (Aho, Kernighan and Weinberger, Awk Programming Language, Addison Wesley, 1988).

Решения к главе 4

1. Чтобы показать, что переполнения не происходит, мы добавим к инварианту условия 0<=l<=n и -l<=u<n. В этом случае мы можем ограничить l+u. Аналогичные условия могут быть использованы, чтобы показать, что обращения к элементам за границами массива никогда не происходит. Мы можем формально определить высказывание mustbe(l, u) как x[l-1]<t и x[u+1]>t, если мы добавим фиктивные граничные элементы x[-1] и x[n], как это сделано в разделе 9.3 главы 9.

2. См. раздел 9.3 главы 9.

5. В качестве введения в эту открытую и широко известную математическую проблему можно прочитать статью Б. Хайеса (B. Hayes, On the ups and downs of hailstone numbers Scientific American, 1, 1984). Более детально проблема обсуждается в статье Лагариаса (J. C. Lagarias, The 3x+1 problem and its generalizations, American Mathematical Monthly, 1, 1985). На момент выхода этой книги составленная Лагариасом библиография по указанной теме содержит 30 страниц, а на сайте http://www.research.att.com/~jcl/3x+1.html можно найти сотни ссылок.

6. Процесс сходится, потому что на каждом шаге количество бобов в банке уменьшается на единицу. На каждом этапе может быть убрано только четное количество белых бобов (0 или 2), поэтому их четность сохраняется. Последний оставшийся боб будет белым тогда и только тогда, когда в исходном состоянии количество белых бобов было нечетным.

7. Поскольку сегменты линий, формирующих лестницу, расположены по возрастанию ординаты, две линии, ограничивающие точку, могут быть найдены двоичным поиском. Лежащая в основе двоичного функция сравнения должна возвращать информацию о том, лежит ли точка выше данной прямой, ниже ее или непосредственно на ней. Как бы вы реализовали такую функцию?

8. См. раздел 9.3 главы 9.

Решения к главе 5

1. Когда я пишу большие программы, я использую длинные имена (из 10 или 20 символов) для глобальных переменных. В этой главе используются короткие имена переменных (типа x, n, t). В большинстве проектов допустимы имена не короче elem, nelems и target. Мне показалось, что короткие имена удобнее для тестовых программ и математических доказательств (вроде того, которое проводится в разделе 4.3). Аналогичные правила действуют в математике. Людям, незнакомым с предметом нужно говорить «квадрат гипотенузы прямоугольного треугольника равен сумме квадратов двух других сторон», но работающие над задачей люди обычно говорят просто «а квадрат плюс бэ квадрат равно цэ квадрат».

Я старался придерживаться стиля кодирования Кернигана и Ритчи, но помещал первую строку кода на одну строчку вместе с открывающей фигурной скобкой и вообще удалял лишние пустые строки для экономии бумаги (достаточно существенной для коротких функций, используемых в этой книге).

Двоичный поиск в разделе 5.1 главы 5 возвращает целое число, равное -1, если значение в массиве не найдено, либо указатель на это значение, если оно обнару-

жено. Стив Макконнелл предположил, что программа поиска должна возвращать два значения: булевскую переменную, сообщающую о наличии или отсутствии элемента в массиве, и индекс этого элемента, используемый только тогда, когда значение булевской переменной истинно.

```
boolean BinarySearch(DataType TargetValue, int *TargetIndex)
/* предусловие. Element[0] <= Element[1] <=
       <= Element[NumElements-1]
   постусловие.
     result == false =>
       значения TargetValue нет в массиве Element[0  NumElements  1]
     result == true =>
       Element[*TargetIndex] == TargetValue
*/
```

Листинг 18.3 на с. 402 книги Макконнелла (McConnell, Code Complete) занимает 1 страницу большого формата, а приведено в этом листинге не что иное, как текст программы сортировки вставкой на языке Паскаль. Код и комментарии к нему занимают 41 строку. Такой стиль подходит для больших программных проектов. В разделе 11.1 главы 11 моей книги тот же алгоритм записан в пяти строках кода.

Очень малая часть программ содержит средства проверки ошибок. В некоторых функциях данные считываются из файлов в массивы размера MAX и функция scanf легко может вызвать переполнение буфера. Массивы, которые должны были бы передаваться функции в качестве параметров, сделаны вместо этого глобальными переменными.

На протяжении всей книги я пользовался сокращениями, допустимыми для учебников и тестовых программ, но не для больших программных проектов. Как Керниган и Пайк отметили в разделе 1.1 своей книги (Kernighan, Pike, Practice of Programming), «*ясность часто достигается краткостью*». Даже несмотря на это, большая часть моих программ написана не в таком сжатом стиле, как, например, программа на C++ из раздела 14.3.

7. Для n=1000 поиск подряд идущих элементов выполняется в среднем за 351 нс, а поиск в случайном порядке выполняется в среднем за 418 нс (примерно на 20% медленнее). Для $n=10^6$ происходит переполнение кэша второго уровня, поэтому программа замедляется в 2,7 раза. Для оптимизированной программы двоичного поиска времена работы другие: при поиске элементов друг за другом время работы составляет 125 нс, а при поиске в случайном порядке — 266 нс, то есть работа замедляется более чем в два раза.

Решения к главе 6

4. Хотите, чтобы ваша система была надежной? Начните с увеличения надежности исходного проекта. Добавить надежность позже будет невозможно. Разрабатывайте структуры данных так, чтобы можно было восстановить информацию даже при частичном разрушении структуры. Проверяйте код вручную и тестируйте его всевозможными способами. Выполняйте свои программы только в надежных операционных системах и на надежной аппаратуре с встроенной коррекцией ошибок.

Учитывайте в своих планах возможную необходимость восстановления системы после сбоя (а сбои будут наверняка). Записывайте информацию обо всех сбоях, чтобы иметь возможность учиться на своих ошибках.

6. *«Сначала заставьте это работать, а уж потом пытайтесь заставить это работать быстро»*. Обычно это хороший совет. Однако Биллу Вульфу потребовалось всего несколько минут, чтобы убедить меня в том, что старая поговорка не так уж верна, как мне казалось. Билл привел в качестве примера программу подготовки документов, в которой подготовка книги занимала несколько часов. Рассуждал он примерно так: *«В этой программе, как в любой другой большой системе, есть ошибки. На данный момент их известно 10 штук, они все не слишком серьезны. В следующем месяце, возможно, будет обнаружено еще 10 таких же ошибок. Если бы вы могли выбирать между устранением 10 известных на данный момент ошибок или увеличением скорости работы программы в 10 раз, что бы вы выбрали?»*

Решения к главе 7

В этих решениях используются значения констант, которые на момент выхода книги могут раза в два отличаться от истинных значений, но не более того.

1. Река Пассаик Ривер не может течь со скоростью 200 миль в час, даже в той ее части, где она превращается в Великий Водопад в Нью-Джерси, высотой 80 футов. Думаю, что на самом деле инженер сказал репортеру, что река течет со скоростью 200 миль в день, что в 5 раз быстрее обычной ее скорости (40 миль в день или чуть меньше 2 миль в час).

2. На старом сменном диске помещается 100 Мбайт. Скорость передачи по линии ISDN составляет 112 Кбит/с или 50 Мбайт/час. Поэтому велосипедист с диском имеет в запасе около двух часов, то есть обладает 15-мильным радиусом превосходства. Если велосипедисту в рюкзак положить 100 DVD, то его «полоса пропускания» возрастет в 17 000 раз. Если ISDN заменить на ATM, то скорость передачи по этой линии возрастет до 155 Мбит/с, то есть в 1400 раз. Тогда преимущество велосипедиста возрастет в 12 раз и у него в запасе будет целый день. На следующий день после того, как был написан этот абзац, я зашел в офис к коллеге и увидел у него на столе стопку из 200 дисков по 5 Гбайт, предназначенных для однократной записи. В 1999 году терабайт готовых к записи носителей представлял собой ошеломляющее зрелище.

3. Емкость дискеты — 1,44 Мбайт. Я печатаю со скоростью примерно 50 слов (или 300 байт) в минуту. Поэтому на заполнение дискеты мне потребуется 4800 минут, или 80 часов. (Эта книга занимает всего полмегабайта, но мне потребовалось значительно больше трех дней, чтобы набрать ее).

4. Я ожидал получить в ответ фразы типа «инструкция, выполнявшаяся за 10 пс, теперь будет выполняться за 1 сотую секунды; диск, поворачивавшийся за 11 мс (5400 об/мин), повернется за 3 часа; поиск кластера замедлится от 20 мс до 6 часов, а две секунды, за которые я набираю свое имя, станут месяцем». Умный читатель написал мне: *«Сколько времени это займет? Столько же, сколько и раньше, потому что ход часов тоже замедлится»*.

5. Для скорости роста от 5 до 10% «правило 72» верно с точностью до 1%.

6. Поскольку 72/1,33 примерно равно 54, мы можем ожидать, что население удвоится к 2052 году (ООН ожидает, что скорость роста вскоре существенно снизится).

9. Если не учитывать замедление из-за использования очереди, 20 мс (скорость обращения), за которые выполняется одна операция с диском, дают в итоге 2 секунды на транзакцию и 1800 транзакций в час.

10. Смертность в некоторой области можно оценить, подсчитав количество сообщений о смертях в местных газетах и прикинув население области, к которой эти сообщения относятся. Другой подход использует закон Литтла и предполагаемое среднее время жизни. Если среднее время жизни 70 лет, то 1/70 или 1.4% населения каждый год умирает.

11. Доказательство закона Литтла, предложенное Петером Деннингом, состоит из двух частей. «*Пусть $l=A/T$ — скорость поступления объектов в систему, где A — количество поступивших за время наблюдения T объектов. Определим скорость ухода объектов как $X=C/T$, где C — количество уходящих за время T объектов. Пусть $n(t)$ — количество объектов в системе в момент времени t, принадлежащий интервалу $[0, T]$. Пусть W будет равно площади под графиком $n(t)$ в объекто-секундах. Эта величина будет отражать полное время ожидания всех объектов в системе за время наблюдения. Среднее время прохода одного объекта определяется как $R=W/C$ и измеряется в объекто-секундах на объект. Среднее количество объектов в системе равно $L=W/T$ и измеряется в объекто-секундах в секунду. Очевидно, что $L=RX$. Эта формула учитывает только скорость ухода объектов. При этом никакого требования на сбалансированность потока не накладывается. Если добавить и это требование, формула примет вид $L=l*R$. В этой форме она встречается в теории систем и очередей*».

12. Когда я прочитал, что монета в четверть доллара живет в среднем 30 лет, мне показалось, что это число слишком велико. Я не припоминал, чтобы мне часто приходилось видеть старые монеты. Поэтому я полез в карман и достал оттуда 12 монет в четверть доллара. Вот их возраст в годах:

```
3 4 5 7 9 9 12 17 17 19 20 34
```

Средний возраст составляет 13 лет, что прекрасно соответствует гипотезе о том, что среднее время жизни монеты в 2 раза больше (поскольку распределение монет по годам должно быть однородно). Если бы у меня в кармане оказался десяток монет моложе 5 лет, я бы, возможно, попробовал провести дальнейшие исследования. Но тут мне пришлось согласиться с тем, что газета оказалась права.

В той же статье говорилось, что в Нью-Джерси будет выпущено не менее 750 миллионов таких монет и что новый тип монет будет выпускаться каждые 10 недель. Это дает примерно 4 миллиарда четвертаков в год или 12 новых монет на каждого жителя США. Время жизни одной монеты в 30 лет означает, что у каждого человека будет по 360 монет. В карманы столько не положишь, но если учесть стопки мелочи, которые вы храните дома, в машинах и те, которые видите в кассах, торговых автоматах и банках, — это вполне допустимо.

Решения к главе 8

1. Девид Грис последовательно разрабатывает и проверяет алгоритм 4 в разделе «Замечание по поводу стандартной стратегии разработки инвариантов цикла и

самих циклов» книги «Наука программирования 2» (David Gries, Science of Computer Programming 2, pp. 207–214).

3. Алгоритм 1 делает примерно $n^3/6$ вызовов функции max, алгоритм 2 делает примерно $n^2/2$ вызовов, а алгоритм 4 — примерно $2n$. Алгоритм 2b использует линейно растущий объем памяти для кумулятивного массива, алгоритм 3 использует логарифмически растущий объем памяти на стеке. В других алгоритмах объем дополнительной памяти постоянен. Алгоритм 4 получает результат за один проход данных, что особенно полезно при работе с файлами.

5. Если cumarr объявлен как

```
float *cumarr.
```

тогда присваивание

```
cumarr = realarray+1
```

даст в итоге, что cumarr[-1] будет указывать на realarray[0].

9. Замените присваивание maxsofar=0 на maxsofar=-inf. Если отрицательная бесконечность вас беспокоит, то и присваивание maxsofar=x[0] тоже должно беспокоить. Почему?

10. Инициализируйте кумулятивный массив cum так, чтобы cum[i]=x[0]+...+x[i]. Сумма подпоследовательности x[l..u] равна нулю, если cum[l-1]=cum[u]. Подпоследовательность с ближайшей к нулю суммой может быть найдена поиском двух ближайших по значению элементов в массиве cum, что может быть выполнено за время $O(n \log n)$ с помощью сортировки. Это время выполнения отличается от оптимального не более чем на константу, потому что любой алгоритм решения такой задачи может быть использован для решения задачи «уникальности элементов» (Добкин и Липтон показали, что эта задача требует именно такого времени выполнения в худшем случае в модели вычислений с деревом решений).

11. Стоимость проезда между i и j может быть найдена как cum[j]-cum[i-1], где cum — массив кумулятивных сумм, как выше.

12. В этом решении используется еще один кумулятивный массив. Тело цикла

```
for i = [l, u]
  x[i] += v
```

можно заменить присваиваниями

```
cum[u]+=v
cum[l-1] -= v
```

Здесь число v добавляется к x[0..u], а затем вычитается из x[0..l-1]. После вычисления всех таких сумм можно получить массив x следующим образом:

```
for (i = n-1, i >= 0; i--)
  x[i] = x[i+1] + cum[i]
```

При этом время выполнения для худшего случая меняется с $O(n^2)$ на $O(n)$. Эта задача возникла при сборе статистики в задаче многих тел Эппеля, описанной в разделе 6.1 главы 6. Приведенное здесь решение ускорило работу функции расчета статистики с четырех часов до двадцати минут. Это увеличение скорости не имело бы значения, если бы программа работала год, но было достаточно важным, поскольку расчеты выполнялись за один день.

13. Подмножество массива m×n с максимальной суммой может быть найдено за время $O(m^2n)$ с помощью алгоритма 2 по измерению длины m и алгоритма 4 по измерению n. Задача размерностью n×n, таким образом, может быть решена за время $O(n^3)$. Этот результат был лучшим на протяжении двух десятилетий. Тамаки и Токуяма предложили несколько более быстрый алгоритм на симпозиуме по дискретным алгоритмам 1998 года (Tamaki, Tokuyama, Symposium on Discrete Algorithms, pp. 446–452). Этот алгоритм выполняется за время $O(n^3[(\log\log n)/(\log n)]^{1/2})$. Они также дают алгоритм, работающий за время $O(n^2 \log n)$, но он ищет подмножество, сумма которого не меньше половины максимальной, и описывают приложения этого метода к анализу баз данных. Оптимальный алгоритм должен работать за время, пропорциональное n^2.

Решения к главе 9

2. Описанные ниже переменные помогут реализовать вариант схемы ван Вайка. В переменной nodesleft будем хранить количество узлов размера NODESIZE, на которые указывает freenode. Когда очередной блок заканчивается, выделяется новый, в котором помещается NODEGROUP узлов.

```
#define NODESIZE 8
#define NODEGROUP 1000
int nodesleft = 0;
char *freenode;
```

Вызовы malloc заменяются на вызовы специальной функции:

```
void *pmalloc(int size)
{  void *p;
  if (size != NODESIZE)
    return malloc(size);
  if (nodesleft == 0) {
    freenode = malloc(NODEGROUP*NODESIZE);
    nodesleft = NODEGROUP;
  }
  nodesleft--;
  p = (void *) freenode;
  freenode += NODESIZE;
  return p;
}
```

Если запрашивается объем памяти, отличный от заданного, наша функция вызывает malloc. Когда блок иссякает, выделяется новый блок. Для тех же входных данных, профиль работы с которыми был приведен в разделе 9.1 главы 9, полное время работы уменьшилось с 2,67 до 1,55 с, а время, затраченное на вызовы malloc, снизилось с 1,41 до 0,31 с (то есть 19,7% нового времени выполнения).

Если программа будет не только выделять, но и освобождать узлы, новая переменная должна указывать на односвязный список свободных узлов. Когда узел освобождается, он помещается в начало списка. Когда список опустошается, выделяется новая группа узлов и эти узлы связываются в новый список.

4. Для массива уменьшающихся значений алгоритм выполняет примерно 2^n операций.

5. Если алгоритм двоичного поиска находит искомое значение, то оно действительно есть в массиве. Если же алгоритм двоичного поиска не находит искомое значение в неотсортированной таблице, то это ничего не значит. Это происходит, когда алгоритм обнаруживает пару соседних элементов, из значений которых следовало бы, что элемента t в массиве нет, если бы массив был отсортирован.

6. Проверить принадлежность символа к цифрам можно с помощью, к примеру, проверки типа:

```
if c >= '0' && c <= '9'
```

Чтобы проверить принадлежность символа к буквам и цифрам, нужно выполнить множество сложных проверок; если важна производительность, нужно в первую очередь делать те проверки, которые выполняются с наибольшей вероятностью. Обычно проще и быстрее использовать таблицу из 256 элементов:

```
#define isupper(c) (uppertable[c])
```

В большинстве систем в каждом элементе таблицы сохраняется несколько бит, которые потом извлекаются с помощью логического умножения:

```
#define isupper(c) (bitable[c] & UPPER)
#define isalnum(c) (bigtable[c] & (DIGIT|LOWER|UPPER))
```

Программисты на C и C++ могут изучить файл ctypes.h, чтобы узнать, как в их системе решается эта проблема.

7. Первый подход заключается в подсчете количества единичных битов в каждом входном блоке (8-битном символе или 32-битном целом числе) и последующем их суммировании. Чтобы найти количество единичных битов в 16-битном целом, нужно проверить значение каждого из битов, или перебрать только единичные биты (выражением типа b &= (b-1)), или воспользоваться таблицей из 2^{16}=65 536 элементов. Как повлияет размер кэша на выбор блока?

Второй подход заключается в подсчете количества блоков отдельно по всем возможным значениям блоков и последующем суммировании произведений количеств блоков с разными значениями на соответствующие эти значениям количества единичных битов.

8. Р. Г. Дроми (R. G. Dromey) написал приведенную ниже программу для нахождения максимального элемента массива x[0..n-1]. В ней элемент x[n] используется в качестве маркера.

```
i = 0
while i < n
  max = x[i]
  x[n] = max
  i++
  while x[i] < max
    i++
```

11. Замена вызовов функций на несколько таблиц из 72 элементов каждая уменьшило время выполнения программы на IBM 7090 с получаса до минуты. Поскольку для расчета лопастей винта вертолета требовалось около трехсот проходов про-

граммы, эти несколько сотен лишних байтов памяти позволили сократить неделю процессорного времени до нескольких часов.

12. По схеме Горнера значение полинома может быть вычислено следующим образом:

```
y = a[n]
for (i = n-1, i >= 0, i--)
  y = x*y + a[i]
```

В программе используется n умножений, и она обычно работает вдвое быстрее, чем предыдущая версия.

Решения к главе 10

1. Любая инструкция языка высокого уровня, обращавшаяся к одному из упакованных полей, преобразовывалась в большое количество инструкций процессора. Обращение к неупакованному полю требовало меньшего количества инструкций. Отказавшись от упаковки, Фельдман незначительно увеличил объем данных, но зато сильно сократил объем кода и время его работы.

2. Некоторые читатели предложили сохранять тройки чисел (x, y, номер точки) с упорядочением по y в массиве x. Тогда можно применять двоичный поиск для нахождения точки с координатами (x, y). Описанную в тексте структуру данных проще всего создать в том случае, если входные данные упорядочены по значениям координат x (а точки с одинаковыми координатами x упорядочены по y). Поиск по такой структуре можно проводить быстрее, если применить двоичный поиск по массиву row между значениями firstincol[i] и firstincol[i+1]-1. Обратите внимание, что значения y появляются в порядке возрастания и что двоичный поиск должен правильно обрабатывать ситуацию с пустым множеством.

4. В альманахах таблицы расстояний между городами приводятся в треугольном формате, что уменьшает занимаемую ими площадь вдвое. В математических таблицах иногда указываются только последние цифры значений функций, а первые значащие цифры приводятся только один раз для каждой группы значений. В программах телепередач место экономится благодаря тому, что указывается только время начала передач (альтернатива: приводить список всех передач, идущих по телевизору в течение определенного времени).

5. Брукс совместил два представления. Функция позволяла получить ответ приближенно, а недостающая точность достигалась прибавлением одной десятичной цифры, бравшейся из массива. Прочитав эту задачу и ее решение, двое рецензентов этого издания отметили, что им недавно приходилось решать задачи, дополняя аппроксимирующую функцию таблицей поправок.

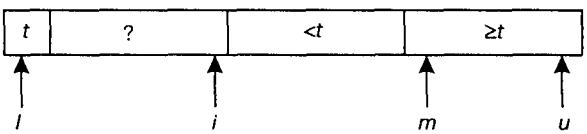
6. Исходный файл требовал 300 Кбайт на диске. Упаковка двух цифр в один байт уменьшила этот объем до 150 Кбайт, но увеличила время чтения файла. (Дело было в те дни, когда на односторонюю дискету размером 5,25 дюйма с двойной плотностью записи помещалось 184 Кбайт). Замена дорогостоящих операций деления с остатком на поиск по таблице уменьшила время считывания файла почти до первоначальной величины, но при этом было израсходовано около 200 байт ОЗУ.

Таким образом, 200 байт ОЗУ позволили сэкономить 150 Кбайт на диске. Некоторые читатели предлагали другой вариант кодирования: c = (a << 4)|b. Значения при этом могут быть восстановлены операциями a = c >> 4 и b = c & 0xF. Джон Линдерман отмечает, что логические операции и операции сдвига работают быстрее операций умножения и деления, и при этом данные, записанные в таком формате, можно прочитать с помощью стандартных утилит типа hex dump.

Решения к главе 11

1. Применять сортировку для поиска минимального или максимального из n вещественных чисел обычно неразумно. В решении задачи 9 показано, как можно найти медиану без сортировки, но в некоторых системах может оказаться проще сначала отсортировать данные. Сортировка удобна при поиске моды распределения, но хэширование может решить эту задачу быстрее. Очевидный алгоритм нахождения среднего значения работает за время, пропорциональное n, но предварительная сортировка таблицы помогает достичь большей численной точности (см. задачу 14.4b).

2. Боб Седжвик показал, что схему разбиения Ломуто можно изменить так, чтобы она работала справа налево, если использовать приведенный на рисунке инвариант.



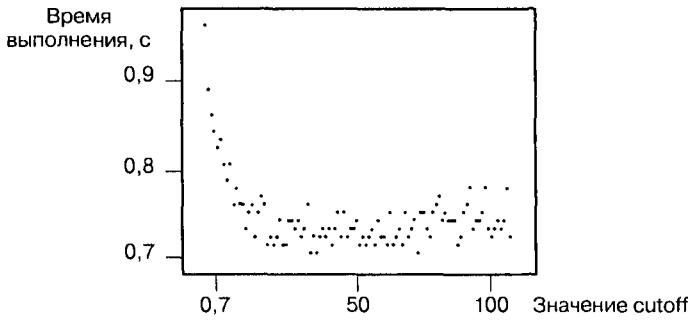
Код, осуществляющий разбиение, будет выглядеть следующим образом:

```
m = u + 1
for (i = u, i >= l; i++)
  if x[i] >= t
    swap(--m, i)
```

Когда цикл завершается, мы знаем, что x[m]=t, поэтому можно делать рекурсивный вызов с параметрами (l, m-1) и (m+1, u). При этом никаких дополнительных перестановок не требуется. Седжвик использовал элемент x[l] в качестве маркера, чтобы избавиться от одной из проверок во внутреннем цикле.

```
m = i = u+1
do
  while x[--i] < t
    ;
  swap(--m, i)
while i != l
```

3. Чтобы определить оптимальное значение cutoff, я прогнал программу со всеми возможными значениями cutoff от 1 до 100 (при n=1 000 000). На графике приведены результаты.



Я выбрал значение 50. Для всех значений cutoff от 30 до 70 результаты отличались не более чем на несколько процентов.

4. Обратитесь к литературе, упоминаемой в разделе 11.6 главы 11.

5. Программа Макилроя работала за время, пропорциональное объему сортируемых данных. Такая производительность является оптимальной для худшего случая. Предполагается, что каждая запись в массиве x[0..n-1] содержит целое значение длины length и указатель на массив bit[0..length-1].

```
void bsort(l, u, depth)
  if l >= u
    return
  for i = [l, u]
    if x[i] length < depth
      swap(i, l++)
  m = l
  for i = [l, u]
    if x[i] bit[depth] == 0
      swap(i, m++)
  bsort(l, m-1, depth+1)
  bsort(m, u, depth+1)
```

Первый раз функция вызывается как bsort (0, n-1, 1). Учтите, что в этой программе значения параметров цикла изменяются в теле цикла. Линейное время работы программы в значительной степени обусловлено тем, что операция swap переставляет указатели на битовые строки, а не сами эти строки.

6. Приведенный ниже код реализует сортировку выбором (selection sort):

```
void selsort()
  for i = [0, n-1)
    for j = [i, n)
      if x[j]<x[i]
        swap(i, j)
```

Еще одна программа реализует сортировку Шелла:

```
void shellsort()
  for (h = 1, h < n, h = 3*h+1)
    .
  loop
```

```
    if (h < 1)
      break
    for i = [h, n)
      for (j = i, j >= h, j -= h)
        if (x[j-h] < x[j])
          break
        swap(j-h, j)
```

9. Приведенный ниже алгоритм выбора принадлежит Хоару. Он представляет собой слегка измененную версию qsort4:

```
void select1(l, u, k)
    предусловие  l <= k <= u
    постусловие  x[l  k-1] <= x[k] <= x[k+1  u]
  if l >= u
    return
  swap(l, randint(l, u))
  t = x[l]; i = l, j = u+1
  loop
    do i++, while i <= u && x[i] < t
    do j--, while x[j] > t
    if i > j
      break
    temp = x[i], x[i] = x[j], x[j] = temp
  swap(l, j)
  if j < k
    select1(j+1, u, k)
  else if j > k
    select1(l, j-1, k)
```

Поскольку рекурсивный вызов стоит в конце функции, ее можно преобразовать в цикл while. В задаче 5.2.2-32 тома «Сортировка и поиск» Д. Кнут показывает, что в среднем такая программа находит медиану n элементов за $3,4n$ сравнений. Вычисление ведется в том же духе, что и для худшего случая в решении задачи 1 из раздела 2.1 главы 2.

14. В этой версии быстрой сортировки используются указатели на массивы. Поскольку она принимает только два параметра x и n, я считаю ее даже более простой, чем qsort1, если, конечно, читатель понимает, что запись x+j+1 означает массив, начинающийся с элемента x[j+1].

```
void qsort5(int x[], int n)
{  int i, j,
  if (n <= 1)
    return,
  for (i = 1, j = 0, i < n, i++)
    if (x[i] < x[0])
      swap(++j, i, x)
  swap(0, j, x),
  qsort5(x, j),
  qsort5(x+j+1, n-j-1),
}
```

Поскольку в этой программе используются указатели, ее можно реализовать на C и C++, но не на языке Java. Функции swap теперь приходится передавать имя массива (то есть указатель на него).

Решения к главе 12

1. Приведенные ниже функции возвращают большое (обычно 30-разрядное) случайное число и случайное число в некотором диапазоне:

```
int bigrand()
{  return RAND_MAX*rand() + rand(); }
int randint(int l, int u)
{  return l + bigrand() % (u-l+1); }
```

2. Чтобы выбрать m целых из диапазона 0..n-1, следует выбрать произвольное значение i из этого диапазона, а затем вернуть значения i, i+1, ... , i+m-1, при необходимости округляя их к нулю. В этом методе каждое число выбирается с вероятностью m/n, но некоторые наборы имеют существенно большую вероятность выбора, чем все остальные.

3. Если выбрано меньше, чем n/2 чисел, вероятность того, что некоторое число еще не выбрано, оказывается больше 1/2. Поэтому отсутствующее в выборке число будет выбрано в среднем менее чем за 2 хода. Логика такая же, как для монеты, которую нужно в среднем бросить дважды, чтобы выпал орел.

4. Можно рассматривать набор S как множество из n изначально пустых корзин. Каждый вызов randint позволяет выбрать корзину, в которую будет помещен шар. Если она уже занята, функция member вернет значение true. Задача о количестве шаров, гарантирующая отсутствие пустых корзин, называется у статистиков «задачей коллекционера» (сколько нужно собрать карточек, чтобы получить все n разных типов?); результат приблизительно равен $n \ln n$. Когда все шары попадают в разные корзины, программа делает m проверок. Задача о количестве шаров, при котором в какой-нибудь корзине их окажется два, называется «парадоксом дней рождений» (в любой группе из 23 и более человек двое почти наверняка родились в один и тот же день года). В какой-либо из корзин окажется два шара, если шаров будет $O(\sqrt{n})$.

7. Чтобы вывести значения в порядке возрастания, можно поместить оператор вывода print после рекурсивного вызова или выводить значение n+1-i, а не само i.

8. Чтобы выводить различные целые числа в случайном порядке, нужно выводить их сразу после того, как они порождаются. См. также решение 1.4. Чтобы вывести числа с повторениями в порядке возрастания, нужно убрать проверку наличия чисел в наборе. Чтобы выводить числа с повторениями в случайном порядке, нужно написать тривиальную программу

```
for i = [0, m)
  print bigrand() % n
```

9. Когда Боб Флойд изучил алгоритм, основанный на наборах, его не удовлетворил тот факт, что часть порождаемых чисел отбрасывается. Поэтому он написал другой алгоритм с наборами, реализацию которого на C++ мы и приводим здесь.

```
void genfloyd(int m, int n)
{  set<int> S,
  set<int>  iterator i,
  for (int j = n-m, j<n, j++) {
    int t = bigrand() % (j+1)
    if (S find(t) == S end())
      S insert(t)  // t нет в S
    else
      S insert(j), // t уже есть в S
  for (i = S begin(), i != S end(), ++i)
    cout << *i << "\n",
}
```

В решении задачи 1 из главы 13 реализован тот же алгоритм с альтернативным интерфейсом наборов. Алгоритм Флойда был впервые опубликован в колонке «Programming Pearls» журнала Communications of the ACM в 1986 году и был вынесен в отдельную главу книги «More Programming Pearls» в 1988 году. Там же вы можете прочитать простое доказательство его правильности.

10. Первую строку мы выбираем всегда, вторую — с вероятностью 1/2, третью — с вероятностью 1/3 и так далее. К концу процесса вероятность выбора всех строк окажется одинаковой (1/n, где n — полное число строк в файле).

```
i = 0
пока на входе есть строки
  с вероятностью 1 0/++i
    choice = текущая строка
print choice
```

11. Именно в этой постановке задача была предложена мной в качестве домашней контрольной работы по курсу разработки прикладных алгоритмов. Студенты, предложившие способ получить ответ за несколько минут компьютерного времени, получили нулевые баллы. Ответ *«я бы поговорил с преподавателем статистики»* получил половину максимального количества баллов, а лучшие ответы звучали так:

Числа 4..16 на игру не влияют, поэтому их можно отбросить. Карта выигрывает, если выбраны числа 1 и 2 (в произвольном порядке) до числа 3. Это происходит в том случае, если число 3 оказывается выбрано последним, что происходит в одном случае из трех. Поэтому вероятность выигрыша при случайном выборе составляет 1/3.

Не давайте условиям задачи ввести себя в заблуждение. Не обязательно использовать компьютер только потому, что он есть.

12. В разделе 5.9 (глава 5 данной книги) рассказывается о книге Кернигана и Пайка «Практика программирования». В разделе 6.8 их книги описан процесс тестирования программы с вероятностным поведением (аналогичная программа для той же задачи описана в разделе 15.3 главы 15).

Решения к главе 13

1. Алгоритм Флойда из решения задачи 9 из главы 12 может быть реализован с помощью класса IntSet следующим образом:

```
void genfloyd(int m, int maxval)
{  int *v = new int[m];
  IntSetSTL S(m, maxval);
  for (int j = maxval-m; j < maxval; j++) {
    int t = bigrand() % (j+1);
    int oldsize = S.size();
    S.insert(t);
    if (S.size() == oldsize) //t уже есть в S
      S.insert(j);
    }
    S.report(v);
    for (int i = 0; i < m; i++)
      cout << v[i] << "\n";
}
```

Когда значения m и maxval равны, элементы помещаются в набор в порядке возрастания, то есть мы получаем самый худший случай для двоичных деревьев поиска.

4. Приведенный ниже итеративный алгоритм вставки для связанных списков оказывается длиннее соответствующего рекурсивного алгоритма, потому что в нем отдельно рассматриваются ситуации вставки элемента сразу после первого в списке и во все прочие позиции.

```
void insert(t)
  if head->val == t
    return
  if head->val > t
    head = new node(t, head)
    n++
    return
  for (p = head; p->next->val < t; p = p-> next)
    ;
  if p->next->val == t
    return
  p->next = new node(t, p->next)
  n++
```

Более простая версия этого алгоритма использует указатель на указатель:

```
void insert(t)
  for (p = &head; (*p)->val < t; p = &((*p)->next))
    ;
  if (*p)->val == t
    return
  *p = new node(t, *p)
  n++
```

Она так же быстра, как и предыдущая. Этот же алгоритм с небольшими изменениями работает и для корзин, а в решении задачи 7 аналогичный подход используется для двоичных деревьев поиска.

5. Чтобы заменить несколько операций выделения одной, нужно иметь указатель на следующий доступный узел:

```
node *freenode;
```

Память под все узлы, которые мы собираемся выделить в будущем, выделяется при вызове конструктора класса:

```
freenode = new node[maxelms]
```

Затем по мере необходимости мы выделяем память под вставляемые элементы:

```
if (p == 0)
  p = freenode++
  p->val = t
  p->left = p->right = 0
  n++
else if
```

Аналогичный метод можно применить и к корзинам. В решении задачи 7 он применен к двоичным деревьям поиска.

6. Вставка узлов в порядке возрастания позволяет измерить скорость поиска в массивах и списках с небольшими затратами на собственно вставку. Такая последовательность элементов создаст наихудшую ситуацию для корзин и двоичных деревьев поиска.

7. Указатели, которые в предыдущей версии были нулевыми (null), будут теперь указывать на маркерный узел. Он инициализируется тем же конструктором:

```
root = sentinel = new node
```

Функция вставки сначала помещает вставляемое значение в маркерный узел, а затем использует указатель на указатели (как в решении задачи 4), спускаясь по дереву до тех пор, пока не будет найден элемент t. После этого для вставки нового узла используется метод из решения задачи 5.

```
void insert(t)
  sentinel->val = t
  p = &root
  while (*p)->val != t
    if t < (*p)->val
      p = &((*p)->left)
    else
      p = &((*p)->right)
  if *p == sentinel
    *p = freenode++
    (*p)->val = t
    (*p)->left = (*p)->right = sentinel
    n++
```

Переменная node объявляется и инициализируется следующим оператором:

```
node **p = &root.
```

9. Чтобы заменить деление на сдвиг, нужно проинициализировать переменные следующим образом:

```
goal = n/m
binshift = 1
for (i = 2, i < goal, i*=2)
```

```
  binshift++
nbins = 1+ (n >> binshift)
```

Функция вставки начинает проход с узла

```
p = &(bin[t>>binshift])
```

10. Для представления случайных наборов можно применять различные структуры. Поскольку мы имеем полную информацию о статистическом распределении данных по корзинам, можно воспользоваться идеями из раздела 13.2 и представлять данные в большинстве корзин с помощью небольшого массива (если корзина переполняется, лишние элементы помещаются в связанный список). Д. Кнут предложил для этой задачи в колонке «Programming Pearls» за май 1986 года «упорядоченную таблицу хэширования», в качестве иллюстрации к его системе документирования программ на языке Паскаль. Эта статья включена в качестве главы 5 в его книгу «Literate programming».

Решения к главе 14

1. Выполнение функции siftdown может быть ускорено вынесением операций с временной переменной из цикла. Функцию siftup можно ускорить вынесением кода из циклов и добавлением маркерного элемента в x[0] для устранения проверки if i==1.

2. Новая версия siftdown имеет лишь небольшие отличия от приведенной в тексте. Присваивание i=1 заменяется на i=l, а сравнения с n заменяются на сравнения с u. Время выполнения получившейся функции будет равно $O(\log u - \log l)$. Приведенный ниже код строит кучу за время $O(n)$:

```
for (i = n-1, i >= 1, i--)
  /* инвариант  maxheap(i+1, n) */
  siftdown(i, n)
  /* maxheap(i, n) */
```

Поскольку высказывание maxheap (l, n) верно для всех чисел l>n/2, границу n-1 в цикле for можно заменить на n/2.

3. Используя функции из решений задач 1 и 2, сортировку heapsort можно записать следующим образом:

```
for (i = n/2; i >= 1; i--)
  siftdown1(i, n)
for (i = n; i >= 2; i--)
  swap(1, i)
  siftdown(1, i-1)
```

Время выполнения остается равным $O(n \log n)$, но константа оказывается меньшей, чем для исходной версии сортировки с помощью кучи. В моей программе, сравнивающей различные сортировки, приводится несколько реализаций алгоритма heapsort. Программу можно скачать с сайта книги.

4. Кучи позволяют перейти от алгоритма $O(n)$ к $O(\log n)$ в перечисленных ниже задачах:

а) на каждом шаге итеративного построения кода Хаффмана выбираются два минимальных элемента в наборе, после чего они сливаются в новый узел.

Это реализуется двумя вызовами extractmin и одним insert. Если входные частоты представить в порядке возрастания, то код Хаффмана может работать с линейной скоростью. Подробности реализации оставляются читателю в качестве упражнения;

б) простой алгоритм суммирования вещественных чисел может оказаться недостаточно точным, если складывать очень большие числа с очень маленькими. Более совершенный алгоритм всегда должен складывать два наименьших элемента набора, и вообще он изоморфен упомянутому выше алгоритму нахождения кодов Хаффмана;

в) куча с миллионом элементов (наименьший находится на вершине кучи) представляет миллион наибольших на данный момент чисел;

г) куча может использоваться для слияния отсортированных файлов. На каждом шаге итеративного алгоритма наименьший элемент извлекается из кучи, а следующий за ним помещается в кучу. Очередной элемент, выводимый из n файлов, может быть найден за время $O(\log n)$.

5. Структура, аналогичная куче, может быть создана поверх последовательности корзин. Каждый узел кучи хранит информацию о самой свободной корзине среди его дочерних. Когда делается выбор, куда класть груз, поиск сначала идет влево, если есть возможность (самая свободная корзина слева имеет достаточно места для груза), либо вправо, если это необходимо. Эта операция выполняется за время, пропорциональное высоте кучи, то есть $O(\log n)$. После размещения груза значения элементов кучи должны быть реорганизованы обратным ходом.

6. В типичной реализации последовательного файла указатель в блоке i указывает на блок i+1. Маккрейт решил, что если добавить еще один указатель на узел 2*i, то к произвольному узлу можно будет обратиться за $O(\log n)$ попыток. Приведенная ниже рекурсивная функция выводит последовательность обращений.

```
void path(n)
    предусловие  n >= 0
    постусловие  выводится путь к узлу n
  if n == 0
    print "start at 0"
  else if even(n)
    path(n/2)
    print "double to ".n
  else
    path(n-1)
    print "increment to". n
```

Обратите внимание, что эта программа изоморфна программе вычисления x^n за $O(\log n)$ шагов, предлагаемой в задаче 9 из главы 4.

7. Модифицированный двоичный поиск начинается с элемента i = 1 и на каждой итерации присваивает переменной i значение 2*i либо 2*i+1. В элементе x[1] содержится медиана, в x[2] — первая квартиль, в x[3] — третья квартиль и так далее. С. Р. Махани (S. R. Mahaney) и Дж. И. Мунро (J. I. Munro) нашли алгоритм упорядочения n-элементного предварительного отсортированного массива в порядке «heapsearch» за время $O(n)$ и с применением постоянной дополнительной памяти.

Чтобы дойти до их метода, попробуйте скопировать отсортированный массив a размера 2^{k-1} в массив b, предназначенный для поиска: элементы, стоящие на четных позициях, в a помещаются во вторую половину массива, Элементы, стоящие на позициях $2 \bmod 4$, — во вторую четверть массива b, и так далее.

11. Стандартная библиотека шаблонов C++ STL поддерживает кучи и операции для них: make_heap, push_heap, pop_heap, sort_heap. Из них легко получить сортировку heapsort:

```
make_heap(a, a+n);
sort_heap(a, a+n);
```

В библиотеке STL имеется также адаптер priority_queue.

Решения к главе 15

1. Многие системы подготовки документов позволяют избавиться от команд форматирования и оставить только текстовую часть файла. Когда я работал с программой поиска повторяющихся строк из раздела 15.2, я обнаружил, что она была очень чувствительна к форматированию текста. За 36 секунд она обработала 4 460 056 символов Библии короля Иакова, и самая длинная повторявшаяся строка была 269 символов длиной. Когда я убрал номера строф, чтобы длинные строки могли выходить за их границы, длина повторявшейся строки возросла до 563 символов, причем она была найдена за приблизительно то же время.

3. Поскольку при каждой операции вставки выполняется много операций поиска, на выделение памяти будет уходить лишь небольшая часть времени. Добавление специальной функции выделения памяти ускорило работу программы лишь на 0.06 с, то есть на 10% от фазы считывания, но лишь на 2% от всего времени работы программы.

5. Можно добавить еще одну таблицу map к программе на C++, чтобы связать с каждым счетчиком последовательность слов. В программе на C мы могли бы упорядочить массив по полю счетчика, а затем перебрать его содержимое (поскольку некоторые слова будут повторяться очень много раз, этот массив должен оказаться намного меньше, чем входной файл). Для большинства документов можно использовать индексацию по ключу и хранить массив связанных списков для счетчиков в диапазоне 1..1000.

7. Учебники по теории алгоритмов предупреждают об опасности получения на входе файла из буквы «a», повторенной 1000 раз. Мне показалось проще испытать программу на файле, состоящем только из переводов строки. На обработку 5000 таких символов было затрачено 2,09 с; на 10 000 — 8,9 с, а на 20 000 — 37,9 с. Время работы программы росло несколько быстрее, чем квадрат размера файла, возможно, из-за $n \log_2 n$ сравнений, каждое из которых выполнялось за время, пропорциональное n. Более реалистичную ситуацию с плохими данными можно имитировать, сделав один файл из двух копий какого-либо большого файла.

8. Подмножество a[i..i+M] содержит M+1 строку. Поскольку массив отсортирован, мы можем быстро определить количество общих символов этих M+1 строк, вызвав comlen для первой и последней строки:

```
comlen(a[i], a[i+M])
```

На сайте книги приведена программа, реализующая этот алгоритм.

9. Считайте первую строку в массив c, найдите ее конец и допишите после нее завершающий символ \0. Затем считайте следующую строку и завершите ее. Проведите сортировку, как и в предыдущих задачах. При сканировании массива используйте исключающее ИЛИ для того, чтобы гарантировать, что ровно одна строка начинается до границы раздела.

14. Приведенная ниже функция хэширует последовательность из k слов, заканчивающихся символами \0.

```
unsigned int hash(char *)
  unsigned int h = 0
  int n
  for (n = k, n >0, p++)
    h = MULT * h + *p
    if (*p == 0)
      n-
  return h % NHASH
```

Программа на сайте книги вызывает эту функцию хэширования вместо двоичного поиска в алгоритме порождения марковского текста, что в среднем уменьшает время ее работы с $O(n \log n)$ до $O(n)$. Элементы хэш-таблицы представляют собой списки, размер которых (в 32-разрядных словах) совпадает с количеством слов в исходном тексте.

Алфавитный указатель

Т

У

Ф

Х

Ц

Ш

Э

Джон Бентли

БИБЛИОТЕКА ПРОГРАММИСТА

Жемчужины программирования

2-е издание

Эта книга предназначена для программистов. Хороший программист должен знать все, что написано до него, только тогда он будет писать хорошие программы. Главы этой книги посвящены наиболее привлекательному аспекту профессии программиста: жемчужинам программирования, рождающимся за пределами работы, в области фантазии и творчества. В них рассматриваются: постановка задач, теория алгоритмов, структуры данных, вопросы повышения эффективности кода, а также верификация и тестирование программ.

- правильная постановка задачи
- оптимизация кода по скорости выполнения
- минимизация используемой памяти
- реализация алгоритмов на Си
- деревья, списки, очереди, кучи
- сортировка
- разбор текста
- верификация программ

и многое другое...

ПИТЕР
WWW.PITER.COM

Addison-Wesley

Уровень пользователя:
опытный/эксперт
Серия:
библиотека программиста

ISBN 5-318-00715-5

9 785318 007156