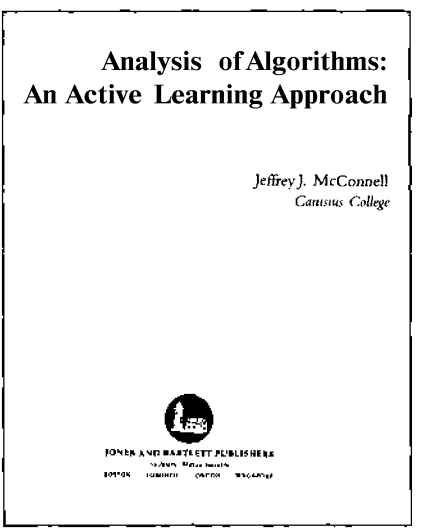
Дж. **Макконнелл**
Основы современных алгоритмов.
2-е дополненное издание
Москва:
Техносфера, 2004. - 368с. ISBN 5-94836-005-9

В учебном пособии обсуждаются алгоритмы решения наиболее широко распространенных классов задач, покрывающих практически всю область программирования: поиск и сортировка, численные алгоритмы и алгоритмы на графах. Особое внимание уделено алгоритмам параллельной обработки, редко освещаемым в литературе на русском языке.

В дополнении ко 2-му изданию на русском языке даны сведения по теории алгоритмов, оценкам трудоемкости и новейшим алгоритмам, не вошедшие в первоначальный вариант книги.

Изложение неформальное и чрезвычайно подробное, с большим количеством упражнений, позволяющих вести самоконтроль. Книга нужна всем, кому приходится самостоятельно писать программы — от программистов банковских систем до научных работников.



ISBN 5-94836-005-9
ISBN 0-7637-1634-0 (англ.)

МИР программирования

ДЖ. МАККОНЕЛЛ

Основы современных алгоритмов

2-е дополненное издание

Перевод с английского
под редакцией С.К.Ландо
Дополнение М.В.Ульянова

*Рекомендовано Ученым Советом Московской **государственной** Академии приборостроения и информатики в качестве **учебного** пособия по направлению подготовки специалистов "Информатика и вычислительная техника"*

ТЕХНОСФЕРА
Москва
2004

Содержание

Фреду и Барни

Предисловие

У этой книги две основные задачи -- объяснить, как алгоритмы влияют на эффективность программ, и научить анализировать разнообразные алгоритмы. Глядя на некоторые современные коммерческие программные продукты, понимаешь, что некоторые их создатели не обращают внимания ни на временную эффективность программ, ни на разумное использование памяти. Они полагают, что если программа занимает слишком много места, то пользователь купит дополнительную память, если она слишком долго работает, то он может купить более быстрый компьютер.

Однако скорость компьютеров не может увеличиваться бесконечно. Она ограничена скоростью перемещения электронов по проводам, скоростью распространения света по оптическим кабелям и скоростью коммутации каналов связи компьютеров, участвующих в вычислениях. Другие ограничения связаны не с производительностью компьютеров, а непосредственно со сложностью решаемой задачи. Есть задачи, для решения которых не хватит человеческой жизни, даже если при этом будут использованы самые быстрые из известных алгоритмов. А поскольку среди этих задач есть и важные, необходимы алгоритмы получения приблизительных ответов.

В начале 80-х годов архитектура компьютеров серьезно ограничивала их скорость и объем памяти. Зачастую общий размер программ и данных не превышал 64К. У современных персональных компьютеров эта величина выросла в 1000 раз. Нынешнее программное обеспечение гораздо сложнее, чем в 1980 году, и компьютеры стали заметно

лучше, но это не повод, чтобы игнорировать вопросы эффективности программ при их разработке. В спецификации некоторых проектов включены ограничения на время выполнения и использование памяти конечным продуктом, которые могут побудить программистов экономить память и увеличивать скорость выполнения. Небольшие размеры карманных компьютеров также накладывают ограничения на размеры и скорость выполнения программ.

Педагогические принципы

Что я слышу, я забываю.
Что я вижу, я запоминаю.
Что я делаю, я знаю.

Конфуций

Книгу можно изучать либо самостоятельно, либо в небольших группах. Для этого главы сделаны независимыми и понятными; такую главу стоит прочитать перед встречей группы. Каждой главе предпосланы указания по ее изучению. Во многих главах приведены дополнительные данные, что позволяет читателю выполнять алгоритмы вручную, чтобы лучше их понять. Результаты применения алгоритмов к этим дополнительным данным приведены в Приложении В. К каждому разделу имеются упражнения, от простых — на трассировку алгоритмов, до более сложных, требующих доказательств. Читатель должен научиться выполнять упражнения каждой главы. При чтении курса лекций на основе этой книги упражнения можно давать в качестве домашнего задания или использовать в классе как для индивидуальной работы студентов, так и для обсуждения в небольших группах. Помимо книги имеется руководство для преподавателя, в котором содержатся указания по работе с материалом при активном коллективном изучении, а также решения упражнений. В главы 2, 3, 5, 6, и 9 включены упражнения по программированию. Программные проекты позволяют читателю превратить алгоритмы этих глав в программы и оттестировать их, а затем сравнить результаты работы реальных программ с полученными посредством теоретического анализа.

Алгоритмы

Анализ алгоритмов не зависит от компьютера или используемого языка программирования, поэтому алгоритмы в книге записываются

в псевдокоде. Эти алгоритмы может прочесть любой, кому знакомо понятие условного выражения (IF или CASE/SWITCH), цикла (FOR или WHILE) и рекурсии.

Планирование курса

Ниже приведен график использования материала в односеместровом курсе:

Глава 1 — 2 недели	Глава 4 — 1 неделя	Глава 7 — 2 недели
Глава 2 — 1 неделя	Глава 5 — 1 неделя	Глава 8 — 1 неделя
Глава 3 — 2 недели	Глава 6 — 2 недели	Глава 9 — 2 недели

Освоение глав 2, 4 и 5 скорее всего не потребует полной недели; оставшееся время можно использовать для введения в курс, знакомства с активным коллективным обучением и для часового экзамена. При наличии у студентов соответствующей подготовки первая глава также может быть пройдена быстрее.

Благодарности

Я хотел бы поблагодарить всех тех, кто помогал мне при подготовке этой книги. Прежде всего, я хотел бы поблагодарить студентов моего курса «Автоматы и алгоритмы» (весна 1997 года, весна 1998 года, весна 1999 года, весна 2000 года и осень 2000 года) за комментарии к ее первым версиям.

Присланные из издательства Джонс и Бартлетт рецензии на книгу были очень полезными; в результате текст стал яснее и появилось несколько дополнений. Я хотел бы также поблагодарить Дугласа Кэмпбелла (Университет Брайэм Янг), Нэнси Киннерсли (Университет Канзаса) и Кирка Пруса (Университет Питтсбурга) за комментарии.

Я хотел бы поблагодарить сотрудников издательства Джонс и Бартлетт: редакторов Эми Роз и Майкла Штранца, а также технического редактора Тару МакКормик за работу над этой книгой. Особенно я благодарен Эми за память о нашей короткой беседе несколько лет назад и последующие усилия по воплощению ее предмета в жизнь. Я также хотел бы поблагодарить Нэнси Янг за редактирование рукописи, а Брук Олбрайт за вычитывание корректуры. Все оставшиеся в тексте ошибки исключительно на совести автора.

И наконец, я признателен Фреду Дансеро за поддержку и советы на многих этапах работы над книгой и Барни за ту замечательную возможность отвлечься, которую может дать только собака.

Глава 1.

Основы анализа алгоритмов

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- читать и разрабатывать алгоритмы;
- читать и разрабатывать рекурсивные алгоритмы;
- опознавать операции сравнения и арифметические операции;
- пользоваться элементарной алгеброй.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- описывать анализ алгоритма;
- объяснять принципы выбора подсчитываемых операций;
- объяснять, как проводить анализ в наилучшем, наихудшем и среднем случае;
- работать с логарифмами, вероятностями и суммированием;
- описывать функции $\theta(f)$, $\Omega(f)$,$O(f)$, скорость роста и порядок алгоритма;
- использовать дерево решений для определения нижней границы сложности;
- приводить простое рекуррентное соотношение к замкнутому виду.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все приведенные примеры и убедитесь, что Вы их поняли. Кроме того, прежде чем читать ответ на вопрос, следует попробовать ответить на него самостоятельно. Подсказка или ответ на вопрос следуют непосредственно после вопроса.

* * *

Одну и ту же задачу могут решать много алгоритмов. Эффективность работы каждого из них описывается разнообразными характеристиками. Прежде чем анализировать эффективность алгоритма, нужно доказать, что данный алгоритм правильно решает задачу. В противном случае вопрос об эффективности не имеет смысла. Если алгоритм решает поставленную задачу, то мы можем посмотреть, насколько это решение эффективно. Данная глава закладывает основу для анализа и сравнения достаточно сложных алгоритмов, с которыми мы познакомимся позднее.

При анализе алгоритма определяется количество «времени», необходимое для его выполнения. Это не реальное число секунд или других промежутков времени, а приблизительное число операций, выполняемых алгоритмом. Число операций и измеряет относительное время выполнения алгоритма. Таким образом, иногда мы будем называть «временем» вычислительную сложность алгоритма. Фактическое количество секунд, требуемое для выполнения алгоритма на компьютере, непригодно для анализа, поскольку нас интересует только относительная эффективность алгоритма, решающего конкретную задачу. Вы также увидите, что и вправду время, требуемое на решение задачи, не очень хороший способ измерять эффективность алгоритма, потому что алгоритм не становится лучше, если его перенести на более быстрый компьютер, или хуже, если его исполнять на более медленном.

На самом деле, фактическое количество операций алгоритма на тех или иных входных данных не представляет большого интереса и не очень много сообщает об алгоритме. Вместо этого нас будет интересовать зависимость числа операций конкретного алгоритма от размера входных данных. Мы можем сравнить два алгоритма по скорости роста числа операций. Именно скорость роста играет ключевую роль, поскольку при небольшом размере входных данных алгоритм A может требовать меньшего количества операций, чем алгоритм B, но при росте объема входных данных ситуация может поменяться на противоположную.

Два самых больших класса алгоритмов — это алгоритмы с повторением и рекурсивные алгоритмы. В основе алгоритмов с повторением лежат циклы и условные выражения; для анализ таких алгоритмов требуется оценить число операций, выполняемых внутри цикла, и число итераций цикла. Рекурсивные алгоритмы разбивают большую задачу на фрагменты и применяются к каждому фрагменту по отдельности. Такие алгоритмы называются иногда «разделяй и властвуй», и их использование может оказаться очень эффективным. В процессе решения большой задачи путем деления ее на меньшие создаются небольшие, простые и понятные алгоритмы. Анализ рекурсивного алгоритма требует подсчета количества операций, необходимых для разбиения задачи на части, выполнения алгоритма на каждой из частей и объединения отдельных результатов для решения задачи в целом. Объединяя эту информацию и информацию о числе частей и их размере, мы можем вывести рекуррентное соотношение для сложности алгоритма. Полученному рекуррентному соотношению можно придать замкнутый вид, затем сравнивать результат с другими выражениями.

Мы начнем эту главу с описания того, что же такое анализ и зачем он нужен. Затем мы очертим круг рассматриваемых операций и параметров, по которым будет производиться анализ. Поскольку для нашего анализа необходима математика, несколько следующих разделов будут посвящены важным математическим понятиям и свойствам, используемым при анализе итеративных и рекурсивных алгоритмов.

1.1. Что такое анализ?

Анализируя алгоритм, можно получить представление о том, сколько времени займет решение данной задачи при помощи данного алгоритма. Для каждого рассматриваемого алгоритма мы оценим, насколько быстро решается задача на массиве входных данных длины N. Например, мы можем оценить, сколько сравнений потребует алгоритм сортировки при упорядочении списка из N величин по возрастанию, или подсчитать, сколько арифметических операций нужно для умножения двух матриц размером N x N.

Одну и ту же задачу можно решить с помощью различных алгоритмов. Анализ алгоритмов дает нам инструмент для выбора алгоритма. Рассмотрим, например, два алгоритма, в которых выбирается наибольшая из четырех величин:

```
largest = a
if b > largest then
   largest = b
end if
return a
if c > largest then
   largest = c
end if
if d > largest then
   largest = d
endif
return largest
```

```
if a > b then
   if a > c then
      if a > d then
         return a
      else
         return d
      end if
   else
      if c > d then
         return c
      else
         return d
      end if
   end if
else
   if b > c then
      if b > d then
         return b
      else
         return d
      end if
   else
      if c > d then
         return c
      else
         return d
      end if
   end if
end if
```

При рассмотрении этих алгоритмов видно, что в каждом делается три сравнения. Первый алгоритм легче прочесть и понять, но с точки зрения выполнения на компьютере у них одинаковый уровень сложности. С точки зрения времени эти два алгоритма одинаковы, но первый требует больше памяти из-за временной переменной с именем largest. Это дополнительное место не играет роли, если сравниваются числа или символы, но при работе с другими типами данных оно может стать существенным. Многие современные языки программирования позволяют определить операторы сравнения для больших и сложных объектов или записей. В этих случаях размещение временной переменной может

потребовать много места. При анализе эффективности алгоритмов нас будет в первую очередь интересовать вопрос времени, но в тех случаях, когда память играет существенную роль, мы будем обсуждать и ее.

Различные характеристики алгоритмов предназначены для сравнения эффективности двух разных алгоритма, решающих одну задачу. Поэтому мы никогда не будем сравнивать между собой алгоритм сортировки и алгоритм умножения матриц, а будем сравнивать друг с другом два разных алгоритма сортировки.

Результат анализа алгоритмов — не формула для точного количества секунд или компьютерных циклов, которые потребует конкретный алгоритм. Такая информация бесполезна, так как в этом случае нужно указывать также тип компьютера, используется ли он одним пользователем или несколькими, какой у него процессор и тактовая частота, полный или редуцированный набор команд на чипе процессора и насколько хорошо компилятор оптимизирует выполняемый код. Эти условия влияют на скорость работы программы, реализующей алгоритм. Учет этих условий означал бы, что при переносе программы на более быстрый компьютер алгоритм становится лучше, так как он работает быстрее. Но это не так, и поэтому наш анализ не учитывает особенностей компьютера.

В случае небольшой или простой программы количество выполненных операций как функцию от N можно посчитать точно. Однако в большинстве случаев в этом нет нужды. В § 1.4 показано, что разница между алгоритмом, который делает $N + 5$ операций, и тем, который делает $N + 250$ операций, становится незаметной, как только N становится очень большим. Тем не менее, мы начнем анализ алгоритмов с подсчета точного количества операций.

Еще одна причина, по которой мы не будем пытаться подсчитать все операции, выполняемые алгоритмом, состоит в том, что даже самая аккуратная его настройка может привести лишь к незначительному улучшению производительности. Рассмотрим, например, алгоритм подсчета числа вхождений различных символов в файл. Алгоритм для решения такой задачи мог бы выглядеть примерно так:

```
for all 256 символов do
   обнулить счетчик
end for
while в файле еще остались символы do
   ввести очередной символ
   увеличить счетчик вхождений прочитанного символа на единицу
end while
```

Посмотрим на этот алгоритм. Он делает 256 проходов в цикле инициализации. Если во входном файле N символов, то втором цикле N проходов. Возникает вопрос «Что же считать?». В цикле for сначала инициализируется переменная цикла, затем в каждом проходе проверяется, что переменная не выходит за границы выполнения цикла, и переменная получает приращение. Это означает, что цикл инициализации делает 257 присваиваний (одно для переменной цикла и 256 для счетчика), 256 приращений переменной цикла, и 257 проверок того, что эта переменная находится в пределах границ цикла (одна дополнительная для остановки цикла). Во втором цикле нужно сделать $N + 1$ раз проверку условия (+1 для последней проверки, когда файл пуст), и N приращений счетчика. Всего операций:

Приращения	$N + 256$
Присваивания	257
Проверки условий	$N + 258$

Таким образом, при размере входного файла в 500 символов в алгоритме делается 1771 операция, из которых 770 приходится на инициализацию (43%). Теперь посмотрим, что происходит при увеличении величины N. Если файл содержит 50 000 символов, то алгоритм сделает 100 771 операцию, из которых только 770 связаны с инициализацией (что составляет менее 1% общего числа операций). Число операций инициализации не изменилось, но в процентном отношении при увеличении N их становится значительно меньше.

Теперь взглянем с другой стороны. Данные в компьютере организованы таким образом, что копирование больших блоков данных происходит с той же скоростью, что и присваивание. Мы могли бы сначала присвоить 16 счетчикам начальное значение, равное 0, а затем скопировать этот блок 15 раз, чтобы заполнить остальные счетчики. Это приведет к тому, что в циклах инициализации число проверок уменьшится до 33, присваиваний до 33 и приращений до 31. Количество операций инициализации уменьшается с 770 до 97, то есть на 87%. Если же мы сравним полученный выигрыш с числом операций по обработке файла в 50 000 символов, экономия составит менее 0.7% (вместо 100 771 операций мы затратим 100 098). Заметим, что экономия по времени могла бы быть даже больше, если бы мы сделали все эти инициализации без циклов, поскольку понадобилось бы только 31 простое присваивание, но этот дополнительный выигрыш дает лишь 0.07% экономии. Овчинка не стоит выделки.

Мы видим, что вес инициализации по отношению к времени выполнения алгоритма незначителен. В терминах анализа при увеличении объема входных данных стоимость инициализации становится пренебрежимо малой.

В самой ранней работе по анализу алгоритмов определена вычислимость алгоритма в машине Тьюринга. При анализе подсчитывается число переходов, необходимое для решения задачи. Анализ пространственных потребностей алгоритма подразумевает подсчет числа ячеек в ленте машины Тьюринга, необходимых для решения задачи. Такого рода анализ разумен, и он позволяет правильно определить относительную скорость двух алгоритмов, однако его практическое осуществление чрезвычайно трудно и занимает много времени. Сначала нужно строго описать процесс выполнения функций перехода в машине Тьюринга, выполняющей алгоритм, а затем подсчитать время выполнения — весьма утомительная процедура.

Другой, не менее осмысленный, способ анализа алгоритмов предполагает, что алгоритм записан на каком-либо языке высокого уровня вроде Pascal, C, C++, Java или достаточно общем псевдокоде. Особенности псевдокода не играют существенной роли, если он реализует основные структуры управления, общие для всех алгоритмов. Такие структуры, как циклы вида **for** или while, механизм ветвления вида **if**, case или switch, присутствуют в любом языке программирования, и любой такой язык подойдет для наших целей. Всякий раз нам предстоит рассматривать один конкретный алгоритм — в нем редко будет задействовано больше одной функции или фрагмента программы, и поэтому мощность упомянутых выше языков вообще не играет никакой роли. Вот мы и будем пользоваться псевдокодом.

В некоторых языках программирования значения булевских выражений вычисляются сокращенным образом. Это означает, что член B в выражении A and B вычисляется только, если выражение A истинно, поскольку в противном случае результат оказывается ложным независимо от значения B. Аналогично, член B в выражении A or B не будет вычисляться, если выражение A истинно. Как мы увидим, не важно, считать ли число сравнений при проверке сложного условия равным 1 или 2. Поэтому, освоив основы этой главы, мы перестанем обращать внимание на сокращенные вычисления булевских выражений.

1.1.1. Классы входных данных

Роль входных данных в анализе алгоритмов чрезвычайно велика, поскольку последовательность действий алгоритма определяется не в последнюю очередь входными данными. Например, для того, чтобы найти наибольший элемент в списке из N элементов, можно воспользоваться следующим алгоритмом:

```
largest = list [1]
for i = 2 to N do
   if (list [i] > largest) then
      largest = list[i]
   end if
end for
```

Ясно, что если список упорядочен в порядке убывания, то перед началом цикла будет сделано одно присваивание, а в теле цикла присваиваний не будет. Если список упорядочен по возрастанию, то всего будет сделано N присваиваний (одно перед началом цикла и $N - 1$ в цикле). При анализе мы должны рассмотреть различные возможные множества входных значений, поскольку, если мы ограничимся одним множеством, оно может оказаться тем самым, на котором решение самое быстрое (или самое медленное). В результате мы получим ложное представление об алгоритме. Вместо этого мы рассматриваем все типы входных множеств.

Мы попытаемся разбить различные входные множества на классы в зависимости от поведения алгоритма на каждом множестве. Такое разбиение позволяет уменьшить количество рассматриваемых возможностей. Например, число различных расстановок 10 различных чисел в списке есть 10!= 3 628 800. Применим к списку из 10 чисел алгоритм поиска наибольшего элемента. Имеется 362 880 входных множеств, у которых первое число является наибольшим; их все можно поместить в один класс. Если наибольшее по величине число стоит на втором месте, то алгоритм сделает ровно два присваивания. Множеств, в которых наибольшее по величине число стоит на втором месте, 362 880. Их можно отнести к другому классу. Видно, как будет меняться число присваиваний при изменении положения наибольшего числа от 1 до N. Таким образом, мы должны разбить все входные множества на N разных классов по числу сделанных присваиваний. Как видите, нет необходимости выписывать или описывать детально все множества, по-

метенные в каждый класс. Нужно знать лишь количество классов и объем работы на каждом множестве класса.

Число возможных наборов входных данных может стать очень большим при увеличении N. Например, 10 различных чисел можно расположить в списке 3 628 800 способами. Невозможно рассмотреть все эти: способы. Вместо этого мы разбиваем списки на классы в зависимости от того, что будет делать алгоритм. Для вышеуказанного алгоритма разбиение основывается на местоположении наибольшего значения. В результате получается 10 разных классов. Для другого алгоритма, например, алгоритма поиска наибольшего и наименьшего значения, наше разбиение могло бы основываться на том, где располагаются наибольшее и наименьшее значения. В таком разбиении 90 классов. Как только мы выделили классы, мы можем посмотреть на поведение алгоритма на одном множестве из каждого класса. Если классы выбраны правильно, то на всех множествах входных данных одного класса алгоритм производит одинаковое количество операции, а на множествах из другого класса это количество операций скорее всего будет другим.

1.1.2. Сложность по памяти

Мы будем обсуждать в основном сложность алгоритмов по времени, однако кое-что можно сказать и про то, сколько памяти нужно тому или иному алгоритму для выполнения работы. На ранних этапах развития компьютеров при ограниченных объемах компьютерной памяти (как внешней, так и внутренней) этот анализ носил принципиальный характер. Все алгоритмы разделяются на такие, которым достаточно ограниченной памяти, и те, которым нужно дополнительное пространство. Нередко программистам приходилось выбирать более медленный алгоритм просто потому, что он обходился имеющейся памятью и не требовал внешних устройств.

Спрос на компьютерную память был очень велик, поэтому изучался и вопрос, какие данные будут сохраняться, а также эффективные способы такого сохранения. Предположим, например, что мы записываем вещественное число в интервале от -10 до $+10$, имеющее один десятичный знак после запятой. При записи в виде вещественного числа большинство компьютеров потратит от 4 до 8 байтов памяти, однако если предварительно умножить число на 10, то мы получим целое число в интервале от -100 до $+100$, и для его хранения выделяется всего один байт. По сравнению с первым вариантом экономия составляет от 3 до 7 байтов. Программа, в которой сохраняется 1000 таких чисел, экономит

от 3000 до 7000 байтов. Если принять во внимание, что еще недавно — в начале 80-х годов прошлого века — у компьютеров была память объемом лишь 65 536 байтов, экономия получается существенной. Именно эта необходимость экономить память наряду с долголетием компьютерных программ привела к проблеме 2000-го года. Если Ваша программа использует много различных дат, то половину места для записи года можно сэкономить, сохраняя значение 99 вместо 1999. Да и авторы программ не предполагали в 80-х годах, что их продукция доживет до 2000-го года.

При взгляде на программное обеспечение, предлагаемое на рынке сегодня, ясно, что подобный анализ памяти проведен не был. Объем памяти, необходимый даже для простых программ, исчисляется мегабайтами. Разработчики программ, похоже, не ощущают потребности в экономии места, полагая, что если у пользователя недостаточно памяти, то он пойдет и купит 32 мегабайта (или более) недостающей для выполнения программы памяти или новый жесткий диск для ее хранения. В результате компьютеры приходят в негодность задолго до того, как они действительно устаревают.

Некоторую новую ноту внесло недавнее распространение карманных компьютеров (PDA -- personal digital assistant). У типичного такого устройства от 2 до 8 мегабайт на все про все — и на данные, и на программы. И поэтому разработка маленьких программ, обеспечивающих компактное хранение данных, становится критической.

1.1.3. Упражнения

1) Напишите на псевдокоде алгоритм, подсчитывающий количество прописных букв в текстовом файле. Сколько сравнений требуется этому алгоритму? Каково максимальное возможное значение числа операций увеличения счетчика? Минимальное такое число? (Выразите ответ через число N символов во входном файле.)

2) В файле записан некоторый набор чисел, однако мы не знаем, сколько их. Напишите на псевдокоде алгоритм для подсчета среднего значения чисел в файле. Какого типа операции делает Ваш алгоритм? Сколько операций каждого типа он делает?

3) Напишите алгоритм, не использующий сложных условий, который по трем введенным целым числам определяет, различны ли они все

между собой. Сколько сравнений в среднем делает Ваш алгоритм? Не забудьте исследовать все классы входных данных.

4) Напишите алгоритм, который получает на входе три целых числа, и находит наибольшее из них. Каковы возможные классы входных данных? На каком из них Ваш алгоритм делает наибольшее число сравнений? На каком меньше всего? (Если разницы между наилучшим и наихудшим классами нет, перепишите свой алгоритм с простыми сравнениями так, чтобы он не использовал временных переменных, и чтобы в наилучшем случае он работал быстрее, чем в наихудшем).

5) Напишите алгоритм для поиска второго по величине элемента в списке из N значений. Сколько сравнений делает Ваш алгоритм в наихудшем случае? (Позднее мы обсудим алгоритм, которому требуется около N сравнений.)

1.2. Что подсчитывать и что учитывать

Решение вопроса о том, что считать, состоит из двух шагов. На первом шаге выбирается значимая операция или группа операций, а на втором - - какие из этих операций содержатся в теле алгоритма, а какие составляют накладные расходы или уходят на регистрацию и учет данных.

В качестве значимых обычно выступают операции двух типов: сравнение и арифметические операции. Все операторы сравнения считаются эквивалентными, и их учитывают в алгоритмах поиска и сортировки. Важным элементом таких алгоритмов является сравнение двух величин для определения — при поиске — того, совпадает ли данная величина с искомой, а при сортировке — вышла ли она за пределы данного интервала. Операторы сравнения проверяют, равна или не равна одна величина другой, меньше она или больше, меньше или равна, больше или равна.

Мы разбиваем арифметические операции на две группы: аддитивные и мультипликативные. Аддитивные операторы (называемые для краткости *сложениями)* включают сложение, вычитание, увеличение и уменьшение счетчика. Мультипликативные операторы (или, короче, *умножения)* включают умножение, деление и взятие остатка *по* модулю. Разбиение на эти две группы связано с тем, что умножения рабо-

тают дольше, чем сложения. На практике некоторые алгоритмы считаются предпочтительнее других, если в них меньше умножений, даже если число сложений при этом пропорционально возрастает. За пределами нашей книги остались алгоритмы, использующие логарифмы и тригонометрические функции, которые образуют еще одну, даже более времяёмкую, чем умножения, группу операций (обычно компьютеры вычисляют их значения с помощью разложений в ряд).

Целочисленное умножение или деление на степень двойки образуют специальный случай. Эта операция сводится к сдвигу, а последний по скорости эквивалентен сложению. Однако случаев, где эта разница существенна, совсем немного, поскольку умножение и деление на 2 встречаются в первую очередь в алгоритмах типа «разделяй и властвуй», где значимую роль зачастую играют операторы сравнения.

Классы входных данных

При анализе алгоритма выбор входных данных может существенно повлиять на его выполнение. Скажем, некоторые алгоритмы сортировки могут работать очень быстро, если входной список уже отсортирован, тогда как другие алгоритмы покажут весьма скромный результат на таком списке. А вот на случайном списке результат может оказаться противоположным. Поэтому мы не будем ограничиваться анализом поведения алгоритмов на одном входном наборе данных. Практически мы будем искать такие данные, которые обеспечивают как самое быстрое, так и самое медленное выполнение алгоритма. Кроме того, мы будем оценивать и среднюю эффективность алгоритма на всех возможных наборах данных.

Наилучший случай

Как показывает название раздела, наилучшим случаем для алгоритма является такой набор данных, на котором алгоритм выполняется за минимальное время. Такой набор данных представляет собой комбинацию значений, на которой алгоритм выполняет меньше всего действий. Если мы исследуем алгоритм поиска, то набор данных является наилучшим, если искомое значение (обычно называемое целевым значением или ключом) записано в первой проверяемой алгоритмом ячейке. Такому алгоритму, вне зависимости от его сложности, потребуется одно сравнение. Заметим, что при поиске в списке, каким бы длинным он ни был, наилучший случай требует постоянного времени. Вообще, вре-

мя выполнения алгоритма в наилучшем случае очень часто оказывается маленьким или просто постоянным, поэтому мы будем редко проводить подобный анализ.

Наихудший случай

Анализ наихудшего случая чрезвычайно важен, поскольку он позволяет представить максимальное время работы алгоритма. При анализе наихудшего случая необходимо найти входные данные, на которых алгоритм будет выполнять больше всего работы. Для алгоритма поиска подобные входные данные — это список, в котором искомый ключ окажется последним из рассматриваемых или вообще отсутствует. В результате может потребоваться N сравнений. Анализ наихудшего случая дает верхние оценки для времени работы частей нашей программы в зависимости от выбранных алгоритмов.

Средний случай

Анализ среднего случая является самым сложным, поскольку он требует учета множества разнообразных деталей. В основе анализа лежит определение различных групп, на которые следует разбить возможные входные наборы данных. На втором шаге определяется вероятность, с которой входной набор данных принадлежит каждой группе. На третьем шаге подсчитывается время работы алгоритма на данных из каждой группы. Время работы алгоритма на всех входных данных одной группы должно быть одинаковым, в противном случае группу следует подразбить. Среднее время работы вычисляется по формуле

$$A(n) = \sum_{i=1}^{m} p_i t_i, \tag{1.1}$$

где через п обозначен размер входных данных, через m — число групп. через p_i — вероятность того, что входные данные принадлежат группе с номером г, а через t_i — время, необходимое алгоритму для обработки данных из группы с номером i.[1]

[1] Несоответствие левой и правой частей формулы (левая зависит от п, а правая нет), является кажущимся: и разбиение на группы, и значения параметров p_i и t, в правой части тоже зависят от n. — *Прим. перев.*

В некоторых случаях мы будем предполагать, что вероятности попадания входных данных в каждую из групп одинаковы. Другими словами, если групп пять, то вероятность попасть в первую группу такая же, как вероятность попасть во вторую, и т.д., то есть вероятность попасть в каждую группу равна 0.2. В этом случае среднее время работы можно либо оценить по предыдущей формуле, либо воспользоваться эквивалентной ей упрощенной формулой

$$A(n) = \frac{1}{m}\sum_{i=1}^{m} t_i, \tag{1.2}$$

справедливой при равной вероятности всех групп.

1.2.1. Упражнения

1) Напишите алгоритм подсчета среднего значения, или медианы, трех целых чисел. Входные данные для такого алгоритма распадаются на шесть групп; опишите их. Какой случай для алгоритма является наилучшим? Наихудшим? Средним? (Если наилучший и наихудший случаи совпадают, то перепишите Ваш алгоритм с простыми условиями, не пользуясь вре́менными переменными, так, чтобы наилучший случай был лучше наихудшего.)

2) Напишите алгоритм, проверяющий, верно ли, что данные четыре целых числа попарно различны. Деление на группы возможных наборов входных данных зависит от структуры Вашего алгоритма либо от структуры задачи. Какой из классов обеспечивает наилучший случай для Вашего алгоритма? Наихудший? Средний? (Если наилучший и наихудший случаи совпадают, то перепишите Ваш алгоритм с простыми условиями, не пользуясь временными переменными, так, чтобы наилучший случай был лучше наихудшего.)

3) Напишите алгоритм, который по данному списку чисел и среднему значению этих чисел определяет, превышает ли число элементов списка, больших среднего значения, число элементов, меньших этого значения, или наоборот. Опишите группы, на которые распадаются возможные наборы входных данных. Какой случай для алгоритма является наилучшим? Наихудшим? Средним? (Если наилучший и наихудший случаи совпадают, то перепишите Ваш

алгоритм так, чтобы он останавливался, как только ответ на поставленный вопрос становится известным, делая наилучший случай лучше наихудшего.)[2]

1.3. Необходимые математические сведения

На протяжении всей книги мы будем пользоваться немногими математическими понятиями. К их числу принадлежат округление числа влево и вправо. Округлением влево (или целой частью) числа X мы назовем наибольшее целое число, не превосходящее X (обозначается $\lfloor X \rfloor$). Так, $\lfloor 2.5 \rfloor = 2$, а $\lfloor -7.3 \rfloor = -8$. Округлением вправо числа X называется наименьшее целое число, которое не меньше, чем X (обозначается через $\lceil X \rceil$). Так, $\lceil 2.5 \rceil = 3$, а $\lceil -7.3 \rceil = -7$. Чаще всего мы будем применять эти операции к положительным числам.

Округление влево и вправо будет применяться, когда мы хотим подсчитать, сколько раз было выполнено то или иное действие, причем результат представляет собой частное некоторых величин. Например, при попарном сравнении N элементов, когда первый элемент сравнивается со вторым, третий с четвертым и т.д., число сравнений будет равно $\lfloor N/2 \rfloor$. Если $N = 10$, то попарных сравнений будет 5, и $\lfloor 10/2 \rfloor = \lfloor 5 \rfloor = 5$. А если $N = 11$, то число сравнений не изменится и останется равным пяти, и $\lfloor 11/2 \rfloor = \lfloor 5.5 \rfloor = 5$.

Факториал натурального числа N обозначается через $N!$ и представляет собой произведение всех натуральных чисел от 1 до N. Например, $3! = 3 \cdot 2 \cdot 1 = 6$, а $6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 720$. Видно, что факториал становится большим очень быстро. Мы внимательней изучим этот вопрос в § 1.4.

1.3.1. Логарифмы

Логарифмам предстоит играть существенную роль в нашем анализе, поэтому нам следует обсудить их свойства. Логарифмом числа x по основанию y называется такая степень, в которую нужно возвести y, чтобы получить x. Так, $\log_{10} 45$ приблизительно равен 1.653, поскольку $10^{1.653} \approx 45$. Основанием логарифма может служить любое число, однако в нашем анализе чаще всего будут встречаться логарифмы по основаниям 10 и 2.

[2] Строго говоря, решение этой задачи зависит от того, знаем ли мы заранее длину входных данных или она выясняется в процессе работы алгоритма. — *Прим. перев.*

Логарифм — строго возрастающая функция. Это означает, что если $X > У$, то $\log_B X > \log_B У$ для любого основания B.[3] Логарифм — взаимно однозначная функция. Это означает, что если $\log_B X = \log_B У$, то $X = У$. Нужно знать также следующие важные свойства логарифма, справедливые при положительных значениях входящих в них переменных:

$$\log_B 1 = 0; \tag{1.3}$$

$$\log_B B = 1; \tag{1.4}$$

$$\log_B(XY) = \log_B X + \log_B У; \tag{1.5}$$

$$\log_B X^Y = Y \log_B X; \tag{1.6}$$

$$\log_A X = \frac{\log_B X}{\log_B A}. \tag{1.7}$$

С помощью этих свойств функцию можно упрощать. Свойство (1.7) позволяет заменять основание логарифма. Большинство калькуляторов позволяют вычислять логарифмы по основанию 10 и натуральные логарифмы, а что если Вам нужно вычислить $\log_{42} 75$? С помощью равенства (1.7) Вы легко получаете ответ: 1.155.

1.3.2. Бинарные деревья

Бинарное дерево представляет собой структуру, в которой каждый узел (или вершина) имеет не более двух узлов-потомков и в точности одного родителя. Самый верхний узел дерева является единственным узлом без родителей; он называется корневым узлом. Бинарное дерево с N узлами имеет не меньше $\lfloor \log_2 N + 1 \rfloor$ уровней (при максимально плотной упаковке узлов). Например, у полного бинарного дерева с 15 узлами один корень, два узла на втором уровне, четыре узла на третьем уровне и восемь узлов на четвертом уровне; наше равенство также дает $\lfloor \log_2 15 \rfloor + 1 = \lfloor 3.9 \rfloor + 1 = 4$ уровня. Заметим, что добавление еще одного узла к дереву приведет к необходимости появления нового уровня, и их число станет равным $\lfloor \log_2 16 \rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$. В самом большом бинарном дереве с N узлами N уровней: у каждого узла этого дерева в точности один потомок (и само дерево представляет собой просто список).

[3] Разумеется, если $B > 1$. — *Прим. перев.*

Если уровни дерева занумеровать, считая что корень лежит на уровне 1, то на уровне с номером K лежит 2^{K-1} узел. У полного бинарного дерева с J уровнями (занумерованными от 1 до J) все листья лежат на уровне с номером «7, и у каждого узла на уровнях с первого по J — 1 в точности два непосредственных потомка. В полном бинарном дереве с J уровнями 2^J — 1 узел. Эта информация не раз нам понадобится в дальнейшем. Советуем для лучшего понимания этих формул порисовать бинарные деревья и сравнить свои результаты с приведенными выше формулами.

1.3.3. Вероятности

Мы собираемся анализировать алгоритмы в зависимости от входных данных, а для этого нам необходимо оценивать, насколько часто встречаются те или иные наборы входных данных. Тем самым, нам придется работать с вероятностями того, что входные данные удовлетворяют тем или иным условиям. Вероятность того или иного события представляет собой число в интервале между нулем и единицей, причем вероятность 0 означает, что событие не произойдет никогда, а вероятность 1 - - что оно произойдет наверняка. Если нам известно, что число различных возможных значений входа в точности равно 10, то мы можем с уверенностью сказать, что вероятность каждого такого входа заключена между 0 и 1, и что сумма всех этих вероятностей равна 1, поскольку один из них наверняка должен быть реализован. Если возможности реализации каждого из входов одинаковы, то вероятность каждого из них равна 0.1 (один из 10, или 1/10).

По большей части, наш анализ будет заключаться в описании всех возможностей, а затем мы будем предполагать, что все они равновероятны. Если общее число возможностей равно N, то вероятность реализации каждой из них равна $1/N$.

Формулы суммирования

При анализе алгоритмов нам придется складывать величины из некоторых наборов величин. Пусть, скажем, у нас есть алгоритм с циклом. Если переменная цикла принимает значение 5, то цикл выполняется 5 раз, а если ее значение равно 20, то двадцать. Вообще, если значение переменной цикла равно M, то цикл выполняется M раз. В целом, если переменная цикла пробегает все значения от 1 до N, то суммарное число выполнений цикла равно сумме всех натуральных чисел

от 1 до N. Мы записываем эту сумму в виде $\sum_{i=1}^{N} г$. В нижней части знака суммы стоит начальное значение переменной суммирования, а в его верхней части — ее конечное значение. Понятно, как такое обозначение связано с интересующими нас суммами.

Если какое-нибудь значение записано в виде подобной суммы, то его зачастую стоит упростить, чтобы результат можно было сравнивать с другими подобными выражениями. Не так уж просто сообразить, что больше из двух чисел $\sum_{i=11}^{N}(i^2 - г)$ и $\sum_{i=0}^{N}(i^2 - 20i)$. Поэтому для упрощения сумм мы будем пользоваться нижеследующими формулами, в которых C — не зависящая от г постоянная.

$$\sum_{i=1}^{N} Ci = C\sum_{i=1}^{N} г \tag{1.8}$$

$$\sum_{i=L}^{N} i = \sum_{i=0}^{N-L} (i + L) \tag{1.9}$$

$$\sum_{i=L}^{N} i = \sum_{i=0}^{N} i - \sum_{i=0}^{L-1} i \tag{1.10}$$

$$\sum_{i=1}^{N}(A + B) = \sum_{i=l}^{N} A + \sum_{i=1}^{N} B \tag{1.11}$$

$$\sum_{i=0}^{N}(N - i) = \sum_{i=0}^{N} i \tag{1.12}$$

Равенство (1.12) означает просто, что сумма чисел от 0 до N равна сумме чисел от N до 0. В некоторых случаях применение этого равенства упрощает вычисления.

$$\sum_{i=1}^{N} 1 = N \tag{1.13}$$

$$\sum_{i=1}^{N} C = CN \tag{1.14}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \tag{1.15}$$

Равенство (1.15) легко запомнить, если разбить числа от 1 до N на пары. Складывая 1 *с* N, *2 с* $N-1$, и т.д., мы получим набор чисел, каждое из которых равно $N+1$. Сколько всего будет таких чисел? Разумеется, их число равно половине числа разбиваемых на пары элементов, т.е. *N/2.* Поэтому сумма всех N чисел равна

$$\frac{N}{2}(N+1) = \frac{N(N+1)}{2}. \tag{1.16}$$

$$\sum_{i=1}^{N} i^2 \quad N\frac{(N+1)(2N+1)}{6} = \frac{2N^3+3N^2+N}{6} \tag{1.17}$$

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1 \tag{1.18}$$

Равенство (1.17) легко запомнить через двоичные числа. Сумма степеней двойки от нулевой до десятой равна двоичному числу 11111111111. Прибавив к этому числу 1, мы получим 100000000000, т.е. 2^{11}. Но этот результат на 1 больше суммы степеней двойки от нулевой до десятой, поэтому сама сумма равна $2^{11}-1$. Если теперь вместо 10 подставить N, то мы придем к равенству (1.17).

$$\sum_{i=1}^{N} A^i = \frac{A^{N+1}-1}{A-1} \quad \text{для любого числа } A \tag{1.19}$$

$$\sum_{i=1}^{N} i2^i = (N-1)2^{N+1} + 2 \tag{1.20}$$

$$\sum_{i=1}^{N} \frac{1}{i} \approx \ln N \tag{1.21}$$

$$\sum_{i=1}^{N} \log_2 i \approx N \log_2 N - 1.5 \tag{1.22}$$

При упрощении сумм можно сначала разбивать их на более простые суммы с помощью равенств (1.8)–(1.12), а затем заменять суммы с помощью остальных тождеств.

1.3.4. Упражнения

1) Типичный калькулятор умеет вычислять натуральные логарифмы (по основанию e) и логарифмы по основанию 10. Как с помощью этих операций вычислить $\log_{27} 59$?

2) Допустим, у нас есть честная кость с пятью гранями, на которых написаны числа от 1 до 5. Какова вероятность выпадения каждого из чисел $1, \ldots, 5$ при бросании кости? В каком диапазоне могут меняться суммы значений при бросании двух таких костей? Какова вероятность появления каждого из чисел в этом диапазоне?

3) Допустим, у нас есть честная 8-гранная кость, на гранях которой написаны числа $1, 2, 3, 3, 4, 5, 5, 5$. Какова вероятность выбросить каждое из чисел от 1 до 5? В каком диапазоне могут меняться суммы значений при бросании двух таких костей? Какова вероятность появления каждого из чисел в этом диапазоне?

4) На гранях четырех кубиков написаны такие числа:

$$d_1 : 1, 2, 3, 9, 10, 11;$$
$$d_2 : 0, 1, 7, 8, 8, 9;$$
$$d_3 : 5, 5, 6, 6, 7, 7;$$
$$d_4 : 3, 4, 4, 5, 11, 12.$$

Вычислите для каждой пары кубиков вероятность того, что на первом кубике выпадет большее значение, чем на втором, и наоборот. Результаты удобно представлять в виде 4 x 4-**матрицы**, в которой строчка соответствует первому кубику, а столбец — второму. (Мы предполагаем, что бросаются разные кубики, поэтому диагональ матрицы следует оставить пустой.) У этих кубиков есть интересные свойства — обнаружите ли Вы их?

5) На столе лежат пять монет. Вы переворачиваете случайную монету. Определите для каждого из четырех приведенных ниже

случаев вероятность того, что после переворота решка будет на большем числе монет.

а. Два орла и три решки
б. Три орла и две решки
в. Четыре орла и одна решка
г. Один орел и четыре решки

6) На столе лежат пять монет. Вы переворачиваете один раз каждую монету. Определите для каждого из четырех приведенных ниже случаев вероятность того, что после переворота решка будет на большем числе монет.

а. Два орла и три решки
б. Три орла и две решки
в. Четыре орла и одна решка
г. Один орел и четыре решки

7) Найдите эквивалентное представление без помощи сумм для следующих выражений:

а) $\sum_{i=1}^{N}(2i+7)$; б) $\sum_{i=1}^{N}(i^2-2i)$; в) $\sum_{i=7}^{N}$ •

г) $\sum_{i=5}^{N}(2i^2+1)$; д) $\sum_{i=1}^{N}6^i$; е) $\sum_{i=7}^{N}4^i$.

1.4. Скорости роста

Точное знание количества операций, выполненных алгоритмом, не играет существенной роли в анализе алгоритмов. Куда более важным оказывается скорость роста этого числа при возрастании объема входных данных. Она называется скоростью роста алгоритма. Небольшие объемы данных не столь интересны, как то, что происходит при возрастании этих объемов.

Нас интересует только общий характер поведения алгоритмов, а не подробности этого поведения. Если внимательно посмотреть на рис. 1.1, то можно отметить следующие особенности поведения графиков функций. Функция, содержащая x^2, сначала растет медленно, однако при росте x ее скорость роста также возрастает. Скорость роста функций, содержащих ж, постоянна на всем интервале значений переменной. Функция $2\log x$ вообще не растет, однако это обман зрения. На

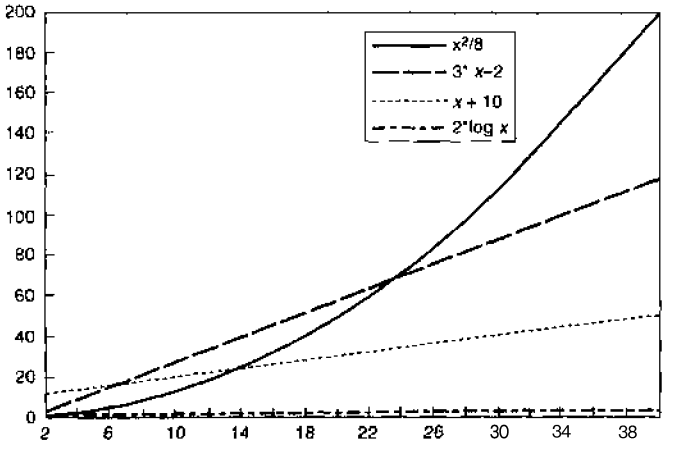


Рис. 1.1. Графики четырех функций

самом деле она растет, только очень медленно. Относительная высота значений функций также зависит от того, большие или маленькие значения переменной мы рассматриваем. Сравним значения функций при $x = 2$. Функцией с наименьшим значением в этой точке является $x^2/8$, а с наибольшим — функция $x + 10$. Однако с возрастанием x функция $x^2/8$ становится и остается впоследствии наибольшей.

Подводя итоги, при анализе алгоритмов нас будет интересовать скорее класс скорости роста, к которому относится алгоритм, нежели точное количество выполняемых им операций каждого типа. Относительный «размер» функции будет интересовать нас лишь при больших значениях переменной x.

Некоторые часто встречающиеся классы функций приведены в таблице на рис. 1.2. В этой таблице мы приводим значения функций из данного класса на широком диапазоне значений аргумента. Видно, что при небольших размерах входных данных значения функций отличаются незначительно, однако при росте этих размеров разница существенно возрастает. Эта таблица усиливает впечатление от рис. 1.1. Поэтому мы и будем изучать, что происходит при больших объемах входных данных, поскольку на малых объемах принципиальная разница оказывается скрытой.

Данные с рис. 1.1 и 1.2 иллюстрируют еще одно свойство функций. Быстрорастущие функции доминируют функции с более медленным ростом. Поэтому если мы обнаружим, что сложность алгоритма представляет собой сумму двух или нескольких таких функций, то будем часто

	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
1	0.0	1.0	0.0	1.0	1.0	2.0
2	1.0	2.0	2.0	4.0	8.0	4.0
5	2.3	5.0	11.6	25.0	125.0	32.0
10	3.3	10.0	33.2	100.0	1000.0	1024.0
15	3.9	15.0	58.6	225.0	3375.0	32768.0
20	4.3	20.0	86.4	400.0	8000.0	1048576.0
30	4.9	30.0	147.2	900.0	27000.0	1073741824.0
40	5.3	40.0	212.9	1600.0	64000.0	1099511627776.0
50	5.6	50.0	282.2	2500.0	125000.0	1125899906842620.0
60	5.9	60.0	354.4	3600.0	216000.0	1152921504606850000.0
70	6.1	70.0	429.0	4900.0	343000.0	1180591620717410000000.0
80	6.3	80.0	505.8	6400.0	512000.0	1208925819614630000000000.0
90	6.5	90.0	584.3	8100.0	729000.0	1 237940039285380000000000000.0
100	6.6	100.0	664.4	10000.0	1000000.0	1267650600228230000000000000000.0

Рис. 1.2. Классы роста функций

отбрасывать все функции кроме тех, которые растут быстрее всего. Если, например, мы установим при анализе алгоритма, что он делает $x^3 - 30x$ сравнений, то мы будем говорить, что сложность алгоритма растет как x^3. Причина этого в том, что уже при 100 входных данных разница между x^3 и $x^3 - 30x$ составляет лишь 0.3%. В следующем разделе мы формализуем эту мысль.

1.4.1. Классификация скоростей роста

Скорость роста сложности алгоритма играет важную роль, и мы видели, что скорость роста определяется старшим, доминирующим членом формулы. Поэтому мы будем пренебрегать младшими членами, которые растут медленнее. Отбросив все младшие члены, мы получаем то, что называется *порядком* функции или алгоритма, скоростью роста сложности которого она является. Алгоритмы можно сгруппировать по скорости роста их сложности. Мы вводим три категории: алгоритмы, сложность которых растет по крайней мере так же быстро, как данная функция, алгоритмы, сложность которых растет с той же скоростью, и алгоритмы, сложность которых растет медленнее, чем эта функция.

Омега большое

Класс функций, растущих по крайней мере так же быстро, как /, мы обозначаем через $\Omega(f)$ (читается *омега большое*). Функция *д* принадлежит этому классу, если при всех значениях аргумента n, больших некоторого порога по, значение $д(n) > cf(n)$ для некоторого положительного числа *с*. Можно считать, что класс $\Omega(f)$ задается указанием свой нижней границы: все функции из него растут по крайней мере так же быстро, как /.

Мы занимаемся эффективностью алгоритмов, поэтому класс $\Omega(f)$ не будет представлять для нас большого интереса: например, в $\Omega(n^2)$ входят все функции, растущие быстрее, чем n^2, скажем n^3 и 2".

О большое

На другом конце спектра находится класс $O(f)$ (читается *о большое)*. Этот класс состоит из функций, растущих не быстрее /. Функция / образует верхнюю границу для класса $O(f)$. С формальной точки зрения функция *д* принадлежит классу $O(f)$, если $g(n) < cf(n)$ для всех n, больших некоторого порога по, и для некоторой положительной константы *с*.

Этот класс чрезвычайно важен для нас. Про два алгоритма нас будет интересовать, принадлежит ли сложность первого из них классу о большое от сложности второго. Если это так, то, значит, второй алгоритм не лучше первого решает поставленную задачу.

Тета большое

Через $\Theta(f)$ (читается *тета большое*) мы обозначаем класс функций, растущих с той же скоростью, что и /. С формальной точки зрения этот класс представляет собой пересечение двух предыдущих классов, $\Theta(f) = \Omega(f) \cap O(f)$.

При сравнении алгоритмов нас будут интересовать такие, которые решают задачу быстрее, чем уже изученные. Поэтому, если найденный алгоритм относится к классу в, то он нам не очень интересен. Мы не будем часто ссылаться на этот класс.

Описание класса О большое

Проверить, принадлежит ли данная функция классу $O(f)$, можно двумя способами: либо с помощью данного выше определе-

ния, либо воспользовавшись следующим описанием:

$$\partial \in O(f), \text{ если } \lim_{n\to\infty} \frac{g(n)}{f(n)} = \text{ с для некоторой константы } c. \qquad (1.23)$$

Это означает, что если предел отношения $g(n)/f(n)$ существует и он меньше бесконечности, то $\partial \in O(f)$. Для некоторых функций это не так-то просто проверить. По правилу Лопиталя, в таком случае можно заменить предел самих функций пределом их производных.

Обозначение

Каждый из классов $\Theta(f)$, $\Omega(f)$ и $O(f)$ является множеством, и поэтому имеет смысл выражение «∂ — элемент этого множества». В анализе, однако, нередко пишут, скажем, $\partial = O(f)$, что на самом деле означает $g \in O(f)$.

1.4.2. Упражнения

1) Расположите следующие функции в порядке возрастания. Если некоторые из них растут с одинаковой скоростью, то объедините их овалом.

2^n	$\log_2 \log_2 n$	$n^3 + \log_2 n$
$\log_2 n$	$n - n^2 + 5n^3$	2^{n-1}
n^2	n^3	$n \log_2 n$
$(\log_2 n)^2$	$\sqrt{n}$	6
$n!$	n	$(3/2)^n$

2) Для каждой из приведенных ниже пар функций f и ∂ выполняется одно из равенств: либо $f = O(\partial)$, либо $\partial = O(f)$, но не оба сразу. Определите, какой из случаев имеет место.

   а) $f(n) = (n^2 - n)/2$, $g(n) = 6n$

   б) $f(n) = n + 2\sqrt{n}$, $g(n) = n^2$

   в) $f(n) = n + n \log_2 n$, $g(n) = n\sqrt{n}$

   г) $f(n) = n^2 + 3n + 4$, $g(n) = n^3$

д) $f(n) = n \log_2 n, \quad g(n) = n\sqrt{n}/2$

е) $f(n) = n + \log_2 \mathrm{n}, \quad g(n) = \sqrt{n}$

ж) $f(n) = 2\log_2 n^2, \quad g(n) = \log_2 \mathrm{n} + 1$

з) $f(n) = 4n \log_2 n + \mathrm{n}, \quad g(n) = (n^2 - n)/2$

1.5. Алгоритмы вида «разделяй и властвуй»

Как указано во введении, алгоритмы вида «разделяй и властвуй» обеспечивают компактный и мощный инструмент решения различных задач; в этом параграфе мы займемся не столько разработкой подобных алгоритмов, сколько их анализом. При подсчете числа сравнений в циклах нам достаточно подсчитать число сравнений в одной итерации цикла и умножить его на число итераций. Этот подсчет становится сложнее, если число итераций внутреннего цикла зависит от параметров внешнего.

Подсчет итераций алгоритмов вида «разделяй и властвуй» не очень прост: он зависит от рекурсивных вызовов, от подготовительных и завершающих действий. Обычно неочевидно, сколько раз выполняется та или иная функция при рекурсивных вызовах. В качестве примера рассмотрим следующий алгоритм вида «разделяй и властвуй».

```
DivideAndConquer(data,N,solution)
data      набор входных данных
N         количество значений в наборе
solution  решение задачи

if (N <= SizeLimit) then
  DirectSolution(data,N,solution)
else
  DivideInput(data,N,smallerSets,smallerSizes,numberSmaller)
for i=1 to numberSmaller do
  DivideAndConquer(smallerSets[i],smallerSizes[i],smallSol[i])
end for
CombineSolutions(smallSol,numberSmaller,solution)
end if
```

Этот алгоритм сначала проверяет, не мал ли размер задачи настолько, чтобы решение можно было найти с помощью простого нерекурсив-

ного алгоритма (названного в тексте **DirectSolution**), и если это так, то происходит его вызов. Если задача слишком велика, то сначала вызывается процедура **DivideInput**, которая разбивает входные данные на несколько меньших наборов (их число равно **numberSmaller**). Эти меньшие наборы могут быть все одного размера или их размеры могут существенно различаться. Каждый из элементов исходного входного набора попадает по крайней мере в один из меньших наборов, однако один и тот же элемент может попасть в несколько наборов. Затем происходит рекурсивный вызов алгоритма **DivideAndConquer** на каждом из меньших входных множеств, а функция CombineSolutions сводит полученные результаты воедино.

Факториал натурального числа несложно вычислить в цикле, однако в нижеследующем примере нам понадобится рекурсивный вариант этого вычисления. Факториал числа N равен N, умноженному на факториал числа $N - 1$. Поэтому годится следующий алгоритм:

```
Factorial(N)
N          число, факториал которого нам нужен
Factorial  возвращает целое число

if (N=1) then
   return 1
else
   smaller = N-1
   answer=Factorial(smaller)
   return (N*answer)
end if
```

Шаги этого алгоритма просты и понятны, и мы можем сравнить их с приведенным выше стандартным алгоритмом. Хотя мы и упоминали ранее в этой главе, что операция умножения сложнее сложения и поэтому умножения надо подсчитывать отдельно, для упрощения примера мы не будем учитывать эту разницу.

При сопоставлении этих двух алгоритмов видно, что предельным размером данных в нашем случае служит 1, и при таких данных никаких арифметических операций не выполняется. Во всех остальных случаях мы переходим к залогу else. Первым шагом в общем алгоритме служит «разбиение ввода на более мелкие части»; в алгоритме вычисления факториала этому шагу соответствует вычисление переменной smaller, которое требует одного вычитания. Следующий шаг общего

алгоритма представляет собой рекурсивный вызов процедур обработки более мелких данных; в алгоритме вычисления факториала — это один рекурсивный вызов, и размер задачи в нем на 1 меньше предыдущего. Последний шаг в общем алгоритме состоит в объединении решений; в алгоритме вычисления факториала это умножение в последнем операторе return.

Эффективность рекурсивного алгоритма

Насколько эффективен рекурсивный алгоритм? Сможем ли мы подсчитать его эффективность, если будем знать, что алгоритм непосредственного вычисления квадратичен, разбиение входных данных логарифмично, а объединение решений линейно (все в зависимости от размеров входных данных)[4], и что входные данные разбиваются на восемь частей, размер каждой из которых равен четверти исходного? Эту задачу не так-то легко решить, не очень-то понятно даже, как к ней приступать. Однако оказывается, что анализ алгоритмов вида «разделяй и властвуй» очень прост, если шаги Вашего алгоритма поставлены в соответствие шагам предложенного выше общего алгоритма этого вида: непосредственное вычисление, разбиение входа, некоторое количество рекурсивных вызовов и объединение полученных решений. Если известно, как эти шаги сочетаются друг с другом, и известна сложность каждого шага, то для определения сложности алгоритма вида «разделяй и властвуй» можно воспользоваться следующей формулой:

$$\mathrm{DAC}(N) = \begin{cases} \mathrm{DIR}(N) & \text{при } N < \mathrm{SizeLimit}, \\ \mathrm{DIV}(N) + \sum_{i=1}^{\mathrm{numberSmaller}} \mathrm{DAC}(\mathtt{smallerSizes}[i]) + \mathrm{COM}(N) & \\ & \text{при } N > \mathrm{SizeLimit}, \end{cases}$$

где DAC — сложность алгоритма **DivideAndConquer**,

DIR — сложность алгоритма **DirectSolution**,

DIV — сложность алгоритма **DivideInput**,

COM — сложность алгоритма CombineSolutions.

[4] **Линейность** алгоритма означает, что его сложность принадлежит классу $O(N)$, сложность квадратичного алгоритма принадлежит классу $O(N^2)$, а логарифмического — классу $O(\log N)$.

При наличии этой общей формулы ответ на вопрос, поставленный в начале предыдущего абзаца, оказывается совсем простым. Нам нужно лишь подставить в общую формулу известные сложности каждого куска. В результате получим

$$\mathrm{DAC}(N) = \begin{cases} N^2 & \text{при TV} < \text{SizeLimit}, \\ \log_2 \mathrm{TV} + \sum_{i=1}^{8} \mathrm{DAC}(N/4) + \mathrm{TV} & \text{при TV} > \text{SizeLimit}, \end{cases}$$

или даже проще, поскольку размеры всех меньших множеств одинаковы:

$$\mathrm{DAC}(N) = \begin{cases} N^2 & \text{при } N < \text{SizeLimit}, \\ \log_2 N + 8\,\mathrm{DAC}(N/4) + N & \text{при } N > \text{SizeLimit}. \end{cases}$$

Равенства такого вида называются рекуррентными, поскольку значение функции выражено в терминах себя самого. Нам бы хотелось найти выражение для сложности, зависящее только от TV, и не зависящее от других вызовов той же функции. Процесс исключения рекурсии из подобных равенств будет описан в § 1.6, где рекуррентные соотношения изучаются подробно.

Вернемся к примеру с факториалом. Мы сопоставили все этапы алгоритма вычисления факториала с общим алгоритмом `DivideAndConquer`. Воспользуемся теперь этим сопоставлением и определим значения, которые следует подставить в приведенную выше общую формулу. Непосредственные вычисления в функции Factorial не требуют действий, каждый из алгоритмов разбиения входных данных и объединения результатов требует по одному действию, и рекурсивный вызов решает задачу, размер данных которой на единицу меньше исходного. В результате мы получаем следующее рекуррентное соотношение на количество вычислений в функции Factorial:

$$\mathrm{Calc}(N) = \begin{cases} 0 & \text{при TV} = 1, \\ 1 + \mathrm{Calc}(N - 1) + 1 & \text{при TV} > 1. \end{cases}$$

1.5.1. Метод турниров

Метод турниров основан на рекурсии, и с его помощью можно решать различные задачи, в которых информация, полученная в результате первого прохода по данным, может облегчить последующие проходы. Если мы воспользуемся им для поиска наибольшего значения, то он потребует построения бинарного дерева, все элементы которого являются листьями. На каждом уровне два элемента объединены в пару, причем наибольший из двух элементов копируется в родительский узел. Процесс повторяется до достижения корневого узла. Полное дерево турнира для фиксированного набора данных изображено на рис. 1.3.

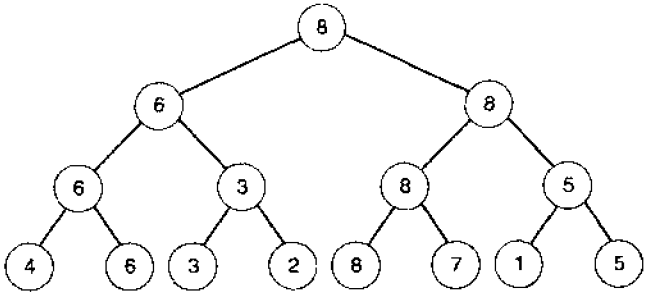


Рис. 1.3. Дерево турнира для набора из восьми значений

В упражнении 5 раздела 1.1.3 упоминалось, что мы разработаем алгоритм для поиска второго по величине элемента списка за количество сравнений порядка N. Метод турниров поможет нам в этом. В результате всякого сравнения мы получаем «победителя» и «проигравшего». Проигравших мы забываем, и вверх по дереву двигаются только победители. Всякий элемент, за исключением наибольшего, «проигрывает» в точности в одном сравнении. Поэтому для построения дерева турнира требуется $N - 1$ сравнение.

Второй по величине элемент мог проиграть только наибольшему. Спускаясь по дереву вниз, мы составляем список элементов, проигравших наибольшему. Формулы для деревьев из раздела 1.3.2 показывают, что число таких элементов не превосходит $\lceil \log_2 N \rceil$. Их поиск по дереву потребует $\lceil \log_2 N \rceil$ сравнений; еще $\lceil \log_2 N1 \rceil - 1$ сравнений необходимо для выбора наибольшего из них. На всю работу уйдет $N + 2\lceil \log_2 N \rceil - 2$, т. е. $O(N)$ сравнений.

С помощью метода турниров можно и сортировать список значений. В главе 3 нам встретится сортировка кучей, основанная на методе турниров.

1.5.2. Нижние границы

Алгоритм является оптимальным, если не существует алгоритма, работающего быстрее. Как узнать, оптимален ли наш алгоритм? Или, быть может, он не оптимален, но все-таки достаточно хорош? Для ответа на эти вопросы мы должны знать наименьшее количество операций, необходимое для решения конкретной задачи. Для этого мы должны изучать именно задачу, а не решающие ее алгоритмы. Полученная в результате нижняя граница указывает объем работы, необходимый для решения этой задачи, и показывает, что любой алгоритм, претендующий на то, что решает ее быстрее, обязан неправильно обрабатывать некоторые ситуации.

Для анализа процесса сортировки списка из трех чисел вновь воспользуемся бинарным деревом. Внутренние узлы дерева будем помечать парами сравниваемых элементов. Порядок элементов, обеспечивающий проход по данной ветви дерева, изображается в соответствующем листе дерева. Дерево для списка из трех элементов изображено на рис. 1.4. Такое дерево называется деревом решения.

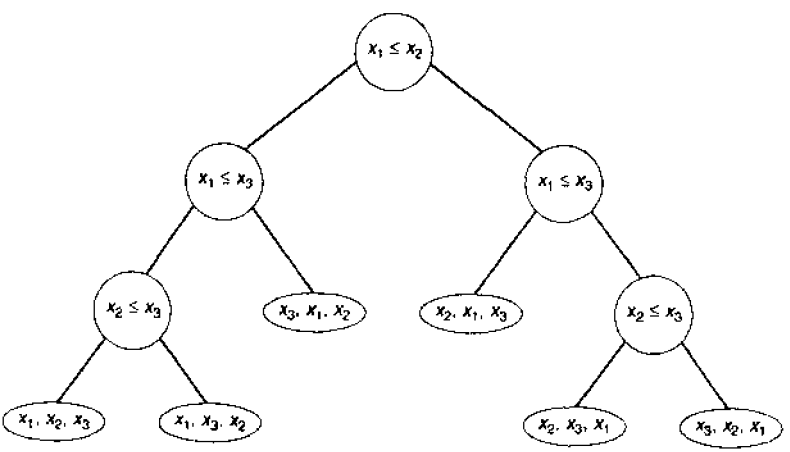


Рис. 1.4. Дерево решений для сортировки трехэлементного списка

Всякий алгоритм сортировки создает свое дерево решений в зависимости от того, какие элементы он сравнивает. Самый длинный путь в дереве решений от корня к листу соответствует наихудшему случаю. Наилучшему случаю отвечает кратчайший путь. Средний случай описывается частным от деления числа ребер в дереве решений на число листов в нем. На первый взгляд, ничего не стоит нарисовать дерево решений и подсчитать нужные числа. Представьте себе, однако, размер дерева решений при сортировке 10 чисел. Как уже говорилось, число возможных порядков равно 3 628 800. Поэтому в дереве будет по меньшей мере 3 628 800 листьев, а может и больше, поскольку к одному и тому же порядку можно прийти в результате различных последовательностей сравнений. В таком дереве должно быть не меньше 22 уровней.

Как же можно получить границы для сложности алгоритма с помощью дерева решений? Мы знаем, что корректный алгоритм должен упорядочивать любой список данных, независимо от исходного порядка элементов в нем. Для любой перестановки входных значений должен существовать по крайней мере один лист, а это означает, что в дереве решений должно быть не меньше $N!$ листьев. Если алгоритм по-настоящему эффективен, то каждая перестановка появится в нем лишь однажды. Сколько уровней должно быть в дереве с $N!$ листьями? Мы уже видели, что в каждом очередном уровне вдвое больше узлов, чем в предыдущем. Число узлов на уровне K равно 2^{K-1}, поэтому в нашем дереве решений будет L уровней, где L — наименьшее целое число, для которого $N! < 2^{L-1}$. Логарифмируя это неравенство, получаем

$$\log_2 N! \leq L - 1.$$

Можно ли избавиться от факториала, чтобы найти наименьшее значение L? Обратимся к свойствам факториала. Воспользуемся тем, что

$$\log_2 N! = \log_2(N(N- l)(N - 2) \ldots 1),$$

откуда в силу равенства (1.5)

$$\log_2(N(N- 1)(N - 2) \ldots 1) = \log_2 N + \log_2 (N - 1) + \cdots + \log_2 1$$

$$= \sum_{i=1}^{N} \log_2 i.$$

Из равенства (1.21) получаем

$$\sum_{i=1}^{N} \log_2 i \approx N \log_2 N - 1.5, \log_2(N!) \sim N \log_2 N.$$

Это означает, что минимальная глубина L дерева решений для сортировки имеет порядок $O(N \log_2 N)$. Теперь мы знаем, что любой алгоритм сортировки порядка $O(N \log N)$является наилучшим, и его можно считать оптимальным. Кроме того, мы знаем, что любой алгоритм, работающий быстрее, чем за $O(N \log N)$операций, не может работать правильно.

При выводе нижней границы для алгоритма сортировки предполагалось, что алгоритм работает путем последовательного попарного сравнения элементов списка. В главе 3 мы познакомимся с другим алгоритмом сортировки (корневая сортировка), который требует линейного времени. Для достижения цели этот алгоритм не сравнивает значения ключей, а разбивает их на «кучки».

1.5.3. Упражнения

1) Числа Фибоначчи можно подсчитывать с помощью приведенного ниже алгоритма. Выпишите рекуррентное соотношение на число «сложений» в алгоритме. Четко выделите непосредственные вычисления, разбиение входных данных и объединение решений.

```
int Fibonacci(N)
N   следует вычислить N-е число Фибоначчи
if  (N=1) or (N=2) then
    return 1
else
    return Fibonacci(N-1)+Fibonacci(N-2)
end if
```

2) Наибольший общий делитель (НОД) двух натуральных чисел M и N - - это наибольшее натуральное число, на которое делится и M, и N. Например, НОД(9,15)=3, а НОД(51,34)=17. Следующий алгоритм вычисляет наибольший общий делитель:

```
GCD(M,N)
M,N   интересующие нас числа
GCD возвращает цел. число,
    наибольший общий делитель чисел M и N

if  (M<N) then
    поменять местами M и N
```

```
end if
if (N=0) then
  return M
else
    quotient = M/N    //Примечание: деление целочисленное
    remainder = M mod N
    return GCD(N,remainder)
end if
```

Найдите рекуррентное соотношение для числа умножений (в данном случае, делений и взятий остатка по модулю) в приведенном алгоритме.

3) Пусть непосредственное (нерекурсивное) вычисление решает некоторую задачу за N^2 операций. Рекурсивному алгоритму при решении той же задачи требуется $N \log_2 N$ операций для разбиения входных данных на две равные части и $\log_2 N$ операций для объединения двух решений. Какой из двух алгоритмов более эффективен?

4) Нарисуйте дерево турнира для следующего набора значений: 13, 1, 7, 3, 9, 5, 2, 11, 10, 8, 6, 4, 12. Какие элементы будут сравниваться на втором этапе поиска второго по величине значения?

5) Найдите нижнюю границу числа сравнений при поиске по списку из N элементов. При обдумывании ответа постарайтесь представить себе, как выглядит дерево решений. *(Подсказка:* Узлы должны быть помечены местоположением ключа.) Что можно сказать о числе необходимых для поиска сравнений, если упаковывать узлы настолько плотно, насколько это возможно?

1.6. Рекуррентные соотношения

Рекуррентные соотношения на сложность алгоритма выводятся непосредственно из вида алгоритма, однако с их помощью нельзя быстро вычислить эту сложность. Для этого следует привести рекуррентные соотношения к так называемому замкнутому виду, отказавшись от их рекуррентной природы. Производится такое приведение посредством последовательных подстановок, позволяющих уловить общий принцип. Проще всего понять этот принцип на ряде примеров.

Рекуррентное соотношение задается в одной из двух форм. Первая используется, если простых случаев немного:

$$T(n) = 2T(n-2) - 15;$$
$$T(2) = 40;$$
$$T(1) = 40.$$

Вторая форма используется, если количество простых случаев относительно велико:

$$T(n) = \begin{cases} 4, & \text{если } n < 4; \\ 4T(n/2) - 1 & \text{в противном случае.} \end{cases}$$

Эти формы эквивалентны. От второй к первой можно перейти, просто выписав список всех простых значений. Это означает, что второе из приведенных выше рекуррентных соотношений можно переписать в виде

$$T(n) = 4T(n/2) - 1;$$
$$\Gamma(4) = 4;$$
$$\Gamma(3) = 4;$$
$$\Gamma(2) = 4;$$
$$T(1) = 4.$$

Рассмотрим следующее рекуррентное соотношение:

$$T(n) = 2T(n-2) - 15;$$
$$T(2) = 40;$$
$$\Gamma(1) = 40.$$

Мы хотим подставить в первое равенство эквивалентное выражение для $T(n-2)$. Для этого заменим каждое вхождение n в это уравнение на $n-2$:

$$T(n-2) = 2T(n-2-2) - 15$$
$$= 2T(n-4) - 15.$$

Теперь, однако, мы должны исключить значение $T(n-4)$. Тем самым, мы должны выполнить последовательность подобных подстановок. Проделаем это для небольших последовательных значений:

$$T(n-2) = 2T(n-4) - 15;$$
$$T(n-4) = 2T(n-6) - 15;$$
$$T(n-6) = 2T(n-8) - 15;$$
$$T(n-8) = 2T(n-10) - 15;$$
$$\Gamma(\text{п} - 10) = 2T(n-12) - 15.$$

Будем теперь подставлять результаты вычислений обратно в исходное уравнение. При этом мы не будем до конца упрощать результат, иначе мы рискуем упустить общую схему. Подстановки дают

$$T(n) = 2T(n-2) - 15 = 2(2T(n-4) - 15) - 15,$$
$$T(n) = 4T(n-4) - 2 \cdot 15 - 15;$$

$$T(n) = 4(2T(n-6) - 15) - 2 \cdot 15 - 15,$$
$$T(n) = 8T(n-6) - 4 \cdot 15 - 2 \cdot 15 - 15;$$

$$\Gamma(\text{п}) = 8(2T(n-8) - 15) - 4 \cdot 15 - 2 \cdot 15 - 15,$$
$$T(n) = 16\text{Т}(\text{п} - 8) - 8 \cdot 15 - 4 - 15 - 2 \cdot 15 - 15;$$

$$T(n) = 16(2T(n-10) - 15) - 8 \cdot 15 - 4 \cdot 15 - 2 \cdot 15 - 15,$$
$$T(n) = 32T(n-10) - 16 \cdot 15 - 8 \cdot 15 - 4 \cdot 15 - 2 \cdot 15 - 15;$$

$$T(n) = 32(2T(n-12) - 15) - 16 \cdot 15 - 8 \cdot 15 - 4 \cdot 15 - 2 \cdot 15 - 15,$$
$$T(n) = 64T(n-12) - 32 \cdot 15 - 16 \cdot 15 - 8 \cdot 15 - 4 \cdot 15 - 2 \cdot 15 - 15.$$

Возможно Вы уже уловили общую схему. Прежде всего, слагаемые с конца в каждом равенстве представляют собой число -15, умноженное на очередную степень двойки. Во-вторых, можно заметить, что коэффициент при рекурсивном вызове функции Т является степенью двойки. Кроме того, аргумент функции Т всякий раз уменьшается на 2.

Можно спросить себя, когда этот процесс завершится. Вернувшись к исходному равенству, вспомним, что мы зафиксировали значения $T(1)$ и T(2). Сколько подстановок должны мы сделать, прежде чем доберемся до одной из этих величин? При n четном имеем 2 = п — (п — 2). Это означает, что мы должны сделать (п — 2)/2 — 1 подстановок, что дает $n/2$ — 1 слагаемых с множителем -15 и $n/2$ — 1 степень двойки перед Т. Проверим, что происходит при п = 14. В этом случае, согласно предыдущему предложению, мы должны сделать пять подстановок; получим шесть слагаемых с множителем -15, и коэффициент при T(2) будет равен 2^6. Как раз такой результат и получается при подстановке п = 14 в последнее равенство.

А что если число n нечетно? Будут ли эти формулы работать по-прежнему? Рассмотрим значение n = 13. Единственное изменение, которое произойдет в равенстве — аргументом функции T будет служить 1, а не 2, однако значение п/2 — 1 будет равно 5 (а не 6). При нечетном n мы должны вместо п/2 — 1 взять п/2. В ответе мы должны рассмотреть два случая.

при четном п:

$$\text{Т(п)} = 2^{(n/2)-1}T(2) - 15\sum_{i=0}^{(n/2)-1} 2^i,$$

$$\text{Г(п)} = 2^{(n/2)-1} \cdot 40 - 15\sum_{i=0}^{(n/2)-1} 2^i$$

при нечетном п.

$$T(n) = 2^{n/2}T(2) - 15\sum_{i=0}^{n/2} 2^i,$$

$$T(n) = 2^{n/2} \cdot 40 - 15\sum_{i=0}^{n/2} 2^i$$

Теперь, применив равенство (1.17), для четного п мы получаем

$$\begin{aligned} T(n) &= 2^{(n-2)-1} \cdot 40 - 15 \cdot (2^{n/2} - 1) \\ &= 2^{n/2} \cdot 20 - 2^{n/2} \cdot 30 + 15 \\ &= 2^{n/2}(20 - 15) + 15 \\ &= 2^{n/2} \cdot 5 + 15, \end{aligned}$$

а при п нечетном

$$\text{Т(п)} = 2^{n/2} \cdot 40 - 15 \cdot (2^{n/2+1} - 1)$$

$$= 2^{n/2} \cdot 40 - 2^{n/2} \cdot 30 + 15$$

$$= 2^{n/2}(40 - 30) + 15$$

$$= 2^{n/2} \cdot 10 + 15.$$

Рассмотрим еще одно рекуррентное соотношение:

$$T(n) = \begin{cases} 5, & \text{если } n < 4; \\ 4T(n/2) - 1 & \text{в противном случае.} \end{cases}$$

Будем рассуждать так же, как и в предыдущем случае. Сначала будем подставлять только значения n; при этом в каждом очередном равенстве получится только половина аргумента предыдущего. В результате приходим к равенствам

$$T(n/2) = 4T(n/4) - 1;$$

$$T(n/4) = 4T(n/8) - 1;$$

$$T(n/8) = 4T(n/16) - 1;$$

$$T(n/16) = 4T(n/32) - 1;$$

$$T(n/32) = 4T(n/64) - 1.$$

Подстановка этих равенств назад в исходное уравнение дает

$$T(n) = 4T(n/2) - 1 = 4(\text{T(п/4)} - 1) - 1,$$

$$T(n) = 16T(n/4) - 4 \cdot 1 - 1;$$

$$T(n) = 16(4T(n/8) - 1) - 4 - 1 - 1,$$

$$T(n) = 64T(n/8) - 16 \cdot 1 - 4 \cdot 1 - 1;$$

$$\text{T(п)} = 64(T(n/16) - 1) - 16 \cdot 1 - 4 \cdot 1 - 1,$$

$$T(n) = 256T(n/16) - 64 \cdot 1 - 16 \cdot 1 - 4 \cdot 1 - 1;$$

$$T(n) = 256(4T(n/32) - 1) - 64 \cdot 1 - 16 \cdot 1 - 4 \cdot 1 - 1,$$

$$T(n) = 1024T(n/32) - 256 \cdot 1 - 64 \cdot 1 - 16 \cdot 1 - 4 \cdot 1 - 1;$$

$$T(n)= 1024(4T(n/64) - 1) - 256 \cdot 1 - 64 \cdot 1 - 16 \cdot 1 - 4 \cdot 1 - 1,$$

$$T(n) = 4096T(n/64) - 1024 \cdot 1 - 256 \cdot 1 - 64 \cdot 1 - 16 \cdot 1 - 4 \cdot 1 - 1.$$

Видно, что коэффициент при -1 увеличивается при каждой подстановке в некоторую степень четверки раз, а степень двойки, на которую мы делим аргумент, всякий раз на 1 больше наибольшей степени четверки при -1. Кроме того степень четверки в коэффициенте при T такая же, что и степень двойки, на которую мы делим аргумент. При $T(n/2^i)$ этот коэффициент равен 4^i, а дальше идут слагаемые от -4^{i-1} до -1. При каком же значении i подстановки должны прекратиться? Поскольку значения явно даны при $n < 4$, можно остановиться, добравшись до $T(4) = T(n/2^{\log_2 n-2})$. В результате получаем

$$T(n) = 4^{\log_2 n-2}T(4) - \sum_{i=0}^{\log_2 n-3} 4^i.$$

Воспользуемся теперь явными значениями и равенством (1.18):

$$T(N) = 4^{\log_2 n-2} \cdot 5 - \frac{4^{\log_2 n-2} - 1}{4 - 1};$$

$$T(N) = 4^{\log_2 n-2} \cdot 5 - \frac{4^{\log_2 n-2} - 1}{3};$$

$$T(N) = \frac{15 \cdot 4^{\log_2 n-2} - 4^{\log_2 n-2} + 1}{3};$$

$$T(N) = \frac{4^{\log_2 n-2}(15 - 1) + 1}{3};$$

$$T(N) = \frac{4^{\log_2 n-2} \cdot 14 + 1}{3}.$$

Видно, что в замкнутом виде рекуррентное соотношение может оказаться и не очень простым и изящным, однако при переходе к нему мы избавляемся от рекурсивного «вызова» и можем поэтому быстро сравнивать полученные выражения и определять их порядки.

1.6.1. Упражнения

Приведите следующие рекуррентные соотношения к замкнутому виду.

1. $$\begin{cases} T(n) = 3T(n-1) - 15, \\ T(1) = 8. \end{cases}$$

2. $$\begin{cases} T(n) = T(n-1) + n - 1. \\ T(1) = 3. \end{cases}$$

3. $$\begin{cases} T(n) = 6T(n/6) + 2n + 3, \\ T(1) = 1, \end{cases}$$ если n — степень числа 6.

4. $$\begin{cases} T(n) = 4T(n/3) + 2n - 1, \\ T(1) = 2, \end{cases}$$ если n — степень числа 3.

1.7. Анализ программ

Предположим, что у нас есть большая сложная программа, которая работает медленнее, чем хотелось бы. Как обнаружить части программы, тонкая настройка которых привела бы к заметному ускорению ее работы?

Можно посмотреть на программу и найти подпрограммы (иногда называемые также процедурами или функциями), в которых много вычислений или циклов, и попытаться усовершенствовать их. Приложив значительные усилия, мы можем обнаружить, что эффект не слишком заметен, поскольку отобранные подпрограммы используются нечасто. Лучше сначала найти часто используемые подпрограммы и попробовать улучшить их. Один из способов поиска состоит в том, чтобы ввести набор глобальных счетчиков, по одному на каждую подпрограмму. При начале работы программы все счетчики обнуляются. Затем в каждую подпрограмму первой строкой вставляется команда увеличения соответствующего счетчика на 1. При всяком обращении к подпрограмме будет происходить увеличение счетчика, и в конце работы наш набор счетчиков укажет, сколько раз происходил вызов каждой подпрограммы. Тогда можно будет увидеть, какие подпрограммы вызывались часто, а какие — всего несколько раз.

Предположим, что в нашей программе некоторая простая подпрограмма вызывалась 50 000 раз, а каждая из сложных подпрограмм лишь однажды. Тогда нужно уменьшить число операций в сложных программах на 50 000, чтобы достичь того же эффекта, что производит удале-

ние всего одной операции в простой подпрограмме. Понятно, что простое улучшение одной подпрограммы найти гораздо проще, чем 50 000 улучшений в группе подпрограмм.

Счетчики можно использовать и на уровне подпрограмм. В этом случае мы создаем набор глобальных счетчиков, по одному в каждой значимой точке, которую мы можем предугадать. Предположим, что мы хотим узнать, сколько раз выполняется каждая из частей then и else некоторого оператора `if`. Тогда можно создать два счетчика, и увеличивать первый из них при попадании в часть then, а второй — при попадании в часть else. В конце работы программы эти счетчики будут содержать интересующую нас информацию. Другими значимыми точками могут оказаться вхождения в циклы и операторы ветвления. Вообще говоря, счетчики стоит устанавливать в любых местах возможной передачи управления.

В конце работы программы установленные счетчики будут содержать информацию о числе выполнений каждого из блоков подпрограммы. Затем можно исследовать возможность улучшить те части подпрограммы, которые выполняют наибольший объем работы.

Этот процесс важен, и во многих компьютерах и системах разработки программного обеспечения есть средства автоматического получения такой информации о программах.

Глава 2.

Алгоритмы поиска и выборки

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- читать и разрабатывать алгоритмы;
- пользоваться формулами для сумм и вероятностей, описанными в главе 1.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- объяснять структуру алгоритма последовательного поиска;
- проводить анализ наихудшего случая в алгоритме последовательного поиска;
- проводить анализ среднего случая в алгоритме последовательного поиска;
- объяснять структуру алгоритма двоичного поиска;
- проводить анализ наихудшего случая в алгоритме двоичного поиска;
- проводить анализ среднего случая в алгоритме двоичного поиска;
- объяснять структуру алгоритмов выборки и проводить их анализ.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все приведенные примеры и убедитесь, что Вы их поняли. Например, составьте список из 5-8 элементов, а затем проследите выполнение алгоритмов последовательного поиска и выборки на этом списке. Проделайте то же самое с алгоритмом двоичного поиска на упорядоченном списке из 7 или 15 элементов. Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

Поиск необходимой информации в списке — одна из фундаментальных задач теоретического программирования. При обсуждении алгоритмов поиска мы предполагаем, что информация содержится в записях, составляющих некоторый список, который представляет собой массив данных в программе. Записи, или элементы списка, идут в массиве последовательно и между ними нет промежутков. Номера записей в списке идут от 1 до N - - полного числа записей. В принципе записи могут быть составлены из полей, однако нас будут интересовать значения лишь одного из этих полей, называемого ключом. Списки могут быть неотсортированными или отсортированными по значению ключевого поля. В неотсортированном списке порядок записей случаен, а в отсортированном они идут в порядке возрастания ключа.

Поиск нужной записи в неотсортированном списке сводится к просмотру всего списка до того, как запись будет найдена. Это простейший из алгоритмов поиска. Мы увидим, что этот алгоритм не очень эффективен, однако он работает на произвольном списке.

В отсортированном списке возможен также двоичный поиск. Двоичный поиск использует преимущества, предоставляемые имеющимся упорядочиванием, для того, чтобы отбрасывать за одно сравнение больше одного элемента. В результате поиск становится более эффективным.

С поиском конкретного значения связана задача выборки, в которой требуется найти элемент, удовлетворяющий некоторым условиям. Скажем, нам может понадобиться пятый по величине элемент, седьмой с конца или элемент со средним значением. Мы обсудим два подхода к этой задаче.

2.1. Последовательный поиск

В алгоритмах поиска нас интересует процесс просмотра списка в поисках некоторого конкретного элемента, называемого целевым. При последовательном поиске мы всегда будем предполагать, хотя в этом и нет особой необходимости, что список не отсортирован, поскольку некоторые алгоритмы на отсортированных списках показывают лучшую производительность. Обычно поиск производится не просто для проверки того, что нужный элемент в списке имеется, но и для того, чтобы получить данные, относящиеся к этому значению ключа. Например, ключевое значение может быть номером сотрудника или порядковым номером, или любым другим уникальным идентификатором. После того, как нужный ключ найден, программа может, скажем, частично изменить связанные с ним данные или просто вывести всю запись. Во всяком случае, перед алгоритмом поиска стоит важная задача определения местонахождения ключа. Поэтому алгоритмы поиска возвращают индекс записи, содержащей нужный ключ. Если ключевое значение не найдено, то алгоритм поиска обычно возвращает значение индекса, превышающее верхнюю границу массива. Для наших целей мы будем предполагать, что элементы списка имеют номера от 1 до N. Это позволит нам возвращать 0 в случае, если целевой элемент отсутствует в списке. Для простоты мы предполагаем, что ключевые значения не повторяются.

Алгоритм последовательного поиска последовательно просматривает по одному элементу списка, начиная с первого, до тех пор, пока не найдет целевой элемент. Очевидно, что чем дальше в списке находится конкретное значение ключа, тем больше времени уйдет на его поиск. Это следует помнить при анализе алгоритма последовательного поиска.

Вот как выглядит полный алгоритм последовательного поиска.

```
SequentialSearch(list,target,N)
list      список для просмотра
target    целевое значение
N         число элементов в списке

for i=1 to N do
   if (target=list[i])
      return i
   end if
end for
return 0
```

2.1.1. Анализ наихудшего случая

У алгоритма последовательного поиска два наихудших случая. В первом случае целевой элемент стоит в списке последним. Во втором его вовсе нет в списке. Посмотрим, сколько сравнений выполняется в каждом из этих случаев. Мы предположили, что все ключевые значения в списке уникальны, и поэтому если совпадение произошло в последней записи, то все предшествующие сравнения были неудачными. Однако алгоритм проделывает все эти сравнения пока не дойдет до последнего элемента. В результате будет проделано N сравнений, где N - - число элементов в списке.

Чтобы проверить, что целевое значение в списке отсутствует, его придется сравнить со всеми элементами списка. Если мы пропустим какой-то элемент, то не сможем выяснить, отсутствует ли целевое значение в списке вообще или оно содержится в каком-то из пропущенных элементов. Это означает, что для выяснения того, что ни один из элементов не является целевым, нам потребуется N сравнений.

Так что N сравнений необходимо как для того, чтобы найти значение, содержащееся в последнем элементе списка, так и для того, чтобы выяснить, что целевого значения в списке нет. Ясно, что N дает верхнюю границу сложности любого алгоритма поиска, поскольку, если сравнений больше N, то это означает, что сравнение с каким-то элементом выполнялось по крайней мере дважды, а значит была проделана лишняя работа, и алгоритм можно улучшить.

Между понятиями верхней границы сложности и сложности в наихудшем случае есть разница. Верхняя граница присуща самой задаче, а понятие наихудшего случая относится к решающему ее конкретному алгоритму. В § 2.2 мы познакомимся с другим алгоритмом поиска, сложность которого в наихудшем случае меньше указанной верхней границы N.

2.1.2. Анализ среднего случая

Для алгоритмов поиска возможны два средних случая. В первом предполагается, что поиск всегда завершается успешно, во втором — что иногда целевое значение в списке отсутствует.

Если целевое значение содержится в списке, то оно может занимать одно из N возможных положений: оно может быть первым, вторым, третьим, четвертым и так далее. Мы будем предполагать все эти положения равновероятными, т.е. вероятность встретить каждое из них равна $1/N$.

Прежде, чем читать дальше, ответьте на следующие вопросы.

- Сколько сравнений требуется, если искомый элемент стоит в списке первым?
- А если вторым?
- А если третьим?
- А если последним из N элементов?

Если Вы внимательно посмотрели на алгоритм, то Вам несложно определить, что ответы будут выглядеть 1, 2, 3 и N соответственно. Это означает, что для каждого из N случаев число сравнений совпадает с номером искомого элемента. В результате для сложности в среднем случае мы получаем равенство[1]

$$\begin{aligned} A(N) &= \frac{1}{N}\sum_{i=1}^{N} i \\ &= \frac{1}{N} \cdot \frac{N(N+1)}{2} \quad \text{в силу равенства} \quad (1.15) \\ &= \frac{N+1}{2}. \end{aligned}$$

Если мы допускаем, что целевого значения может не оказаться в списке, то количество возможностей возрастает до $N + 1$.

Как мы уже видели, при отсутствии элемента в списке его поиск требует N сравнений. Если предположить, что все $N+1$ возможностей равновероятны, то получится следующая выкладка:

$$\begin{aligned} A(N) &= \frac{1}{N+1}\left(\sum_{i=1}^{N} i + N\right) \\ &= \frac{1}{N+1}\sum_{i=1}^{N} i + \frac{1}{N+1} \cdot N \end{aligned}$$

[1] Если эти формулы Вам незнакомы, обратитесь к разделу 1.2.1.

$$= \frac{1}{N+1} \cdot \frac{N(N+1)}{2} + \frac{N}{N+1}$$

$$= \frac{N}{2} + \frac{N}{N+1} = \frac{N}{2} + \frac{1}{N+1}$$

$$\approx \frac{N+2}{2}.$$

(Когда N становится очень большим, значение $1/(N+1)$ оказывается близким к 0.)

Видно, что допущение о возможности отсутствия элемента в списке увеличивает сложность среднего случая лишь на 1/2. Ясно, что по сравнению с длиной списка, которая может быть очень велика, эта величина пренебрежимо мала.

2.1.3. Упражнения

1) Последовательный поиск можно применять и на отсортированном списке. Напишите алгоритм `SortedSequentialSearch`, выдающий тот же результат, что и предыдущий алгоритм, однако работающий быстрее за счет того, что он останавливается, когда целевое значение оказывается меньше текущего значения в списке. При разработке алгоритма воспользуйтесь функцией Compare(x,y), определенной следующим образом:

$$\text{Compare}(x,y) = \begin{cases} -1 & \text{при } x < y \\ 0 & \text{при } x = y \\ 1 & \text{при } x > y. \end{cases}$$

Вызов функции Compare следует считать за одно сравнение, и ее лучше всего использовать как переключатель. Проведите анализ в наихудшем случае, в среднем случае при условии, что целевое значение найдено, и в среднем случае при условии, что целевое значение не найдено. *(Примечание:* В этом последнем случае число возможностей велико, поскольку досрочный выход может произойти в любом месте как только целевое значение становится меньше текущего.)

2) Какова средняя сложность последовательного поиска, если вероятность того, что целевое значение будет найдено, равна 0.25, тогда как вероятность его присутствия в первой половине списка в случае, если оно там вообще есть, равна 0.75?

2.2. Двоичный поиск

При сравнении целевого значения со средним элементом отсортированного списка возможен один из трех результатов: значения равны, целевое значение меньше элемента списка, либо целевое значение больше элемента списка. В первом, и наилучшем, случае поиск завершен. В остальных двух случаях мы можем отбросить половину списка.

Когда целевое значение меньше среднего элемента, мы знаем, что если оно имеется в списке, то находится перед этим средним элементом. Когда же оно больше среднего элемента, мы знаем, что если оно имеется в списке, то находится после этого среднего элемента. Этого достаточно, чтобы мы могли одним сравнением отбросить половину списка. При повторении этой процедуры мы сможем отбросить половину оставшейся части списка. В результате мы приходим к следующему алгоритму[2]:

```
BinarySearch(list,target,N)
list     список для просмотра
target   целевое значение
N        число элементов в списке

start=1
end=N
while start<=end do
   middle=(start+end)/2
   select(Compare(list[middle],target)) from
      case -1: start=middle+1
      case  0: return middle
      case  1: end=middle-1
   end select
end while
return 0
```

В этом алгоритме переменной start присваивается значение, на 1 большее, чем значение переменной middle, если целевое значение пре-

[2]Функция Compare(x,y) определена в упражнении 1 раздела 2.1.3. Как там сказано, она возвращает значения -1, 0 или 1, если, соответственно, значение x меньше, чем значение y, эти значения равны или значение x больше значения y. При анализе алгоритмов вызов функции Compare считается за одно сравнение.

вышает значение найденного среднего элемента. Если целевое значение меньше значения найденного среднего элемента, то переменной end присваивается значение, на 1 меньшее, чем значение переменной middle. Сдвиг на 1 объясняется тем, что в результате сравнения мы знаем, что среднее значение не является искомым, и поэтому его можно исключить из рассмотрения.

Всегда ли цикл останавливается? Если целевое значение найдено. то ответ, разумеется, утвердительный, поскольку выполняется оператор return. Если нужное значение не найдено, то на каждой итерации цикла либо возрастает значение переменной start, либо уменьшается значение переменной end. Это означает, что их значения постепенно сближаются. В какой-то момент эти два значения становятся равными, и цикл выполняется еще один раз при соблюдении равенств start=end=middle. После этого прохода (если элемент с этим индексом не является искомым) либо переменная start окажется на 1 больше, чем middle и end, либо наоборот, переменная end окажется на 1 меньше, чем middle и start. В обоих случаях условие цикла while окажется ложным, и цикл больше выполняться не будет. Поэтому выполнение цикла завершается всегда.

Возвращает ли этот алгоритм правильный результат? Если целевое значение найдено, то ответ безусловно утвердительный, поскольку выполнен оператор return. Если же средний элемент оставшейся части списка не подходит, то на каждом проходе цикла происходит исключение половины оставшихся элементов, поскольку все они либо чересчур велики, либо чересчур малы. В предыдущем абзаце уже говорилось, что в результате мы придем к единственному элементу, который следует проверить[3]. Если это нужный нам ключ, то будет возвращено значение переменной middle. Если же значение ключа отлично от искомого, то значение переменной start превысит значение end или наоборот, значение переменной end станет меньше значения start. Если бы целевое значение содержалось в списке, то оно было бы меньше или больше значения элемента middle. Однако значения переменных start и end показывают, что предыдущие сравнения исключили все остальные возможности, и поэтому целевое значение отсутствует в списке. Значит цикл завершит работу, а возвращенное значение будет равно нулю, что

[3]Это вытекает и из того, что нам приходится последовательно делить пополам нацело. Вне зависимости от исходного размера списка при последовательном делении пополам (и отбрасывании при необходимости дробной части) мы неизбежно придем к списку из одного элемента.

указывает на неудачу поиска. Таким образом, алгоритм возвращает верный ответ.

Поскольку алгоритм всякий раз делит список пополам, мы будем предполагать при анализе, что $N = 2^k - 1$ для некоторого k. Если это так, то сколько элементов останется при втором проходе? А при третьем? Вообще говоря, ясно, что если на некотором проходе цикл имеет дело со списком из $2^j - 1$ элементов, то в первой половине списка $2^{j-1} - 1$ элементов, один элемент в середине, и $2^{j-1} - 1$ элементов во второй половине списка. Поэтому к следующему проходу остается $2^{j-1} - 1$ элемент (при $1 < j < k$). Это предположение позволяет упростить последующий анализ, но необходимости в нем нет, как Вы увидите в последующих упражнениях.

2.2.1. Анализ наихудшего случая

В предыдущем абзаце мы показали, что степень двойки в длине оставшейся части списка при всяком проходе цикла уменьшается на единицу. Было также показано, что последняя итерация цикла производится, когда размер оставшейся части становится равным 1, а это происходит при $j = 1$ (так как $2^1 - 1 = 1$). Это означает, что при $N = 2^k - 1$ число проходов не превышает k. Решая последнее уравнение относительно k, мы заключаем, что в наихудшем случае число проходов равно $k = \log_2(N + 1)$.

В анализе может помочь и дерево решений для процесса поиска. В узлах дерева решений стоят элементы, которые проверяются на соответствующем проходе. Элементы, проверка которых будет осуществляться в том случае, если целевое значение меньше текущего, располагаются в левом поддереве от текущего элемента, а сравниваемые в случае, если целевое значение больше текущего — в правом поддереве. Дерево решений для списка из семи элементов изображено на рис. 2.1. В общем случае дерево относительно сбалансировано, поскольку мы всегда выбираем середину различных частей списка. Поэтому для подсчета числа сравнений мы можем воспользоваться формулами для бинарных деревьев из раздела 1.3.2.

Поскольку мы предполагаем, что $N = 2^* - 1$, соответствующее дерево решений всегда будет полным. В нем будет k уровней, где $k = \log_2(N+ 1)$. Мы делаем по одному сравнению на каждом уровне, поэтому полное число сравнений не превосходит $\log_2(N + 1)$.

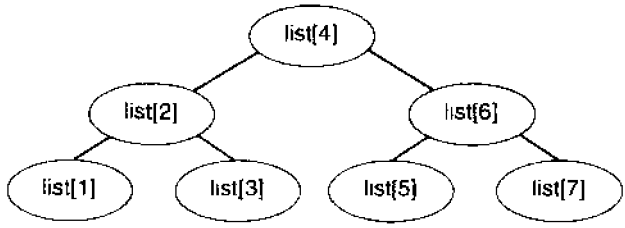


Рис. 2.1. Дерево решений для поиска в списке из семи элементов

2.2.2. Анализ среднего случая

Как и в случае последовательного поиска, при анализе среднего случая мы рассмотрим две возможности. В первом случае целевое значение наверняка содержится в списке, а во втором его может там и не быть.

В первой ситуации у целевого значения N возможных положений. Мы будем считать, что все они равноправны, и вероятность каждого из них тем самым равна $1/N$. Если рассмотреть бинарное дерево, описывающее процесс поиска, то видно, что для поиска корневого элемента дерева — единственного элемента уровня 1 — требуется одно сравнение, для поиска элементов в узлах второго уровня -- два сравнения, третьего уровня -- три сравнения. Вообще, для поиска элементов в узлах уровня i требуется i сравнений. В разделе 1.3.2 показано, что на уровне i имеется 2^{i-1} узел, и при $N = 2^k - 1$ в дереве k уровней. Это означает, что полное число сравнений для всех возможных случаев можно подсчитать, просуммировав произведения числа узлов на каждом уровне на число сравнений на этом уровне. В результате анализ среднего случая дает

$$A(N) = \frac{1}{N}\sum_{i=1}^{k} i2^{i-1} \qquad (\text{для } N = 2^k - 1)$$

$$= \frac{1}{N}\,\frac{1}{2}\sum_{i=1}^{k} i2^{i}.$$

Воспользовавшись равенством (1.19), мы можем упростить последнее выражение:

$$
\begin{aligned}
A(N) &= \frac{1}{N} \cdot \frac{1}{2}((k-1)2^{k+1}+2) \\
&= \frac{1}{N}((k-1)2^{k}+1)) \\
&= \frac{1}{N}(k2^{k}-2^{k}+1) \\
&= \frac{k2^{k}-(2^{k}-1)}{N} \\
&= \frac{k2^{k}-N}{N} \\
&= \frac{k2^{k}}{N}-1.
\end{aligned}
$$

Поскольку $N = 2^k - 1$, справедливо равенство $2^k = N + 1$. Поэтому

$$
\begin{aligned}
A(N) &= \frac{k(N+1)}{N}-1 \\
&= \frac{kN+k}{N}-1.
\end{aligned}
$$

При росте N значение k/N становится близким к нулю, и

$$
\begin{aligned}
A(N) &\approx k-1 \qquad \text{(для } N = 2^k - 1\text{)}; \\
A(N) &\approx \log_2(N+1)-1.
\end{aligned}
$$

Рассмотрим теперь вторую возможность, когда целевого значения может не оказаться в списке. Число возможных положений элемента в списке по-прежнему равно N, однако на этот раз есть еще $N + 1$ возможностей для целевого значения не из списка. Число возможностей равно $N + 1$, поскольку целевое значение может быть меньше первого элемента в списке, больше первого, но меньше второго, больше второго, но меньше третьего, и так далее, вплоть до возможности того, что целевое значение больше N-го элемента. В каждом из этих случаев отсутствие элемента в списке обнаруживается после k сравнений. Всего в вычислении участвует $2N + 1$ возможностей. Подводя итог, получаем

$$
A(N) = \frac{1}{2N+1}\left(\sum_{i=1}^{k} i2^{i-1} + (N+1)k\right) \qquad \text{для } N = 2^k - 1.
$$

Серия подстановок, аналогичная приведенной выше, дает

$$
\begin{aligned}
A(N) &= \frac{((k-1)2^k+1)+(N+1)k}{2N+1} \\
&= \frac{((k-1)2^k+1)+(2^k-1+1)k}{2(2^k-1)+1} \\
&= \frac{(k2^k-2^k+1)+2^k k}{2^{k+1}-1} \\
&= \frac{k2^{k+1}-2^k+1}{2^{k+1}-1} \\
&\approx \frac{k2^{k+1}-2^k+1}{2^{k+1}} \\
&\approx k-\frac{1}{2} = \log_2(N+1)-\frac{1}{2} \quad \text{для } N = 2^k-1.
\end{aligned}
$$

Последняя величина лишь незначительно превышает результат для случая, когда известно, что элемент находится в списке. Например, для списка из $2^{20} - 1 = 1\ 048\ 575$ элементов в первом случае мы получаем результат около 19, а во втором — около 19.5.

2.2.3. Упражнения

1) Нарисуйте дерево решений для алгоритма двоичного поиска в списке из 12 элементов. Внутренние узлы дерева должны быть помечены проверяемыми элементами, левый непосредственный потомок внутреннего узла должен указывать, что происходит, когда целевое значение меньше проверяемого элемента, правый — когда оно больше.

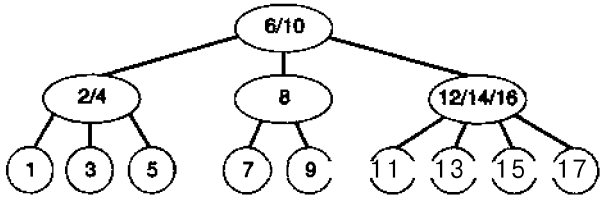
2) При анализе алгоритма двоичного поиска мы все время предполагали, что длина списка равна $2^k - 1$ для некоторого k. Здесь мы займемся случаем, когда это не так.

   а) Что дает анализ наихудшего случая при $N \neq 2^k - 1$?

   б) Что дает анализ среднего случая при $N \neq 2^k - 1$ в предположении, что целевое значение присутствует в списке? *Подсказка:* Подумайте, какие изменения при такой длине списка происходят на нижнем уровне дерева поиска.

в) Что дает анализ среднего случая при $N \neq 2^k - 1$ в предположении, что целевое значение может и отсутствовать в списке? *Подсказка:* Подумайте, какие изменения при такой длине списка происходят на нижнем уровне дерева поиска.

3) При выполнении двоичного поиска на большом наборе данных число необходимых сравнений может оставаться довольно большим. Так, например, один поиск в телефонном справочнике большого города вполне может потребовать 25 сравнений. Для улучшения этого показателя используются деревья общего вида, в которых у узла может быть больше двух непосредственных потомков. В узле общего дерева записывается несколько значений ключа, и непосредственные потомки узла задают поддеревья, в которых содержатся (а) элементы, меньшие всех значений ключа в узле, (б) элементы, большие первого значения, но меньшие остальных значений ключа в узле, (в) элементы, большие первых двух значений ключа в узле, но меньшие остальных его значений, и т.д. На нижеследующем рисунке изображен пример общего дерева поиска. В корне этого дерева стоят ключи 6 и 10, поэтому при поиске значения, меньшего 6, мы попадаем в левую ветвь; при поиске значения между 6 и 10 мы пойдем по средней ветви, а при поиске значения, большего 10 — по правой.



Напишите алгоритм поиска по общему дереву. В ответе можно предполагать, что в каждом узле содержится два массива: Keys [3] и Links `[4]`, и что в Вашем распоряжении имеется функция `Compare(keyList,searchKey)`, возвращающая положительное целое число — номер совпавшего значения — в случае, если сравнение оказалось успешным, и отрицательное целое число, указывающее номер ветви на которую следует перейти, если ни одно из значений в узле не совпало с проверяемым. (Например, вызов функции `Compare([2,6,10]`,6) возвращает 2, поскольку це-

левое значение совпадает со вторым элементом списка, а вызов `Compare([2,6,10],7)` возвращает -3, поскольку целевое значение 7 следует искать в третьей ветви, соответствующей интервалу между 6 и 10.) Когда алгоритм будет готов, проведите анализ наихудшего и среднего случаев в предположении, что дерево полное и у каждого внутреннего узла четыре непосредственных потомка. (Быть может, Вы захотите нарисовать такое дерево.) Как повлияет на Ваш анализ в обоих случаях предположение о том, что дерево неполное или что у некоторых внутренних узлов меньше четырех непосредственных потомков?

2.3. Выборка

Иногда нам нужен элемент из списка, обладающий некоторыми специальными свойствами, а не имеющий некоторое конкретное значение. Другими словами, вместо записи с некоторым конкретным значением поля нас интересует, скажем, запись с наибольшим, наименьшим или средним значением этого поля. В более общем случае нас может интересовать запись с K-ым по величине значением поля.

Один из способов найти такую запись состоит в том, чтобы отсортировать список в порядке убывания; тогда запись с K-ым по величине значением окажется на K-ом месте. На это уйдет гораздо больше сил, чем необходимо: значения, меньшие искомого, нас, на самом деле, не интересуют. Может пригодиться следующий подход: мы находим наибольшее значение в списке и помещаем его в конец списка. Затем мы можем найти наибольшее значение в оставшейся части списка, исключая уже найденное. В результате мы получаем второе по величине значение списка, которое можно поместить на второе с конца место в списке. Повторив эту процедуру K раз, мы найдем K-ое по величине значение. В результате мы приходим к алгоритму

```
FindKthLargest(list,K,N)
list      список для просмотра
N         число элементов в списке
K         порядковый номер по величине требуемого элемента

for i=1 to K do
   largest=list[1]
   largestLocation=1
```

```
    for j=2 to N-(i-1) do
        if list[j]>largest then
            largest=list[j]
            largestLocation=j
        end if
    end for
    Swap(list[N-(i-1)],list[largestLocation])
end for
return largest
```

Какова сложность этого алгоритма? На каждом проходе мы присваиваем переменной largest значение первого элемента списка, а затем сравниваем эту переменную со всеми остальными элементами. При первом проходе мы делаем $N-1$ сравнение, на втором — на одно сравнение меньше. На K-ом проходе мы делаем $N-K$ сравнений. Поэтому для поиска K-ого по величине элемента нам потребуется

$$\sum_{i=1}^{K}(N-i) = NK - \frac{K(K-1)}{2}$$

сравнений, что по порядку величины равно $O(KN)$. Следует заметить также, что если K больше, чем $N/2$, то поиск лучше начинать с конца списка. Для значений K близких к 1 или к N этот алгоритм довольно эффективен, однако для поиска значений из середины списка есть и более эффективные алгоритмы.

Поскольку нам нужно лишь K-ое по величине значение, точное местоположение больших $K-1$ элементов нас на самом деле не интересует, нам достаточно знать, что они больше искомого. Для всякого элемента из списка весь список распадается на две части: элементы, большие взятого, и меньшие элементы. Если переупорядочить элементы в списке так, чтобы все большие шли за выбранным, а все меньшие — перед ним, и при этом выбранный элемент окажется на P-ом месте, то мы будем знать, что он P-ый по величине. Такое переупорядочивание потребует сравнения выбранного элемента со всеми остальными, т.е. $N-1$ сравнений. Если нам повезло и $P = K$, то работа закончена. Если $K < P$, то нам нужно некоторое большее значение, и проделанную процедуру можно повторить со второй частью списка. Если $K > P$, то нам нужно меньшее значение, и мы можем воспользоваться первой частью списка; при этом, однако, K следует уменьшить на число откинутых во второй

части элементов. В результате мы приходим к следующему рекурсивному алгоритму.

```
KthLargestRecursive(list,start,end,K)
list     список для просмотра
start    индекс первого рассматриваемого элемента
end      индекс последнего рассматриваемого элемента
K        порядковый номер по величине требуемого элемента

if start< end then
  Partition(list,start,end,middle)
  if middle=K then
    return list[middle]
  else
    if K<middle then
      return KthLargestRecursive(list,middle+1,end,K)
    else
      return KthLargestRecursive(list,start,middle-1,K-middle)
    end if
  end if
end if
```

Если предположить, что в среднем при разбиении список будет делиться на две более или менее равные половины, то число сравнений будет приблизительно равно $N + N/2 + N/4 + N/8 + \cdots + 1$, т.е. $2N$. Поэтому сложность алгоритма линейна и не зависит от K. Более детально мы рассмотрим процесс разбиения при обсуждении быстрой сортировки в главе 3 и там же более подробно его проанализируем.

2.3.1. Упражнения

1) В списке из пяти различных значений медиана — это третье по величине значение. В главе 3 мы увидим, что упорядочивание списка из пяти значений требует семи сравнений. Докажите, что медиану можно найти не более, чем за шесть сравнений.

2) С помощью алгоритмов этого параграфа можно найти медиану в списке, если искать $N/2$-ый по величине элемент. Покажите, что в наихудшем случае сложность поиска такого элемента равна $O(N^2)$.

2.4. Упражнение по программированию

1) Напишите на основе алгоритма последовательного поиска функцию и протестируйте ее. С помощью метода 3 из приложения Б сгенерируйте список случайных чисел в промежутке от 1 до N и проводите поиск в этом списке; здесь N может быть любым числом от 100 до 1000. В Вашей программе должен быть глобальный счетчик, который следует установить в нуль в начале программы и увеличивать на 1 непосредственно перед очередным сравнением. Затем главная программа должна вызывать функцию последовательного поиска для каждого элемента от 1 до N. Подсчитанное после этого общее число сравнений следует разделить на N; результатом будет среднее число сравнений при последовательном поиске.

2) Повторите шаг 1 для двоичного поиска в упорядоченном списке.

3) Составьте отчет, в котором сравните результаты первого и второго шагов с предсказаниями выполненного в этой главе анализа.

Глава 3.
Алгоритмы сортировки

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- читать и разрабатывать итеративные и рекурсивные алгоритмы;
- пользоваться формулами для сумм и вероятностей, описанными в главе 1;
- решать рекуррентные соотношения;
- описывать скорость и порядок роста функции.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- объяснять алгоритм сортировки вставками и проводить его анализ;
- объяснять алгоритм пузырьковой сортировки и проводить его анализ;
- объяснять алгоритм сортировки Шелла и проводить его анализ;
- объяснять алгоритм корневой сортировки и проводить его анализ;
- трассировать алгоритмы пирамидальной сортировки и FixHeap;
- проводить анализ алгоритма пирамидальной сортировки;
- объяснять алгоритм сортировки слиянием и проводить его анализ;

- объяснять алгоритм быстрой сортировки и проводить его анализ;
- объяснять алгоритм внешней многофазной сортировки слиянием и проводить его анализ.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Особенно полезно проследить за исполнением (выполнить трассировку) каждого из алгоритмов сортировки вставками, пузырьковой, Шелла, пирамидальной, слиянием и быстрой, скажем, на списках [6, 2, 4, 7, 1, 3, 8, 5] и [15, 4, 10, 8, 6, 9, 16, 1, 7, 3, 11, 14, 2, 5, 12, 13]. Проследите выполнение алгоритма корневой сортировки на трех стопках 1, 2, 3 для списка [1113, 2231, 3232, 1211, 3133, 2123, 2321, 1312, 3223, 2332, 1121, 3312]. Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

В этой главе мы рассматриваем новый класс невычислительных алгоритмов — алгоритмы сортировки. Благодаря значительной экономии времени при двоичном поиске по сравнению с последовательным поиском разработчики программного обеспечения нередко предпочитают хранить информацию в отсортированном виде, чтобы поиск нужных данных можно было вести посредством двоичного или других методов, отличных от полного перебора.

Все изучаемые нами алгоритмы будут сортировать список в порядке возрастания ключевого значения. Осознавая, что тип ключа может быть самым разным — целым, символьной строкой или еще чем-либо более сложным, — мы будем пользоваться некоей стандартной операцией сравнения пары ключей. Для простоты мы будем предполагать, что все ключевые значения в списке попарно различны: присутствие дубликатов повлияло бы на результаты нашего анализа лишь незначительно. Читатель должен понимать, что изменение упорядочения множества ключей привело бы к другому упорядочению списка. Если, например, поменять местами результаты сравнения «меньше» и «больше», то список окажется отсортированным в порядке убывания значений ключей.

Обсуждаемые в настоящей главе восемь сортировок служат лишь примерами алгоритмов сортировки, однако они демонстрируют широкий спектр возможных вариантов поведения. Первая из них, сор-

тировка вставками, сортирует список, вставляя очередной элемент в нужное место уже отсортированного списка. Пузырьковая сортировка сравнивает элементы попарно, переставляя между собой элементы тех пар, порядок в которых нарушен. Сортировка Шелла представляет собой многопроходную сортировку, при которой список разбивается на подсписки, каждый из которых сортируется отдельно, причем на каждом проходе число подсписков уменьшается, а их длина растет. При корневой сортировке список разбивается на стопки, и при каждом проходе используется отдельная часть ключа. Пирамидальная сортировка строит бинарное дерево, значение каждого узла в котором превышает значение потомков. В результате наибольший элемент списка помещается в корень, и при его удалении и выборе очередной пирамиды в корне оказывается следующий по величине элемент. Этот процесс повторяется пока все элементы не окажутся в новом, уже отсортированном, списке. Сортировка слиянием берет два уже отсортированных списка и создает, сливая их, новый отсортированный список. Быстрая сортировка представляет собой рекурсивный алгоритм, который выбирает в списке осевой элемент, а затем разбивает список на две части, состоящие из элементов соответственно меньших или больших выбранного.

Последняя сортировка предназначена для работы со списками, размер которых настолько велик, что они не помещаются целиком в памяти компьютера. Эта внешняя сортировка работает с группами записей в последовательных файлах; ее задача — минимизировать количество обращений к внешним устройствам (операция, требующая гораздо больше времени, чем сравнение). Читателю стоит принять во внимание, что даже расширенная память в современных компьютерах, в которую можно записать одновременно все записи базы данных, не решает проблемы: операции обмена в памяти все равно приводят к задержкам. Операционная система осуществляет такой обмен, переписывая сегменты виртуальной памяти на жесткие диски, хотя эти операции и не указаны в алгоритме явно.

3.1. Сортировка вставками

Основная идея сортировки вставками состоит в том, что при добавлении нового элемента в уже отсортированный список его стоит сразу вставлять в нужное место вместо того, чтобы вставлять его в произвольное место, а затем заново сортировать весь список. Сортировка

вставками считает первый элемент любого списка отсортированным списком длины 1. Двухэлементный отсортированный список создается добавлением второго элемента исходного списка в нужное место одноэлементного списка, содержащего первый элемент. Теперь можно вставить третий элемент исходного списка в отсортированный двухэлементный список. Этот процесс повторяется до тех пор, пока все элементы исходного списка не окажутся в расширяющейся отсортированной части списка.

Вот алгоритм, реализующий этот процесс:

```
InsertionSort(list,N)
list  сортируемый список элементов
N     число элементов в списке
for i=2 to N do
   newElement=list[i]
   location=i-1
   while(location >= 1) and (list[location]>newElement) do
      // сдвигаем все элементы, большие очередного
      list[location+1]=list[location]
      location=location-1
   end while
   list[location+1]=newElement
end for
```

Этот алгоритм заносит новое вставляемое значение в переменную `newElement`. Затем он освобождает место для этого нового элемента, подвигая (в цикле while) все большие элементы на одну позицию в массиве. Последняя итерация цикла переносит элемент с номером location+1 в позицию `location+2`. Это означает, что позиция location+1 освобождается для «нового» элемента.

3.1.1. Анализ наихудшего случая

Если посмотреть на внутренний цикл while, то видно, что наибольшее количество операций выполняется в случае, если вновь добавляемый элемент меньше всех элементов, содержащихся в уже отсортированной части списка. В этой ситуации выполнение цикла завершается, когда значение переменной location становится равным 0. Поэтому наибольшее количество работы алгоритм произведет, когда всякий новый элемент добавляется в начало списка. Такое возможно только, если

элементы исходного списка идут в убывающем порядке. Это один возможный наихудший случай, однако бывают и другие.

Посмотрим, как происходит обработка такого списка. Первым вставляется второй элемент списка. Он сравнивается всего с одним элементом. Второй вставляемый элемент (третий по порядку) сравнивается с двумя предыдущими, а третий вставляемый элемент — с тремя предыдущими; вообще, i-ый вставляемый элемент сравнивается с i предыдущими, и этот процесс повторяется $N - 1$ раз. Таким образом, сложность сортировки вставками в наихудшем случае равна

$$W(N) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2};$$

$$W(N) = O(N^2).$$

3.1.2. Анализ среднего случая

Анализ среднего случая разобьем на два этапа. Сначала подсчитаем среднее число сравнений, необходимое для определения положения очередного элемента. Затем среднее число всех необходимых операций можно подсчитать, воспользовавшись результатом первого шага. Начнем с подсчета среднего числа сравнений, необходимых для определения местоположения i-го элемента. Мы уже говорили о том, что добавление i-го элемента к списку требует по крайней мере одного сравнения даже, если элемент уже стоит на нужном месте.

Сколько существует возможных положений для г-ого элемента? Посмотрим на короткие списки и попробуем обобщить результат на произвольный случай. Для первого добавляемого элемента есть две возможности: он может оказаться первым или вторым в отсортированном списке из двух элементов. У второго добавляемого элемента три возможных положения: с номерами 1, 2 и 3. Оказывается, что i-ый добавляемый элемент может занимать одно из $i + 1$ возможных положений. Мы будем предполагать, что все эти возможности равновероятны.

Сколько требуется сравнений для того, чтобы добраться до каждой из г + 1 возможных позиций? Посмотрим снова на малые значения i и попробуем обобщить результат. Если четвертый добавляемый элемент попадает на позицию 5, то уже первое сравнение окажется ложным. Если правильная его позиция 4, то первое сравнение будет истинным, а

второе — ложным. При попадании в позицию 3 первое и второе сравнение будут истинными, а третье — ложным. В позиции 2 первое, второе и третье сравнение оказываются успешными, а четвертое — нет. При попадании в первую позицию истинными будут первое, второе, третье и четвертое сравнения, а ложных сравнений не будет. Это означает, что для r-го элемента, попадающего на позиции $i+1$, r, $i-1$, ..., 2, число сравнений будет равно соответственно 1, 2, 3, ..., r, а при попадании на первую позицию число сравнений равно r. Среднее число сравнений для вставки i-го элемента дается равенством

$$A_i = \frac{1}{i+1}\left(\sum_{p=1}^{i} p + i\right) = \frac{1}{i+1}\left(\frac{i(i+1)}{2} + i\right)$$

$$= \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}.$$

Мы подсчитали среднее число операций при вставке i-го элемента. Теперь необходимо просуммировать эти результаты для каждого из $N-1$ элементов списка:

$$A(N) = \sum_{i=1}^{N-1} A_i = \sum_{i=1}^{N-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right).$$

Дважды применяя формулу (1.11), получаем

$$A(N) = \sum_{i=1}^{N-1} \frac{i}{2} + \sum_{i=1}^{N-1} 1 - \sum_{i=1}^{N-1} \frac{1}{i+1}.$$

Воспользовавшись равенствами (1.9) и (1.10), имеем

$$\sum_{i=1}^{N-1} \frac{1}{i+1} = \sum_{i=2}^{N} \frac{1}{i} = \sum_{i=1}^{N} \frac{1}{i} - 1.$$

Теперь равенства (1.15), (1.14) и (1.20) дают

$$A(N) \approx \frac{(N-1)N}{2} + (N-1) - (\ln N - 1)$$

$$
\begin{aligned}
&= \frac{N^2 - N}{4} + (N - 1) - (\ln N - 1) \\
&= \frac{N^2 + 3N - 4}{4} - (\ln N - 1) \\
&\approx \frac{N^2}{4}; \\
A(N) &= O(N^2).
\end{aligned}
$$

3.1.3. Упражнения

1) Выпишите состояние списка [7, 3, 9, 4, 2, 5, 6, 1, 8] после каждого прохода алгоритма `InsertionSort`.

2) Выпишите состояние списка [3, 5, 2, 9, 8, 1, 6, 4, 7] после каждого прохода алгоритма `InsertionSort`.

3) В разделе 3.1.1 показано, что наихудший случай отвечает убывающему списку. Так, список [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] дает максимально возможное число сравнений на десяти элементах, т.е. 45. Найдите еще один список из десяти элементов с наихудшим поведением. Что можно сказать о входном списке с наихудшим поведением в случае произвольной длины?

4) При внимательном изучении алгоритма `InsertionSort` видно, что вставка нового элемента основана на алгоритме последовательного поиска. В главе 2 мы показали, что двоичный поиск гораздо эффективнее. Модифицируйте сортировку вставками, применив двоичный поиск места вставки нового элемента. Обратите внимание на то, что при уникальных значениях ключей стандартный двоичный поиск всегда заканчивается неудачей. Замените стандартный поиск функцией, возвращающей значение искомой позиции.

   а) Проведите анализ наихудшего случая в новом алгоритме сортировки вставками.

   б) Проведите анализ среднего случая в новом алгоритме сортировки вставками.

3.2. Пузырьковая сортировка

Перейдем теперь к пузырьковой сортировке. Ее основной принцип состоит в выталкивании маленьких значений на вершину списка в то время, как большие значения опускаются вниз. У пузырьковой сортировки есть много различных вариантов. В этом параграфе мы опишем один из них, а еще несколько будут рассматриваться в упражнениях.

Алгоритм пузырьковой сортировки совершает несколько проходов по списку. При каждом проходе происходит сравнение соседних элементов. Если порядок соседних элементов неправильный, то они меняются местами. Каждый проход начинается с начала списка. Сперва сравниваются первый и второй элементы, затем второй и третий, потом третий и четвертый и так далее; элементы с неправильным порядком в паре переставляются. При обнаружении на первом проходе наибольшего элемента списка он будет переставляться со всеми последующими элементами пока не дойдет до конца списка. Поэтому при втором проходе нет необходимости производить сравнение с последним элементом. При втором проходе второй по величине элемент списка опустится во вторую позицию с конца. При продолжении процесса на каждом проходе по крайней мере одно из следующих по величине значений встает на свое место. При этом меньшие значения тоже собираются наверху. Если при каком-то проходе не произошло ни одной перестановки элементов, то все они стоят в нужном порядке, и исполнение алгоритма можно прекратить. Стоит заметить, что при каждом проходе ближе к своему месту продвигается сразу несколько элементов, хотя гарантированно занимает окончательное положение лишь один.

Вот алгоритм, реализующий этот вариант пузырьковой сортировки:

```
BubbleSort(list,N)
list  сортируемый список элементов
N     число элементов в списке

numberOfPairs=N
swappedElements=true
while swappedElements do
   numberOfPairs=numberOfPairs-1
   swappedElements=false
   for i=1 to numberOfPairs do
      if list[i] > list[i+1] then
         Swap(list[i],list[i+1])
```

```
            swappedElements=true
        end if
    end for
end while
```

3.2.1. Анализ наилучшего случая

Мы вкратце рассмотрим наилучший случай, чтобы предупредить неверную интерпретацию флага `swappedElements`. Рассмотрим, в каком случае объем выполняемой работы минимален. При первом проходе цикл `for` должен быть выполнен полностью, и поэтому алгоритму потребуется по меньшей мере $N - 1$ сравнение. Следует рассмотреть две возможности: при первом проходе была по крайней мере одна перестановка, и перестановок не было. В первом случае перестановка приводит к изменению значения флага swappedElements на true, а значит цикл while будет выполнен повторно, что потребует еще $N - 2$ сравнений. Во втором случае флаг swappedElements сохранит значение false, и выполнение алгоритма прекратится.

Поэтому в лучшем случае будет выполнено $N-1$ сравнений, что происходит при отсутствии перестановок на первом проходе. Это означает, что наилучший набор данных представляет собой список элементов, уже идущих в требуемом порядке.

3.2.2. Анализ наихудшего случая

Поскольку в наилучшем возможном списке элементы идут в требуемом порядке, стоит посмотреть, не дает ли обратный порядок элементов наихудший случай. Если наибольший элемент стоит первым, то он будет переставляться со всеми остальными элементами вплоть до конца списка. Перед первым проходом второй по величине элемент занимает вторую позицию, однако в результате первого сравнения и первой перестановки он переставляется на первое место. В начале второго прохода на первой позиции уже находится второй по величине элемент, и он переставляется со всеми остальными элементами вплоть до предпоследнего. Этот процесс повторяется для всех остальных элементов, поэтому цикл `for` будет повторен $N-1$ раз. Поэтому обратный порядок входных данных приводит к наихудшему случаю[1].

[1] В этом случае максимальным оказывается как число сравнений, так и число перестановок. Однако поскольку нас интересует только количество сравнений, суще-

Сколько сравнений выполняется в наихудшем случае? На первом проходе будет выполнено $N - 1$ сравнений соседних значений, на втором — $N - 2$ сравнений. Дальнейшее исследование показывает, что при каждом очередном проходе число сравнений уменьшается на 1. Поэтому сложность в наихудшем случае дается формулой

$$W(N) = \sum_{i=N-1}^{1} i = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2}$$

$$\approx \frac{1}{2}N^2 = O(N^2).$$

3.2.3. Анализ среднего случая

Мы уже говорили, что в наихудшем случае цикл `for` будет повторяться $N - 1$ раз. В среднем случае мы будем предполагать, что появление прохода без перестановки элементов равновероятно в любой из этих моментов. Посмотрим, сколько сравнений будет произведено в каждом из случаев. При остановке после первого прохода число сравнений равно $N - 1$. После двух проходов число сравнений равно $N - 1 + N - 2$. Обозначим через $C(i)$ число сравнений, выполненных на первых $г$ проходах. Поскольку алгоритм останавливает свою работу, если ни одной перестановки не произошло, при анализе среднего случая нужно рассмотреть все эти возможности. В результате приходим к равенству

$$A(N) = \frac{1}{N-1} \sum_{i=1}^{N-1} C(i),$$

где, как и раньше, $C(i)$ равно числу сравнений на протяжении первых i проходов цикла `for`. Сколько же на это требуется сравнений? Значение $C(г)$ дается равенством

$$C(i) = \sum_{j=N-i}^{г} j = \sum_{j=i}^{N-1} j = \sum_{i=1}^{N-1} j - \sum_{j=1}^{i-1} j$$

ствуют и другие наихудшие наборы данных. Читатель без труда покажет, например, что любой список, минимальный элемент в котором стоит последним, оказывается наихудшим. Есть ли другие возможности?

(последнее равенство вытекает из равенства (1.10)), или

$$C(i) = \frac{(N-1)N}{2} - \frac{(i-1)i}{2} = \frac{N^2 - N - i^2 + i}{2}.$$

Подставляя последнее равенство в выражение для $A(N)$, мы получаем

$$A(N) = \frac{1}{N-1} \sum_{i=1}^{N-1} \frac{N^2 - N - i^2 + i}{2}.$$

Поскольку N не зависит от г, равенства (1.11) и (1.14) дают

$$A(N) = \frac{1}{N-1} \left((N-1)\frac{N^2 - N}{2} + \sum_{i=1}^{N-1} \frac{-i^2 + i}{2} \right).$$

Воспользовавшись вновь равенством (1.11) и элементарными алгебраическими выкладками, мы получаем

$$A(N) = \frac{N^2 - N}{2} + \frac{1}{2(N-1)} \left(\sum_{i=1}^{N-1} (-i^2) + \sum_{i=1}^{N-1} i \right).$$

Окончательно, равенства (1.15) и (1.16) дают

$$A(N) \approx \frac{N^2 - N}{2} + \frac{1}{2(N-1)} \left(-\frac{(N-1)N(2N-1)}{6} + \frac{(N-1)N}{2} \right)$$

$$= \frac{N^2 - N}{2} - \frac{N(2N-1)}{12} + \frac{N}{4}$$

$$= \frac{6N^2 - 6N - 2N^2 + N + 3N}{12}$$

$$= \frac{4N^2 - 2N}{12}$$

$$\approx \frac{1}{3}N^2 = O(N^2).$$

3.2.4. Упражнения

1) Выпишите результаты всех проходов алгоритма Bubble Sort на списке [7, 3, 9, 4, 2, 5, 6, 1, 8].

2) Выпишите результаты всех проходов алгоритма `BubbleSort` на списке [3, 5, 2, 9, 8, 1, 6, 4, 7].

3) Еще в одном варианте пузырьковой сортировки запоминается информация о том, где произошел последний обмен элементов, и при следующем проходе алгоритм не заходит за это место. Если последними поменялись i-ый и $i + 1$-ый элементы, то при следующем проходе алгоритм не сравнивает элементы за i-м.

   а) Напишите этот вариант пузырьковой сортировки.

   б) Напишите краткое доказательство того, почему этот вариант пузырьковой сортировки действительно работает.

   в) Меняет ли этот вариант пузырьковой сортировки анализ наихудшего случая?

   г) Анализ среднего случая новой пузырьковой сортировки отличается от исходного. Объясните подробно, что следует учитывать при анализе среднего случая в новом варианте.

4) Еще в одном варианте пузырьковой сортировки нечетные и четные проходы выполняются в противоположных направлениях: нечетные проходы в том же направлении, что и в исходном варианте, а четные — от конца массива к его началу. При нечетных проходах большие элементы сдвигаются к концу массива, а при четных проходах — ме́ньшие элементы двигаются к его началу.

   а) Напишите этот вариант пузырьковой сортировки.

   б) Напишите краткое доказательство того, почему этот вариант пузырьковой сортировки действительно работает.

   в) Меняет ли этот вариант пузырьковой сортировки анализ наихудшего случая? Обоснуйте свой ответ.

   г) Анализ среднего случая новой пузырьковой сортировки отличается от исходного. Объясните подробно, что следует учитывать при анализе среднего случая в новом варианте.

5) В третьем варианте пузырьковой сортировки идеи первого и второго измененных вариантов совмещены. Этот алгоритм двигается по массиву поочередно вперед и назад и дополнительно выстраивает начало и конец списка в зависимости от места последнего изменения.

   а) Напишите этот третий вариант пузырьковой сортировки.

   б) Напишите краткое доказательство того, почему этот вариант пузырьковой сортировки действительно работает.

   в) Меняет ли этот вариант пузырьковой сортировки анализ наихудшего случая? Обоснуйте свой ответ.

   г) Анализ среднего случая третьего варианта пузырьковой сортировки отличается от исходного. Объясните подробно, что следует учитывать при анализе среднего случая в новом варианте.

6) Докажите строго, что в результате первого прохода цикла в алгоритме Bubble Sort наибольший элемент действительно оказывается на своем месте.

7) Докажите строго, что если на очередном проходе в алгоритме `BubbleSort` обменов не произошло, то элементы списка идут в правильном порядке.

3.3. Сортировка Шелла

Сортировку Шелла придумал Дональд Л. Шелл. Ее необычность состоит в том, что она рассматривает весь список как совокупность перемешанных подсписков. На первом шаге эти подсписки представляют собой просто пары элементов. На втором шаге в каждой группе по четыре элемента. При повторении процесса число элементов в каждом подсписке увеличивается, а число подсписков, соответственно, падает. На рис. 3.1 изображены подсписки, которыми можно пользоваться при сортировке списка из 16 элементов.

На рис. 3.1 (а) изображены восемь подсписков, по два элемента в каждом, в которых первый подсписок содержит первый и девятый элементы, второй подсписок — второй и десятый элементы, и так далее. На рис. 3.1 (б) мы видим уже четыре подсписка по четыре элемента в

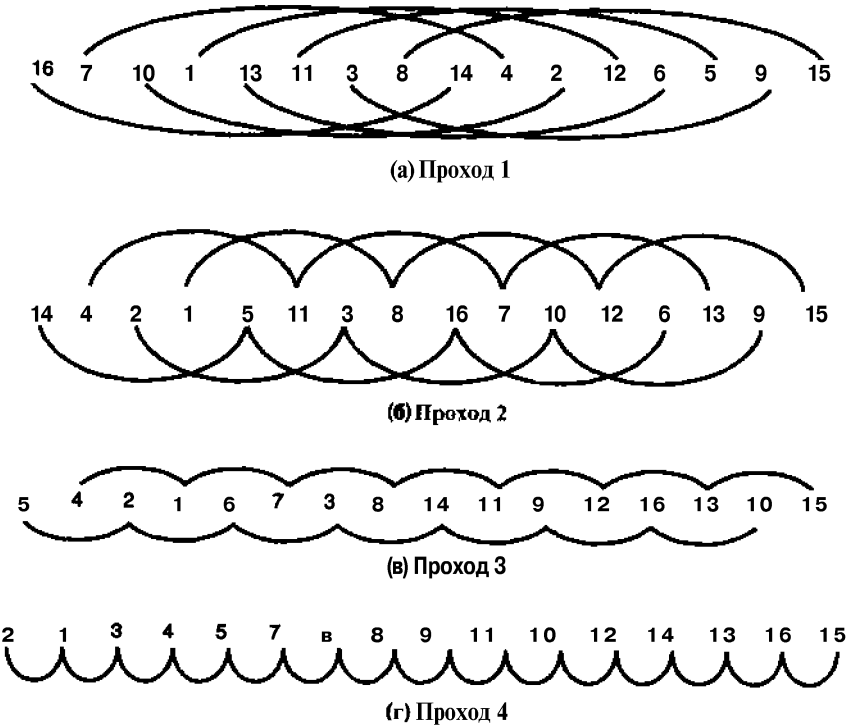


Рис. **3.1.** Четыре прохода сортировки Шелла

каждом. Первый подсписок на этот раз содержит первый, пятый, девятый и тринадцатый элементы. Второй подсписок состоит из второго, шестого, десятого и четырнадцатого элементов. На рис. 3.1 (в) показаны два подсписка, состоящие из элементов с нечетными и четными номерами соответственно. На рис. 3.1 (г) мы вновь возвращаемся к одному списку.

Сортировка подсписков выполняется путем однократного применения сортировки вставками из § 3.1. В результате мы получаем следующий алгоритм:

```
Shellsort(list,N)
list   сортируемый список элементов
N      число элементов в списке

passes = [log_2 N]
```

```
while (passes>=1) do
   increment=2^passes-1
   for start=1 to increment do
      InsertionSort(list,N,start,increment)
   end for
   passes=passes-1
end while
```

Переменная increment содержит величину шага между номерами элементов подсписка. (На рис. 3.1 шаг принимает значения 8, 4, 2 и 1.) В алгоритме мы начинаем с шага, на 1 меньшего наибольшей степени двойки, меньшей, чем длина элемента списка. Поэтому для списка из 1000 элементов первое значение шага равно 511. Значение шага также равно числу подсписков. Элементы первого подсписка имеют номера 1 и `1+increment`; первым элементом последнего подсписка служит элемент с номером increment.

При последнем исполнении цикла while значение переменной passes будет равно 1, поэтому при последнем вызове функции InsertionSort значение переменной increment будет равно 1.

Анализ этого алгоритма основывается на анализе внутреннего алгоритма InsertionSort. Прежде, чем перейти к анализу алгоритма ShellSort, напомним, что в § 3.1 мы подсчитали, что в наихудшем случае сортировка вставками списка из N элементов требует $(N^2 - N)/2$ операций, а в среднем случае она требует $N^2/4$ элементов.

3.3.1. Анализ алгоритма

Начнем анализ с подсчета числа вызовов функции InsertionSort и числа элементов в списке при каждом из этих вызовов. Обратимся к частному случаю списка длины 15. При первом проходе значение переменной increment равно 7, поэтому будет выполнено семь вызовов на списках длины 2. При втором проходе значение переменной increment равно 3, поэтому произойдет три вызова на списках длины 5. На третьем, и последнем, проходе мы делаем один вызов на списке длины 15. Из предыдущих результатов мы знаем, что на списке из двух элементов алгоритм InsertionSort делает одно сравнение в наихудшем случае. Число сравнений в наихудшем случае на списке из 5 элементов равно 10. Число сравнений в наихудшем случае на списке из 15 элементов равно 105. Сложив все эти числа, мы получаем всего 142 сравнения ($7 \cdot 1 + 3 \cdot 10 + 1 \cdot 105$). Хорошую ли оценку мы получили, однако? Если

вернуться к анализу из раздела 3.1.1, то можно вспомнить, что в наихудшем для сортировки вставками случае каждый новый элемент добавляется в начало списка. При последнем проходе алгоритма ShellSort мы знаем, что этот наихудший случай не может осуществиться ввиду того, что на более ранних этапах также осуществлялась сортировка. Быть может, другой подход поможет подсчитать объем оставшейся работы?

При анализе алгоритмов сортировки мы будем иногда подсчитывать количество инверсий в списке. Инверсия — это пара элементов списка, идущих в неправильном порядке. Например, в списке $[3,2,4,1]$ четыре инверсии, а именно, $(3,2)$, $(3,1)$, $(2,1)$ и $(4,1)$. Больше всего инверсий, как нетрудно видеть, в списке, элементы которого идут в обратном порядке; их число равно $(N^2 - N)/2$.

Один из способов анализа алгоритма сортировки состоит в том, чтобы подсчитать число инверсий, отличающее начальное состояние списка от отсортированного списка. Каждая перестановка элементов в алгоритме уменьшает количество инверсий по крайней мере на одну. Если, например, пузырьковая сортировка обнаружила в результате сравнения пару соседних элементов, идущих в неправильном порядке, то их перестановка уменьшает количество инверсий в точности на единицу. То же самое справедливо и для сортировки вставками, поскольку сдвиг большего элемента на одну позицию вверх приводит к уничтожению инверсии, образованной сдвигаемым и вставляемым элементами. Поэтому в пузырьковой сортировке и сортировке вставками (обе сложности $O(N^2)$) всякое сравнение может привести к удалению одной (или менее) инверсии.

В основе сортировки Шелла лежит сортировка вставками, поэтому может показаться, что и для нее справедливо то же самое утверждение. Однако, если принять во внимание, что в сортировке Шелла упорядочиваются перемешанные подсписки, вызванный отдельным сравнением обмен элементов местами может уменьшить число инверсий более, чем на одну. При первом проходе на рис. 3.1 сравнивались элементы 16 и 14; поскольку их порядок оказался неправильным, они были переставлены. Переместив элемент 16 с первого места на девятое, мы избавились от семи инверсий, содержащих этот элемент (вторые элементы этих инверсий расположены в позициях со второй по восьмую). Анализ сортировки Шелла вызывает сложности, поскольку та же самая перестановка элементов привела к появлению новых семи инверсий, содержащих элемент 14. Поэтому она уменьшила общее число инверсий лишь на единицу. Если посмотреть на перестановку элементов 7 и 4,

то результат выглядит так же. Однако в целом ситуация улучшается. На первом проходе после восьми сравнений оказались удаленными 36 инверсий. На втором проходе выполнено 19 сравнений и удалено 24 инверсии. Наконец, на последнем шаге было сделано 19 сравнений и удалено 10 последних инверсий. Общее число сравнений равно 62. Даже если бы мы заменили худший случай при анализе вызовов сортировки вставками средним, мы все равно насчитали бы 152 сравнения.

Полный анализ сортировки Шелла чрезвычайно сложен, и мы не собираемся на нем останавливаться. Было доказано, что сложность этого алгоритма в наихудшем случае при выбранных нами значениях шага равна $O(N^{3/2})$. Полный анализ сортировки Шелла и влияния на сложность последовательности шагов (которое мы обсуждаем в следующем разделе) можно найти в третьем томе книги Д. Кнута *Искусство программирования*, М., Мир, 1978.

3.3.2. Влияние шага на эффективность

Выбор последовательности шагов может оказать решающее влияние на сложность сортировки Шелла, однако попытки поиска оптимальной последовательности шагов не привели к искомому результату. Изучалось несколько возможных вариантов; здесь представлены результаты этого изучения.

Если проходов всего два, то при шаге порядка $1.72\sqrt[3]{N}$ на первом проходе и 1 на втором проходе получим сортировку сложности $O(N^{5/3})$.

Другой возможный набор шагов имеет вид $hj = (3^j - 1)/2$ для всех тех j, при которых $hj < N$. Эти значения hj удовлетворяют равенствам $h_{j+1} = 3h_j + 1$ и $hj = 1$, поэтому определив наибольшее из них, мы можем подсчитать оставшиеся по формуле $hj = (h_{j+1} - 1)/3$. Такая последовательность шагов приводит к алгоритму сложности $O(N^{3/2})$.

Еще один вариант предусматривает вычисление всех значений 2^i3^j, меньших длины списка (при произвольных целых г и j); вычисленные значения используются, в порядке убывания, в качестве значений шагов. Например, при $N = 40$ имеется следующая последовательность шагов: 36 (2^23^2), 32 (2^53^0), 27 (2^03^3), 24 (2^33^1), 18 (2^13^2), 16 (2^43^0), 12 (2^23^1), 9 (2^03^2), 8 (2^33^0), 6 (2^13^1), 4 (2^23^0), 3 (2^03^1), 2 (2^13^0), 1 (2^03^0). С помощью такой последовательности шагов сложность сортировки Шелла понижается до $O(N(\log N))$. Следует заметить, что большое число проходов значительно увеличивает накладные расходы, поэтому при небольших размерах списка применение этой последовательности не слишком эффективно.

Сортировка Шелла — это один общий вид сортировки, однако выбор параметров может существенно повлиять на ее эффективность.

3.3.3. Упражнения

1) Выпишите результаты всех проходов алгоритма ShellSort с шагами 7, 5, 3 и 1 на списке [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]. Сколько сравнений было выполнено?

2) Выпишите результаты всех проходов алгоритма ShellSort с шагами 8, 4, 2 и 1 на списке [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]. Сколько сравнений было выполнено?

3) Выпишите результаты всех проходов алгоритма ShellSort с шагами 5, 2 и 1 на списке [7, 3, 9, 4, 2, 5, 6, 1, 8]. Сколько сравнений было выполнено?

4) Выпишите результаты всех проходов алгоритма ShellSort с шагами 5, 2 и 1 на списке [3, 5, 2, 9, 8, 1, 6, 4, 7]. Сколько сравнений было выполнено?

5) Напишите использованный в этом разделе новый вариант алгоритма InsertSort.

6) В этом разделе мы смотрим на сортировку как на удаление инверсий. Какое максимальное число инверсий в списке из N элементов можно удалить, переставив два несоседних элемента списка? Приведите пример такой перестановки в списке длины 10.

3.4. Корневая сортировка

При корневой сортировке упорядочивание списка происходит без непосредственного сравнения ключевых значений между собой. При этом создается набор «стопок», а элементы распределяются по стопкам в зависимости от значений ключей. Собрав значения обратно и повторив всю процедуру для последовательных частей ключа, мы получаем отсортированный список. Чтобы такая процедура работала, распределение по стопкам и последующую сборку следует выполнять очень аккуратно.

Подобная процедура используется при ручной сортировке карт. В некоторых библиотеках в докомпьютерные времена при выдаче книги заполнялась карта выдачи. Карты выдачи были занумерованы, а сбоку на них проделывался ряд выемок — в зависимости от номера карты. При возвращении книги карта выдачи удалялась, и ее клали в стопку. Затем вся стопка прокалывалась длинной иглой на месте первой выемки, и иглу поднимали. Карты с удаленной выемкой оставались на столе, а остальные были наколоты на иглу. Затем две получившиеся стопки переставлялись так, что карточки на игле шли за карточками с отверстиями. Затем игла втыкалась в месте второй выемки и весь процесс повторялся. После прокалывания по всем позициям карты оказывались в порядке возрастания номеров.

Эта процедура ручной обработки разделяет карты по значению младшего разряда номера на первом шаге и старшего разряда на последнем. Компьютеризованный вариант этого алгоритма использует 10 стопок:

```
RadixSort(list,N)
list   сортируемый список элементов
N      число элементов в списке

shift=1
for loop=1 to keySize do
   for entry=1 to N do
      bucketNumber=(list[entry].key/shift) mod 10
      Append(bucket[bucketNumber],list[entry])
   end for entry
   list=CombineBuckets()
   shift=shift*10
end for loop
```

Начнем с обсуждения этого алгоритма. При вычислении значения переменной bucketNumber из ключа вытаскивается одна цифра. При делении на **shift** ключевое значение сдвигается на несколько позиций вправо, а последующее применение операции mod оставляет лишь цифру единиц полученного числа. При первом проходе с величиной сдвига 1 деление не меняет числа, а результатом операции mod служит цифра единиц ключа. При втором проходе значение переменной **shift** будет уже равно 10, поэтому целочисленное деление и последующее применение операции mod дают цифру десятков. При очередном проходе будет получена следующая цифра числа.

Функция `CombineBuckets` вновь сводит все стопки от bucket [0] до bucket [9] в один список. Этот переформированный список служит основой для следующего прохода. Поскольку переформирование стопок идет в определенном порядке, а числа добавляются к концу каждой стопки, ключевые значения постепенно оказываются отсортированными.

Исходный список

310 213 023 130 013 301 222 032 201 111 323 002 330 102 231 120

Номер стопки	Содержимое			
0	310	130	330	120
1	301	201	111	231
2	222	032	002	102
3	213	023	013	323

(а) Первый проход, разряд единиц

Список, полученный в результате первого прохода

310 130 330 120 301 201 111 231 222 032 002 102 213 023 013 323

Номер стопки	Содержимое			
0	301	201	002	102
1	310	111	213	013
2	120	222	023	323
3	130	330	231	032

(б) Второй проход, разряд десятков

Список, полученный в результате второго прохода

301 201 002 102 310 111 213 013 120 222 023 323 130 330 231 032

Номер стопки	Содержимое			
0	002	013	023	032
1	102	111	120	130
2	201	213	222	231
3	301	310	323	330

(в) Третий проход, разряд сотен

Рис. 3.2. Три прохода корневой сортировки

На рис. 3.2 показаны три прохода на списке трехзначных чисел. Для упрощения примера в ключах используются только цифры от 0 до 3, поэтому достаточно четырех стопок.

При взгляде на рис. 3.2 (в) становится ясно, что если вновь собрать стопки по порядку, то получится отсортированный список.

3.4.1. Анализ

Анализ корневой сортировки требует учета дополнительной информации, не только числа операций. Способ реализации алгоритма оказывает существенное влияние на эффективность. Нас будет интересовать эффективность алгоритма как по времени, так и по памяти.

Каждый ключ просматривается по одному разу на каждый разряд, присутствующий в самом длинном ключе. Поэтому если в самом длинном ключе M цифр, а число ключей равно N, то сложность корневой сортировки имеет порядок $O(MN)$. Однако длина ключа невелика по сравнению с числом ключей. Например, при ключе из шести цифр число различных записей может быть равно миллиону. Обсуждение в § 1.4 показывает теперь, что длина ключа не играет роли и алгоритм имеет линейную по длине сложность $O(N)$. Это очень эффективная сортировка, поэтому возникает вопрос, зачем вообще нужны какие-либо другие методы.

Ключевым препятствием в реализации корневой сортировки служит ее неэффективность по памяти. В предыдущих алгоритмах сортировки дополнительная память нужна была разве что для обмена двух записей, и размер этой памяти равняется длине записи. Теперь же дополнительной памяти нужно гораздо больше. Если стопки реализовывать массивами, то эти массивы должны быть чрезвычайно велики. На самом деле, величина каждого из них должна совпадать с длиной всего списка, поскольку у нас нет оснований полагать, что значения ключей распределены по стопкам равномерно, как на рис. 3.2. Вероятность того, что во всех стопках окажется одинаковое число записей, совпадает с вероятностью того, что все записи попадут в одну стопку. И то, и другое может произойти. Поэтому реализация на массивах приведет к выделению места дополнительно под $10N$ записей для числовых ключей, $26N$ записей для буквенных ключей, причем это место еще увеличивается, если в ключ могут входить произвольные символы или в буквенных ключах учитывается регистр. Кроме того, при использовании массивов нам потребуется время на копирование записей в стопки на этапе формирования стопок и на обратное их копирование в список на этапе

сборки списка. Это означает, что каждая запись «перемещается» $2M$ раз. Для длинных записей такое копирование требует значительного времени.

Другой подход состоит в объединении записей в список со ссылками. Тогда, как помещение записи в стопку, так и ее возвращение в список требует лишь изменения ссылки. Требуемая дополнительная память по-прежнему остается значительной, поскольку на каждую ссылку в стандартной реализации уходит от двух до четырех байтов, а значит требуемая дополнительная память составит $2N$ или $4N$ байтов.

3.4.2. Упражнения

1) С помощью алгоритма RadixSort отсортируйте список [1405, 975, 23, 9803, 4835, 2082, 7368, 573, 804, 746, 4703, 1421, 4273, 1208, 521, 2050]. Выпишите стопки при каждом проходе и состояние списка после каждой сборки списка.

2) С помощью алгоритма RadixSort отсортируйте список [117, 383, 4929, 144, 462, 1365, 9726, 241, 1498, 82, 1234, 8427, 237, 2349, 127, 462]. Выпишите стопки при каждом проходе и состояние списка после каждой сборки списка.

3) В корневой сортировке ключ можно также рассматривать как последовательность битов. Так, можно смотреть на целочисленный 4-байтовый ключ как на последовательность из 32 битов, а на 15-символьный ключ (состоящий из 15 байтов) как на последовательность из 120 битов. Затем эти последовательности битов разбиваются на части; разбиение задает число проходов и число стопок. Так, например, 120-битовый ключ можно разбить на 12 частей по 10 битов в каждой или на 10 частей по 12 битов в каждой, или на 5 частей по 24 бита в каждой.

   а) Предположим, что ключ представляет собой число в интервале от 0 до 2^{64}; выберите два возможных (поменьше и побольше) значения числа битов, которые используются на каждом проходе, и укажите соответствующее число стопок и число проходов.

   б) Предположим, что ключ представляет собой 40-байтовую символьную строку; выберите два возможных (поменьше и побольше) значения числа битов, которые используются на ка-

ждом проходе, и укажите соответствующее число стопок и число проходов.

в) Можете ли Вы на основе своих ответов на вопросы а) и б) дать общие рекомендации о том, как выбирать разбиения ключа и число проходов?

3.5. Пирамидальная сортировка

В основе пирамидальной сортировки лежит специальный тип бинарного дерева, называемый пирамидой; значение корня в любом поддереве такого дерева больше, чем значение каждого из его потомков. Непосредственные потомки каждого узла не упорядочены, поэтому иногда левый непосредственный потомок оказывается больше правого, а иногда наоборот. Пирамида представляет собой полное дерево, в котором заполнение нового уровня начинается только после того, как предыдущий уровень заполнен целиком, а все узлы на одном уровне заполняются слева направо.

Сортировка начинается с построения пирамиды. При этом максимальный элемент списка оказывается в вершине дерева: ведь потомки вершины обязательно должны быть меньше. Затем корень записывается последним элементом списка, а пирамида с удаленным максимальным элементом переформировывается. В результате в корне оказывается второй по величине элемент, он копируется в список, и процедура повторяется пока все элементы не окажутся возвращенными в список. Вот как выглядит соответствующий алгоритм:

```
сконструировать пирамиду
for i=1 to N do
    скопировать корень пирамиды в список
    переформировать пирамиду
end for
```

Некоторые детали в этом алгоритме следует уточнить. Сначала мы должны определить, из чего состоят процедуры конструирования и переформирования пирамиды: это влияет на эффективность алгоритма.

Нас интересует реализация алгоритма. Накладные расходы на создание бинарного дерева растут с ростом списка. Можно, однако, воспользоваться местом, занятым списком. Можно «перестроить» список

в пирамиду, если учесть, что у каждого внутреннего узла два непосредственных потомка, за исключением, быть может, одного узла на предпоследнем уровне. Следующее отображение позволяет сформировать требуемую пирамиду. Непосредственных потомков элемента списка с номером r будем записывать в позиции $2i$ и $2i + 1$. Заметим, что в результате положения всех потомков будут различны. Мы знаем, что если $2i > N$, то узел r оказывается листом, а если $2i = N$, то у узла r ровно один потомок. На рис. 3.3 изображена пирамида и ее списочное представление.

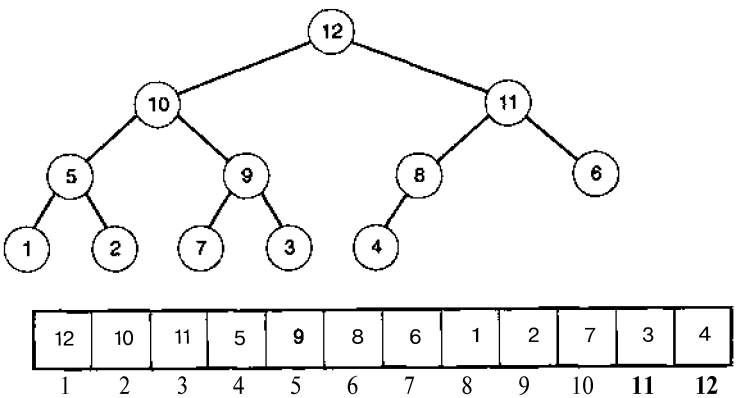


Рис. 3.3. Пирамида и ее списочное представление

Переформирование пирамиды

При копировании наибольшего элемента из корня в список корневой узел остается свободным. Мы знаем, что в него должен попасть больший из двух непосредственных потомков, и мы должны рассмотреть, кто встает на его место, и т.д. При этом пирамида должна оставаться настолько близкой к полному дереву, насколько это возможно. Переформирование пирамиды мы начинаем с самого правого элемента на нижнем ее уровне. В результате узлы с нижней части дерева будут убираться равномерно. Если за этим не следить, и все большие значения оказываются в одной части пирамиды, то пирамида разбалансируется и эффективность алгоритма падает. Вот что получается в результате:

```
FixHeap(list,root,key,bound)
list  сортируемый список/пирамида
root  номер корня пирамиды
```

```
key  ключевое значение, вставляемое в пирамиду
bound правая граница (номер) в пирамиде

vacant = root
while 2*vacant <= bound do
   largerChild = 2*vacant

   // поиск наибольшего из двух непосредственных потомков
   if (largerChild<bound) and (list[largerChild+1]>
                  list[largerChild+1]) then
      largerChild=largerChild+1
   end if

   // находится ли ключ выше текущего потомка?
   if key>list[largerChild] then
      // да, цикл завершается
      break
   else
      // нет, большего непосредственного потомка
      //следует поднять
      list[vacant]=list[largerChild]
      vacant=largerChild
   end if
end while
list[vacant]=key
```

При взгляде на параметры этого алгоритма у Вас мог возникнуть вопрос, зачем нужна переменная root. Действительно, поскольку процедура не рекурсивна, корень пирамиды всегда должен быть первым элементом. Мы увидим, однако, что этот дополнительный параметр позволит нам строить пирамиду снизу вверх. Значение правой границы введено потому, что при переписывании элементов из пирамиды в список пирамида сжимается.

Построение пирамиды

Наш подход к функции FixHeap позволяет сформировать начальное состояние пирамиды. Любые два значения можно считать листьями свободного узла. Поскольку все элементы являются листьями, нет необходимости делать что-либо со второй половиной списка. Мы можем

просто делать из листьев маленькие пирамиды, а затем постепенно собирать их вместе, пока все значения не попадут в список. Следующий цикл позволяет реализовать эту процедуру:

```
for i=N/2 down to 1 do
   FixHeap(list,i,list[i],N)
end for
```

Окончательный алгоритм

Сложив вместе все написанные куски и добавив необходимые операции по перенесению элементов из пирамиды в список, мы получаем окончательный алгоритм

```
for i=N/2 down to 1 do
   FixHeap(list,i,list[i],N)
end for
for i=N down to 2 do
   max=list[1]
   FixHeap(list,1,list[i] ,i-1)
   list[i]=max
end for
```

3.5.1. Анализ наихудшего случая

Начнем с анализа процедуры FixHeap, поскольку остальная часть алгоритма строится на ее основе. На каждом уровне пирамиды алгоритм сравнивает двух непосредственных потомков, а также большего из непосредственных потомков и ключ. Это означает, что число сравнений для пирамиды глубины D не превышает $2D$.[2]

На стадии конструирования пирамиды мы сначала вызываем процедуру FixHeap для каждого узла второго уровня с конца, т.е. построенные пирамиды имеют глубину 1. Затем она вызывается для каждого узла третьего уровня снизу, т.е. строятся пирамиды глубины 2. При последнем проходе, на уровне корня, будет сформирована пирамида глубины $\lfloor \log_2 N \rfloor$. Нам осталось только подсчитать, для какого количества узлов будет вызываться процедура FixHeap при каждом

[2] **Глубина** узла на единицу меньше его уровня. Максимальная глубина пирамиды из четырех уровней равна 3. Корень находится на глубине 0, его непосредственные потомки — на глубине 1, их сыновья — на глубине 2 и т.д.

проходе. На корневом уровне узел один, а на втором уровне у него два сына. Число узлов на следующем уровне не превосходит четырех, а на следующем — восьми. Сводя эти результаты воедино, мы приходим к следующим формулам

$$W_{\text{Construction}}(N) = \sum_{i=0}^{D-1} 2(D - i)2^i = 2D \sum_{i=0}^{D-1} 2^i - 2 \sum_{i=0}^{D-1} i2^i.$$

Равенства (1.17) и (1.19) дают

$$\begin{aligned} W_{\text{Construction}}(N) &= 2D(2^D - 1) - 2[(D - 2)2^D + 2] \\ &= 2^{D+1}D - 2D - 2^{D+1}D + 2^{D+2} - 4 \\ &= 2^{D+2} - 2D - 4. \end{aligned}$$

Подставив теперь $D = \log_2 N$, получим

$$\begin{aligned} W_{\text{Construction}}(N) &= 4 \cdot 2^{\log_2 N} - 2\log_2 N - 4 \\ &= 4N - 2\log_2 N - 4 = O(N). \end{aligned}$$

Поэтому этап построения пирамиды имеет линейную сложность по числу элементов списка.

Рассмотрим теперь основной цикл алгоритма. В этом цикле мы удаляем из пирамиды один элемент, а затем вызываем процедуру FixHeap. Цикл выполняется до тех пор, пока в пирамиде не останется один элемент. При каждом проходе число элементов уменьшается на 1, а как меняется глубина пирамиды? Мы уже говорили о том, что глубина всей пирамиды равна $\lfloor \log_2 N \rfloor$, поэтому если в пирамиде осталось K узлов, то ее глубина равна $\lfloor \log_2 K \rfloor$, а число сравнений в два раза больше. Это означает, что в наихудшем случае в цикле выполняется

$$W_{\text{loop}} = \sum_{k=1}^{N-1} 2\lfloor \log_2 k \rfloor = 2 \sum_{k=1}^{N-1} \lfloor \log_2 k \rfloor$$

операций. Однако для такой суммы у нас нет стандартного выражения.

Посмотрим, как можно преодолеть это препятствие. При $k = 1$ имеем $\lfloor \log_2 k \rfloor = 0$. При k равном 2 или 3 имеем $\lfloor \log_2 k \rfloor = 1$. Далее, $\lfloor \log_2 k \rfloor = 2$ при k от 4 до 7 и $\lfloor \log_2 k \rfloor = 3$ при k от 8 до 15. Мы замечаем, что в каждом из случаев результат j принимается на 2^j значениях

аргумента. Это правило соблюдается для всех уровней пирамиды за исключением, быть может, последнего, если он не полон. На этом уровне $N - 2^{\lfloor \log_2 N \rfloor}$ элементов[3].

Поэтому последнее равенство можно переписать в виде

$$W_{\text{loop}} = 2\left(\sum_{k=1}^{d-1} k2^k + d(N - 2^d)\right), \text{ где } d = \lfloor \log_2 N \rfloor -$$

Равенство (1.19) дает

$$\begin{aligned} W_{\text{loop}}(N) &= 2[(d-2)2^d + 2 + d(N - 2^d)] \\ &= d2^{d+1} - 2^{d+2} + 4 + 2dN - d2^{d+1} \\ &= 2dN - 2^{d+2} + 4. \end{aligned}$$

Теперь, подставив $d = \lfloor \log_2 N \rfloor$, получим

$$\begin{aligned} W_{\text{loop}}(N) &= 2\lfloor \log_2 N \rfloor N - 2^{\lfloor \log_2 N \rfloor + 2} + 4 \\ &= 2\lfloor \log_2 N \backslash N - 4\lfloor \log_2 N \rfloor + 4 = O(N \log_2 N). \end{aligned}$$

Для получения окончательного результата осталось сложить сложности процедур построения и цикла. В результате получаем

$$\begin{aligned} W(N) &= W_{\text{Construction}}(N) + W_{\text{loop}}(N) \\ &= 4N - 2\log_2 N - 4 + 2\lfloor \log_2 N \rfloor N - 4\lfloor \log_2 N \rfloor + 4 \\ &\approx 4N + 2\lfloor \log_2 N \rfloor (N - 3); \\ W(N) &= O(N \log_2 N). \end{aligned}$$

3.5.2. Анализ среднего случая

Для анализа среднего случая в пирамидальной сортировке мы применим непривычный прием. Начнем с наилучшего случая, когда в исходном массиве значения расположены в обратном порядке. Несложно проверить, что такое расположение автоматически задает правильную

[3]Поскольку в показателе экспоненты стоит целая часть, неравенство $2^{\lfloor \log_2 N \rfloor} < N$ справедливо во всех случаях, за исключением того, когда N является степенью двойки; в последнем случае оно превращается в равенство.

пирамиду. Поэтому при каждом вызове процедуры FixHeap будет выполняться ровно два сравнения, подтверждающие, что элементы расположены правильно. Поскольку процедура FixHeap вызывается приблизительно для половины элементов, причем каждый вызов требует двух сравнений, на этапе построения пирамиды будет выполнено около N сравнений, т.е. столько же, сколько и в наихудшем случае.

Отметим, что независимо от начального расположения элементов результатом начального этапа всегда будет пирамида. Поэтому в любом случае цикл `for` будет исполняться столько же раз, сколько и в наихудшем: для получения отсортированного списка нам нужно забрать из пирамиды все элементы, всякий раз переформируя ее. Поэтому в наилучшем случае пирамидальная сортировка делает около $N + N \log N$ сравнений, т.е. сложность наилучшего случая равна $O(N \log N)$.

Итак, сложности наилучшего и наихудшего случая для пирамидальной сортировки совпадают и равны $O(N \log N)$. Это означает, что сложность среднего случая также обязана равняться $O(N \log N)$.

3.5.3. Упражнения

1) Каков будет порядок элементов списка [23, 17, 21, 3, 42, 9, 13, 1, 2, 7, 35, 4] после применения к нему этапа построения пирамиды?

2) Каков будет порядок элементов списка [3, 9, 14, 12, 2 17, 15, 8, 6, 18, 20, 1] после применения к нему этапа построения пирамиды?

3) Второй цикл `for` в пирамидальной сортировке можно было бы сократить, добавив условие окончания $r > 3$. Следует ли что-либо добавить после этого цикла, и если да, то что, для того, чтобы окончательный список по-прежнему был отсортированным? Приводят ли подобные изменения к уменьшению числа сравнений (дайте подробное объяснение)?

4) Докажите, что список, заполненный элементами в порядке убывания, действительно представляет собой пирамиду.

3.6. Сортировка слиянием

Сортировка слиянием — первая из встречающихся нам рекурсивных сортировок. В ее основе лежит замечание, согласно которому слияние двух отсортированных списков выполняется быстро. Список из одного элемента уже отсортирован, поэтому сортировка слиянием разбивает

список на одноэлементные куски, а затем постепенно сливает их. Поэтому вся деятельность заключается в слиянии двух списков.

Сортировку слиянием можно записать в виде рекурсивного алгоритма, выполняющего работу, двигаясь вверх по рекурсии. При взгляде на нижеследующий алгоритм Вы увидите, что он разбивает список пополам до тех пор, пока номер первого элемента куска меньше номера последнего элемента в нем. Если же в очередном куске это условие не выполняется, это означает, что мы добрались до списка из одного элемента, который тем самым уже отсортирован. После возврата из двух вызовов процедуры **MergeSort** со списками длиной один вызывается процедура **MergeLists**, которая сливает эти два списка, в результате чего получается отсортированный список длины два. При возврате на следующий уровень два списка длины два сливаются в один отсортированный список длины 4. Этот процесс продолжается, пока мы не доберемся до исходного вызова, при котором две отсортированные половины списка сливаются в общий отсортированный список. Ясно, что процедура MergeSort разбивает список пополам при движении по рекурсии вниз, а затем на обратном пути сливает отсортированные половинки списка. Вот этот алгоритм:

```
MergeSort(list,first,last)
list  сортируемый список элементов
first номер первого элемента в сортируемой части списка
last  номер последнего элемента в сортируемой части списка

if first<last then
   middle=(first+last)/2
   MergeSort(list,first,middle)
   MergeSort(list,middle+1,last)
   MergeLists(list,first,middle,middle+1,last)
end if
```

Ясно, что всю работу проделывает функция MergeLists. Теперь займемся ею.

Пусть A *m* B — два списка, отсортированных в порядке возрастания. Такое упорядочивание означает, что первым элементом в каждом списке является наименьший элемент в нем, а последним — наибольший. Мы знаем, что при слиянии двух списков в один наименьший элемент в объединенном списке должен быть первым либо в части A, либо в части B, а наибольший элемент в объединенном списке должен быть

последним либо в части *A*, либо в части *B*. Построение нового списка *C*, объединяющего списки *A* и *B*, мы начнем с того, что перенесем меньший из двух элементов A[1] и *B*[1] в *C*[1]. Но что должно попасть в *C*[2]? Если элемент *A*[1] оказался меньше, чем *B*[1], то мы перенесли его в *C*[1], и следующим по величине элементом может оказаться *B*[1] или *A[2]*. И тот, и другой вариант возможны, поскольку мы знаем только, что *A[2]* больше, чем *A*[1] и меньше, чем *A*[3], однако нам неизвестно отношение между величинами списка *A* и списка *B*. Похоже, что проще всего осуществить слияние, если завести два указателя, по одному на *A* и *B*, и увеличивать указатель того списка, очередной элемент в котором оказался меньше. Общая процедура продолжает сравнивать наименьшие элементы еще не просмотренных частей списков A и *B* и перемещать меньший из них в список *C*. В какой-то момент, однако, элементы в одном из списков *A* или *B* закончатся. В другом списке при этом останутся элементы, большие последнего элемента в первом списке. Нам необходимо переместить оставшиеся элементы в конец общего списка.

В совокупности эти рассуждения приводят к следующему алгоритму:

```
MergeLists(list,start1,end1,start2,end2)
list   упорядочиваемый список элементов
start1 начало "списка" A
endl   конец "списка" A
start2 начало "списка" B
end2   конец "списка" B

// предполагается, что элементы списков A и B
// следуют в списке list друг за другом
finalStart=start1
finalEnd=end2
indexC=l
while (startl<=endl) and (start2<=end2) do
   if list[start1]<list[start2] then
      result[indexC]=list[start1]
      start1=start1+1
   else
      result[indexC]=list[start2]
      start2=start2+l
   end if
```

```
   indexC=indexC+1
end while

// перенос оставшейся части списка
if (start1<=end1) then
   for i=start1 to end1 do
      result[indexC]=list[i]
      indexC=indexC+1
   end for
else
   for i=start2 to end2 do
      result[indexC]=list[i]
      indexC=indexC+1
   end for
end if

// возвращение результата в список
indexC=1
for i=finalStart1 to finalEnd do
   list [i]=result[indexC]
   indexC=indexC+1
end for
```

3.6.1. Анализ алгоритма MergeLists

Сравнением элементов занимается только процедура MergeLists, поэтому мы начнем с ее анализа. Посмотрим на случай, когда все элементы списка A меньше всех элементов списка B. Что будет делать процедура MergeLists? Она по-прежнему сравнивает $A[1]$ и $B[1]$, и поскольку элемент *A[1]* меньше, он переносится в список C. Затем сравниваются элементы $A[2]$ и *B[1]* и меньший элемент *A[2]* переносится в список C. Процесс сравнения всех элементов списка A с $B[1]$ продолжается пока мы не дойдем до конца списка A, так как все элементы в нем меньше, чем $B[1]$. Это означает, что алгоритм выполняет NA сравнений, где через NA обозначено число элементов в списке A. Наоборот, если все элементы списка B оказываются меньше всех элементов списка A, то число сравнений оказывается равным N_B, числу элементов в списке B.

А что если первый элемент списка A больше первого элемента списка B, но все элементы списка A меньше второго элемента списка B?

Тогда мы сравниваем $A[1]$ и $B[1]$, переносим элемент $B[1]$ в список C и оказываемся в той же ситуации, что и раньше: после сравнения каждого элемента списка A с $B[2]$ мы будем переносить его в C. В этом случае, однако, мы проделали не только NA сравнений элементов списка A с $B[2]$, но и сравнили *A[1]* с *B[1]*, поэтому полное число сравнений будет равно $NA + 1$. В остальных случаях также ясно, что описанная в первом абзаце настоящего раздела ситуация может быть, и действительно оказывается, наилучшим случаем.

Мы видели, что если все элементы списка A заключены между $B[1]$ и $B[2]$, то число сравнений увеличивается по сравнению с ситуацией, когда все элементы списка A меньше всех элементов списка B. Посмотрим, не приводит ли противоположная возможность к наихудшему случаю. Посмотрим, что будет, если значения элементов списков A и B идут «через один». Другими словами, что происходит, если *A[1]* находится между $B[1]$ и $B[2]$, *A[2]* находится между *B[2]* и $B[3]$, $A[3]$ находится между $B[3]$ и $B[4]$ и т.д. Отметим, что в результате каждого сравнения один из элементов списков A или B переносится в C. В том упорядочивании, которое рассматривалось выше, перенос происходит из обоих списков поочередно: сначала из B, затем из A, затем опять из B и т.д. пока не окажутся перенесенными все элементы за исключением последнего в списке A. Поскольку по результатам сравнения элементов обоих списков мы перенесли все элементы кроме одного, число сравнений в этом наихудшем случае оказывается равным $NA + N_B - 1$.

3.6.2. Анализ алгоритма MergeSort

Получив представление о рамках сложности алгоритма MergeLists, обратимся теперь к алгоритму MergeSort. На основе техники § 1.5 посмотрим на части алгоритма MergeSort. Заметим прежде всего, что эта функция вызывается рекурсивно пока значение переменной `first` меньше значения переменной last. Это означает, что если значения этих переменных равны или `first` больше, чем last, то рекурсивного вызова не происходит. Если `first` и last равны, то длина списка равна единице, если `first` больше, чем last, то мы имеем дело со списком длины нуль. В обоих случаях алгоритм ничего не делает, и число сравнений равно нулю.

Разбиение списка на две части происходит при вычислении значения переменной middle. Это вычисление не использует сравнений, поэтому вклад разбиения в число сравнений равен нулю. Значение переменной middle находится в точности посередине между `first` и last, поэтому

список разбивается на два подсписка, длина каждого из которых равна половине длины целого списка. Каждый из двух подсписков списка из N элементов состоит из $N/2$ элементов. По результатам анализа алгоритма MergeLsits это означает, что на этапе слияния потребуется в лучшем случае $N/2$ сравнений, а в худшем число сравнений будет равно $N/2+ N/2- 1 - N - 1$. Складывая все эти величины, получаем рекуррентные соотношения на сложность в наихудшем (W) и наилучшем (B) случаях:

$$W(N) = 2W(N/2)+ N - 1,$$

$$W(0)= W(1)= 0;$$

$$B(N) = 2B(N/2)+ N/2,$$

$$B(0)= B(1) = 0.$$

Решим теперь эти рекуррентные соотношения с помощью методов § 1.6. Начнем с худшего случая:

$$W(N/2)= 2W(N/4)+ N/2 - 1;$$

$$W(N/4)= 2W(N/8)+ N/4 - 1;$$

$$W(N/8) = 2W(N/16)+ N/8 - 1;$$

$$W(N/16)= 2W(N/32)+ N/16 - 1.$$

Теперь выполним подстановку:

$$\begin{aligned}
W(N) &= 2W(N/2)+ N - 1 \\
&= 2(2W(N/4)+ N/2 - 1) + N - 1 \\
&= 4W(N/4) + N - 2 + N-1 \\
&= 4(2W(N/8)+ N/4 - 1) + N - 2 + N-1 \\
&= 8W(N/8) + N - 4 + N-2 + N-1 \\
&= 8(2W(N/16)+ N/8 - 1) + N - 4 + N-2 + N-1 \\
&= 16W(N/16) + N-8 + N - 4 + N-2 + N - 1 \\
&= 16(2W(N/32)+ N/16 - 1) + N - 8 + N - 4 + N - 2 + N - 1
\end{aligned}$$

$$= 32W(N/32) + N - 16 + N\text{-}8 + N - 4 + N\text{-}2 + N - 1.$$

Видно, что коэффициент при W растет с той же скоростью, что и знаменатель. В конце концов этот член становится равным $W(1)$, т.е. нулю, и это слагаемое исчезает. Заметим, что при каждой подстановке происходит добавление слагаемого N и вычитание очередной большей степени двойки. Каков же будет результат? В правой части последнего равенства пять слагаемых N, сумма степеней двойки от нулевой ($1 = 2^0$) до четвертой ($16 = 2^4$), а также $W(N/32) = W(N/2^5)$. Поэтому когда мы дойдем до последнего равенства, содержащего $W(N/2^{\log N}) = W(N/N)$равенство приобретет следующий замкнутый вид:

$$W(N) = NW(1) + N\log_2 N - \sum_{i=0}^{\log_2 N - 1} 2^i$$

$$= N\log_2 N - (2^{\log_2 N} - 1)$$

$$= N\log_2 N - N + 1.$$

Это означает, что $W(N) = O(N\log N)$. Разница при переходе от $W(N)$ к $B(N)$ состоит в том, что N заменяется на $N/2$. При изучении роли, которую в подстановках играет N, можно подсчитать, что $B(N) = N\log_2 N/2$, поэтому, как и в наихудшем случае, $B(N) = O(N\log N)$.

Это означает, что сортировка слиянием чрезвычайно эффективна даже в наихудшим случае; проблема, однако, состоит в том, что процедуре `MergeLists` для слияния списков требуется дополнительная память.

3.6.3. Упражнения

1) Выпишите состояние списка $[7, 3, 9, 4, 2, 5, 6, 1, 8]$ после каждого прохода алгоритма MergeSort.

2) Выпишите состояние списка $[3, 5, 2, 9, 8, 1, 6, 4, 7]$ после каждого прохода алгоритма MergeSort.

3) При обсуждении алгоритма MergeLists упоминалось, что наилучший случай имеет место, если все элементы списка A меньше всех элементов списка B. Однако это ничего не говорит о производительности всего алгоритма на произвольном наборе входных данных. Сколько сравнений сделает алгоритм MergeSort на списке $[1, 2, 3, 4, 5, 6, 7, 8]$? Вообще, сколько сравнений будет выполнено

на списке, элементы которого уже идут в возрастающем порядке? Изучите действие алгоритма подробно.

4) Сколько в точности сравнений сделает алгоритм MergeSort на списке $[8, 7, 6, 5, 4, 3, 2, 1]$? Вообще, сколько сравнений будет выполнено на списке, элементы которого уже идут в убывающем порядке? Изучите действие алгоритма подробно.

5) Найдите порядок чисел от 1 до 8, при котором алгоритм MergeSort делает максимально возможное число сравнений, т.е. 17. *(Указание:* Пройдите процесс сортировки в обратном направлении.)

3.7. Быстрая сортировка

Быстрая сортировка — еще один рекурсивный алгоритм сортировки. Выбрав элемент в списке, быстрая сортировка делит с его помощью список на две части. В первую часть попадают все элементы, меньшие выбранного, а во вторую — большие элементы. Мы уже видели применение этой процедуры в алгоритме выбора в § 2.3. Алгоритм Quicksort действует по-другому, поскольку он применяется рекуррентно к обеим частям списка. В среднем такая сортировка эффективна, однако в наихудшем случае ее эффективность такая же, как у сортировки вставками и пузырьковой.

Быстрая сортировка выбирает элемент списка, называемый осевым, а затем переупорядочивает список таким образом, что все элементы. меньшие осевого, оказываются перед ним, а большие элементы — за ним. В каждой из частей списка элементы не упорядочиваются. Если $г$ — окончательное положение осевого элемента, то нам известно лишь, что все значения в позициях с первой по г — 1 меньше осевого, а значения с номерами от $г + 1$ до N больше осевого. Затем алгоритм Quicksort вызывается рекурсивно на каждой из двух частей. При вызове процедуры Quicksort на списке, состоящем из одного элемента, он ничего не делает, поскольку одноэлементный список уже отсортирован.

Поскольку основная работа алгоритма заключается в определении осевого элемента и последующей перестановке элементов, главная процедура в алгоритме Quicksort должна лишь проследить за границами обеих частей. Поскольку перемещение ключей происходит в процессе разделения списка на две части, вся работа по сортировке выполняется при возвращении по рекурсии.

Вот алгоритм быстрой сортировки:

```
Quicksort(list,first,last)
list  упорядочиваемый список элементов
first номер первого элемента в сортируемой части списка
last номер последнего элемента в сортируемой части списка

if first<last then
   pivot=PivotList(list,first,last)
   Quicksort(list,first,pivot-1)
   Quicksort(list,pivot+1,last)
end if
```

Расщепление списка

У функции PivotList есть по крайней мере два варианта. Первый из них, который легко запрограммировать и понять, описан в настоящем разделе. Второй вариант записать труднее, однако он работает быстрее. Он описан в упражнениях.

Функция PivotList берет в качестве осевого элемента первый элемент списка и устанавливает указатель pivot в начало списка. Затем она проходит по списку, сравнивая осевой элемент с остальными элементами списка. Обнаружив элемент, меньший осевого, она увеличивает указатель `PivotPoint`, а затем переставляет элемент с новым номером PivotPoint и вновь найденный элемент. После того, как сравнение части списка с осевым элементом уже произведено, список оказывается разбит на четыре части. Первая часть состоит из первого осевого — элемента списка. Вторая часть начинается с положения `first+1`, кончается в положении PivotPoint и состоит из всех просмотренных элементов, оказавшихся меньше осевого. Третья часть начинается в положении `PivotPoint+1` и заканчивается указателем параметра цикла index. Оставшаяся часть списка состоит из еще не просмотренных значений. Соответствующее разбиение приведено на рис. 3.4.

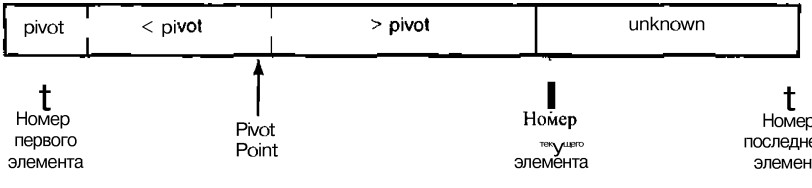


Рис. 3.4. Значения указателей в алгоритме PivotList

Вот алгоритм PivotList.

```
PivotList(list, first,last)
list  обрабатываемый список
first номер первого элемента
last  номер последнего элемента

PivotValue=list [first]
PivotPoint=first
for index=first+1 to last do
   if list [index]<PivotValue then
      PivotPoint=PivotPoint+1
      Swap(list[PivotPoint], list[index])
   end if
end for
// перенос осевого значения на нужное место
Swap(list[first],list[PivotPoint])
return PivotPoint
```

3.7.1. Анализ наихудшего случая

При вызове процедуры PivotList на списке из N элементов она выполняет $N - 1$ сравнение, поскольку значение PivotValue сравнивается со всеми остальными элементами списка. Как мы уже говорили, быстрая сортировка представляет собой алгоритм вида разделяй и властвуй, поэтому можно предположить, что в наилучшем случае PivotList создает две части одинакового размера. Так оно и есть. Тогда в наихудшем случае размеры списков должны быть максимально неравны. Наибольшая разность величин списков достигается, когда значение PivotValue либо больше, либо меньше всех остальных элементов списка. В этом случае в одном из списков нет ни одного элемента, а в другом $N - 1$ элемент. Если такая ситуация возникает при всяком рекурсивном вызове, то всякий раз будет происходить удаление одного элемента (PivotValue). Это означает, что число сравнений дается формулой

$$W(N) = \sum_{i=1}^{N} (i - 1) = \frac{N(N-1)}{2}.$$

Какой исходный порядок элементов приводит к такому результату? Если на каждом проходе выбирается первый элемент, то этот элемент

должен быть наименьшим (или наибольшим). Уже отсортированный список дает один пример наихудшего упорядочивания! Для рассмотренных нами раньше алгоритмов сортировки наихудший и средний случаи имеют приблизительно одинаковую сложность. Как мы увидим, для быстрой сортировки это не так.

3.7.2. Анализ среднего случая

Напомним, что анализ сортировки Шелла был основан на числе инверсий, удаляемых по результатам каждого сравнения. Мы отметили тогда, что и пузырьковая сортировка, и сортировка вставками плохо ведут себя в среднем случае, поскольку каждая из них удаляет по одной инверсии на одно сравнение.

А как удаляет инверсии быстрая сортировка? Рассмотрим список из N элементов, с которым работает алгоритм PivotList. Предположим, что PivotValue больше всех остальных элементов списка. Это означает, что в конце выполнения процедуры значение `PivotPoint` равно N, и поэтому переменная PivotValue будет указывать не на первый, а на последний элемент списка. Возможно также, что последний элемент списка является наименьшим в нем. Поэтому перестановка этих двух значений отправляет наибольший элемент списка из первой позиции в конец, а наименьший — из конца в начало. Если наибольший элемент стоит первым, то он входит в $N - 1$ инверсий с остальными элементами списка, а стоящий последним наименьший элемент также образует $N - 1$ инверсий с остальными элементами списка. Таким образом, одна перестановка двух крайних элементов убивает $2N - 2$ инверсий. Именно по этой причине средний случай быстрой сортировки значительно отличается от наихудшего.

Алгоритм PivotList выполняет основную часть работы, поэтому анализ среднего случая мы начнем именно с него. Заметим сначала, что после выполнения алгоритма PivotList любая из N позиций может содержать PivotValue.

Для анализа среднего случая рассмотрим каждую из этих возможностей и усредним результаты. При анализе наихудшего случая мы заметили, что процедура PivotList делает $N - 1$ сравнений, разбивая список из N элементов. Слияние этих списков не требует никаких действий. Наконец, заметим, что когда PivotList возвращает значение P, происходит рекурсивный вызов процедуры Quicksort на списках длины $P - 1$ и $N - P$. Проводимый нами анализ среднего случая должен

учитывать все N возможных значений P. Окончательно получаем рекуррентное соотношение

$$A(N) = (N-1) + \frac{1}{N}\left(\sum_{i=1}^{N}(A(i-1) + A(N-i))\right) \text{ для } N > 2,$$

$$A(1) = A(0) = 0.$$

Если повнимательней посмотреть на сумму, то видно, что аргумент первого слагаемого пробегает значения от 0 до $N-1$, а аргумент второго слагаемого — те же значения, но в порядке убывания.

Это означает, что каждое из слагаемых $A(i)$ участвует в сумме дважды, и формулу можно упростить:

$$A(N) = (N-1) + \frac{1}{N}\left(2\sum_{i=1}^{N-1} A(i)\right) \text{ для } N \geq 2,$$

$$A(1) = A(0) = 0.$$

В таком виде соотношение выглядит очень сложным, поскольку выражает $A(N)$ не через одно значение с меньшим аргументом, а через все эти значения сразу.

Преодолеть эту трудность можно двумя способами. Первый состоит в том, чтобы, используя свою подготовку, нащупать правильную формулу для результата, а затем доказать, что она удовлетворяет рекуррентному соотношению. Второй способ заключается в том, чтобы посмотреть сразу на две формулы: для $A(N)$ и для $A(N-1)$. Эти два выражения отличаются лишь несколькими членами.

Чтобы избавиться от дробей, подсчитаем $A(N)\cdot N$ и $A(N-1)\cdot(N-1)$. Получаем

$$A(N) \cdot N = (N-1)N + 2\sum_{i=0}^{N-1} A(i),$$

$$A(N) \cdot N = (N-1)N + 2A(N-1) + 2\sum_{i=0}^{N-2} A(i);$$

$$A(N-1)\cdot(N-1) = (N-2)(N-1) + 2\sum_{i=0}^{N-2} A(i).$$

Вычтя третье равенство из второго и проведя необходимые упрощения, получим

$$\begin{aligned} A(N)\cdot N - A(N-1)\cdot(N-1) &= 2A(N-1) + (N-1)N - (N-2)(N-1) \\ &= 2A(N-1) + N^2 - N - (N^2 - 3N + 2) \\ &= 2A(N-1) + 27\mathrm{V} - 2. \end{aligned}$$

Прибавление к обеим частям последнего равенства $A(N-1) \cdot (N-1)$ дает

$$\begin{aligned} A(N) \cdot N &= 2A(N-1) + A(N-1) \cdot (N-1) + 2N - 2 \\ &= A(N-1) \cdot (2 + N - 1) + 27\mathrm{V} - 2, \end{aligned}$$

откуда мы выводим окончательное рекуррентное соотношение

$$A(N) = \frac{(N+1)A(N-1) + 2N - 2}{N},$$

$$A(1) = A(0) = 0.$$

Решить его несложно, хотя решение и требует некоторой аккуратности, необходимой, чтобы учесть все слагаемые в правой части. Детальный анализ дает $A(N) \approx 1.4(N+1)\log_2 N$. Таким образом, средняя сложность алгоритма быстрой сортировки равна $O(N \log_2 N)$.

3.7.3. Упражнения

1) Протрассируйте алгоритм Quicksort на списке [23, 17, 21, 3, 42, 9, 13, 1, 2, 7, 35, 4]. Выпишите состояние списка и стека значений (`first,last,pivot`) перед каждым вызовом. Подсчитайте число выполненных сравнений и обменов элементов местами.

2) Протрассируйте алгоритм Quicksort на списке [3, 9, 14, 12, 2, 17, 15, 8, 6, 18, 20, 1]. Выпишите состояние списка и стека значений (`first,last,pivot`) перед каждым вызовом. Подсчитайте число выполненных сравнений и обменов элементов местами.

3) Мы показали, что на отсортированном списке алгоритм Quicksort работает плохо, поскольку осевой элемент всегда оказывается меньше всех остальных элементов списка. Попытка просто выбрать другую позицию в списке приводит к тому же результату, поскольку

при «несчастливом выборе» мы можем всякий раз получать наименьшее из оставшихся значений. Лучше было бы рассмотреть значения `list[first]`, `list[last]`, `list[(first+last)/2]` и выбрать среднее из них. Тогда сравнения, необходимые для выбора среднего значения, следует учесть при анализе алгоритма.

а) Решите упражнение 1) для модифицированного способа выбора оси.

б) Решите упражнение 2) для модифицированного способа выбора оси.

в) Сколько сравнений будет сделано при сортировке списка из N ключей в наихудшем случае? *(Замечание:* Теперь значение `PivotValue` уже наверняка не будет наименьшим, однако результат по-прежнему может быть очень плохим.)

4) Еще одна модификация алгоритма PivotList предусматривает наличие двух указателей в списке. Первый идет снизу, второй — сверху. В основном цикле алгоритма нижний указатель увеличивается до тех пор, пока не будет обнаружен элемент, больший `PivotValue`, а верхний уменьшается пока не будет обнаружен элемент, меньший PivotValue. Затем найденные элементы меняются местами. Этот процесс повторяется пока два указателя не перекроются. Эти внутренние циклы работают очень быстро, поскольку исключаются накладные расходы на проверку конца списка; проблема, однако, состоит в том, что при перекрещивании они делают лишний обмен. Поэтому алгоритм следует подкорректировать, добавив в него еще один обмен. Полный алгоритм выглядит так:

```
PivotList(list,first,last)
list  обрабатываемый список элементов
first номер первого элемента
last  номер последнего элемента

PivotValue=list[first]
lower=first
upper=last+1
do
   do upper=upper-1 until list[upper]<=PivotValue
```

```
    do lower=lower+1 until list[lower]>=PivotValue
    Swap(list[upper],list[lower])
until lower>=upper
// устранение лишнего обмена
Swap(list[upper],list[lower])
// перенос оси в нужное место
Swap(list[first],list[upper])
return upper
```

(Замечание: Этому алгоритму требуется место для хранения в конце списка дополнительного элемента, большего всех допустимых значений ключа.)

а) Решите упражнение 1) для модифицированного алгоритма PivotList.

б) Решите упражнение 2) для модифицированного алгоритма PivotList.

в) Какая операция выполняется в модифицированном алгоритме существенно реже, чем в исходном?

г) Сколько сравнений делает новый алгоритм PivotList в наихудшем случае на списке из N элементов? *(Замечание:* Это число не равно $N-1$.) Какое влияние оказывает это значение на общую эффективность быстрой сортировки в наихудшем случае?

5) Сколько сравнений выполнит алгоритм Quicksort на списке, состоящем из N одинаковых значений?

6) Какое максимальное число раз может алгоритм Quicksort перенести наибольшее или наименьшее значение?

3.8. Внешняя многофазная сортировка слиянием

Иногда сортируемый список оказывается настолько велик, что он не помещается целиком в память компьютера. Напомним, что хотя изучаемые нами алгоритмы имеют дело с упорядочиванием ключей, подразумевается, что эти ключи связаны с целыми записями. Во многих случаях длина записи значительно превышает длину ключа. Иногда

длина записи очень велика и перестановка двух записей занимает столько времени, что анализ эффективности алгоритма должен учитывать как число сравнений, так и число обменов.

Иногда допустимо объявить массив, размер которого достаточен для размещения всех требуемых данных, хотя размеры этого массива и значительно превышают доступный компьютеру объем памяти. Тогда операционная система пользуется виртуальной памятью, и следует учитывать эффективность ее использования. Однако даже и в этом случае объем обменов между оперативной памятью и дисками может быть значительным. Даже в эффективных алгоритмах сортировки вроде Quicksort соотношение между частями разбиения и длиной блока логической памяти может оказаться таким, что оно приведет к большому числу переписываний блоков. Нередко эта проблема не осознается, пока программа не будет реализована на компьютере, причем ее скорость оказывается настолько неудовлетворительной, что приходится применять средства технического анализа для выяснения узких мест. Даже и в этом случае анализ может оказаться безрезультатным, если операционная система не предоставляет инструментов отслеживания обменов в виртуальной памяти.

Для выяснения эффективности алгоритмов сортировки мы подсчитывали число выполняемых ими сравнений. Однако объем работы по чтению или записи на диск блоков виртуальной памяти может значительно превышать трудоемкость логических и арифметических операций. Эта работа выполняется операционной системой, и поэтому у нас нет реальных средств воздействия на ее эффективность.

При другом подходе можно воспользоваться файлами с прямым доступом и заменить непосредственные обращения к массиву операциями поиска в файле для выхода в нужную позицию с последующим чтением блока. В результате размер используемой логической памяти уменьшается, а значит уменьшается неконтролируемая нагрузка на виртуальную память. Объем операций ввода-вывода все равно остается значительным — управляем ли мы им сами или полагаемся на операционную систему.

В результате все описанные в последних семи параграфах алгоритмы сортировки на больших объемах данных оказываются непрактичными. Обратимся теперь к другой возможности: пусть у нас есть четыре файла и инструмент их слияния.

Оценим сначала разумное число записей, которые можно хранить в оперативной памяти одновременно. Объявим массив, длина S которого

равна этой величине; этот массив будет использоваться на двух этапах сортировки. На первом шаге мы прочитаем S записей и отсортируем их с помощью подходящей внутренней сортировки. Этот набор уже отсортированных записей перепишем в файл A. Затем прочитаем еще S записей, отсортируем их и перепишем в файл B. Этот процесс продолжается, причем отсортированные блоки записей пишутся попеременно то в файл Л, то в файл B. Вот алгоритм, реализующий первый этап:

```
CreateRuns(S)
S    размер создаваемых отрезков

CurrentFile=A
while конец входного файла не достигнут do
   read S записей из входного файла
   sort S записей
   if CurrentFile=A then
      CurrentFile=B
   else
      CurrrentFile=A
   end if
end while
```

После того, как входной файл полностью разбит на отсортированные отрезки, мы готовы перейти ко второму шагу - - слиянию этих отрезков. Каждый из файлов A и B содержит некоторую последовательность отсортированных отрезков, однако, как и в случае сортировки слиянием, мы ничего не можем сказать о порядке записей в двух различных отрезках.

Процесс слияния будет аналогичен функции `MergeLists` из § 3.6, однако теперь вместо того, чтобы переписывать записи в новый массив, мы будем записывать их в новый файл. Поэтому мы начинаем с чтения половинок первых отрезков из файлов A и B. Читаем мы лишь по половине отрезков, поскольку мы уже выяснили, что в памяти может находиться одновременно лишь S записей, а нам нужны записи из обоих файлов. Будем теперь сливать эти половинки отрезков в один отрезок файла C. После того, как одна из половинок закончится, мы прочтем вторую половинку из того же файла. Когда обработка одного из отрезков будет завершена, конец второго отрезка будет переписан в файл C. После того, как слияние первых двух отрезков из файлов A и B будет завершено, следующие два отрезка сливаются в файл D. Этот

процесс слияния отрезков продолжается с попеременной записью слитых отрезков в файлы C и D. По завершении мы получаем два файла, разбитых на отсортированные отрезки длины $2S$. Затем процесс повторяется, причем отрезки читаются из файлов C и D, а слитые отрезки длины $4S$ записываются в файлы A и B. Ясно, что в конце концов отрезки сольются в один отсортированный список в одном из файлов. Вот алгоритм осуществления второго этапа:

```
PolyPhaseMerge(S)
S    размер исходных отрезков
Size=S
Input1=A
Input2=B
CurrentOutput=C
while not done do
   while отрезки не кончились do
      слить отрезок длины Size из файла Input1
          с отрезком длины  Size из файла Input2
          записав результат в CurrentOutput
      if (CurrentOutput=A) then
         CurrentOutput=B
      elsif (CurrentOutput=B) then
         CurrrentOutput=A
      elsif (CurrentOutput=C) then
         CurrrentOutput=D
      elsif (CurrentOutput=D) then
         CurrrentOutput=C
      end if
   end while
   Size=Size*2
   if (Input1=A) then
      Input1=C
      Input2=D
      CurrentOutput=A
   else
      Input1=A
      Input2=B
      CurrentOutput=C
   end if
end while
```

Прежде, чем перейти к анализу, посмотрим, с каким количеством отрезков мы имеем дело, и как это количество влияет на число проходов. Если в исходном файле N записей и в память помещается одновременно 5 записей, то после выполнения процедуры `CreateRuns` мы получаем $R = \lceil N/S \rceil$ отрезков, распределенных по двум файлам. При каждом проходе алгоритма PolyPhaseMerge пары отрезков сливаются, поэтому число отрезков уменьшается вдвое. После первого прохода будет $\lceil R/2 \rceil$ отрезков, после второго — $\lceil R/4 \rceil$ и, в общем случае, после j-го прохода будет $\lceil R/2^j \rceil$ отрезков. Работа завершается, если остается один отрезок, т.е. после D проходов, где $\lceil R/2^D \rceil = 1$, т.е. при $D = \lceil \log_2 R \rceil$. Это означает, что алгоритм завершает свою работу после $\lceil \log_2 K \rceil$ проходов процесса слияния.

3.8.1. Число сравнений при построении отрезков

Мы не уточнили алгоритм, используемый при построении отсортированных отрезков; будем предполагать, что сложность используемой сортировки равна $O(N \log N)$. Поскольку длина отрезка равна 5, на каждый отрезок требуется $O(S \log 5)$ операций. Общее число отрезков равно R, поэтому на сортировку всех отрезков потребуется $O(RS \log 5) = O(N \log S)$ сравнений. Таким образом, сложность этапа построения отрезков равна $O(N \log S)$.

3.8.2. Число сравнений при слиянии отрезков

Мы показали в разделе 3.6.1, что в наихудшем случае слияние двух списков *из* A и B элементов алгоритмом `MergeLists` требует $A + B - 1$ сравнений. В нашем случае при первом проходе сливаются R отрезков длины S, т.е. всего происходит $R/2$ слияний, и каждое слияние требует не более 25 — 1 сравнений. Значит общее число сравнений при первом проходе не превосходит $R/2 \cdot (25 - 1) = RS - R/2$. При втором проходе мы имеем дело с Л/2 отрезками длины 25 каждый, поэтому всего выполняется Л/4 слияний, каждому из которых требуется не более 2(25) — 1 сравнений, т.е. общее число сравнений не превосходит $R/4 \cdot (45 - 1) = RS - R/4$. На третьем проходе мы сливаем $R/4$ отрезков длины 45, и каждое из этих $R/8$ слияний требует не более 2(45) — 1 сравнений, поэтому общее число сравнений оценивается величиной $R/8 \cdot (85 - 1) = RS - R/8$.

Если теперь вспомнить, что число проходов алгоритма слияния равно $\log_2 R$, то можно подсчитать общее число сравнений на этапе слияния:

$$\sum_{i=1}^{\log_2 R} (RS - R/2^i) = \underbrace{\sum_{i=1}^{\log_2 R} RS}_{} - \sum_{i=1}^{\log_2 R} R/2^i$$

$$= RS \log_2 R - R \sum_{i=1}^{\log_2 R} 1/2^i$$

$$\approx RS \log_2 R - R$$

$$= N \log_2 R - R.$$

Во втором равенстве мы воспользовались тем, что при вычислении суммы $1/2 + 1/4 + 1/8 + \ldots$ всегда получается число, меньшее 1, однако все более и более приближающееся к 1 при возрастании числа слагаемых. Чтобы представить себе это утверждение более наглядно, вообразите, что Вы стоите на расстоянии одного метра от стены и приближаетесь к ней шаг за шагом, всякий раз шагая на половину оставшегося от стены расстояния. Поскольку каждый шаг приближает Вас к стене лишь на половину оставшегося до нее расстояния, Вы никогда не достигнете ее, однако будете все время приближаться к ней. Аналогично, в приведенном вычислении к уже имеющейся сумме всякий раз добавляется половина разности между ней и числом 1. Это означает, что сумма будет все время приближаться к 1, но никогда не достигнет этого числа. Можно также воспользоваться равенством (1.18) при A — 0.5, произведя небольшой сдвиг (суммирование в (1.18) начинается при $r = 0$, а не при $r = 1$, как в нашем случае).

Итак, сложность этапа слияния отрезков равна $O(N \log R)$. Поэтому сложность алгоритма в целом есть

$$O(N \log S + N \log R) = O[N(\log S + \log R)]$$

$$= O[N \log(RS)] \quad \text{(в силу равенства (1.5))}$$

$$= O[N \log(S \cdot N/S)]$$

$$= O(N \log N).$$

3.8.3. Число операций чтения блоков

Чтение данных большими блоками происходит значительно быстрее, чем поэлементное чтение. Поэтому эффективнее всего многофазная сортировка слиянием работает при чтении записей большими блоками. Но в любом случае число операций чтения значительно влияет на эффективность.

На этапе построения отрезков мы читаем по одному блоку данных на отрезок, что дает R операций чтения блоков. Поскольку в памяти помещается только S записей и на этапе слияния нам нужны записи из двух отрезков, нам придется читать на этом этапе блоки размера $S/2$. При каждом проходе алгоритма слияния мы будем читать все данные; это означает, что на каждом проходе будет читаться $N/(S/2) - 2R$ блоков. Число проходов равно $\log_2 R$, поэтому общее число операций чтения при слиянии равно $2R \log_2 R$.

Таким образом, общее число операций чтения в алгоритме равно $R + 2R \log_2 R = O(R \log R)$.

3.8.4. Упражнения

1) Какие изменения следует произвести в алгоритме внешней многофазной сортировки, чтобы он мог обойтись тремя файлами вместо четырех? Как такие изменения повлияют на число сравнений и операций чтения блоков? *(Указание:* В новом варианте нельзя переключаться от одной пары файлов к другой и обратно.)

2) Какие изменения следует произвести в алгоритме внешней многофазной сортировки, чтобы он пользовался шестью файлами вместо четырех, сливая одновременно три списка? Как такие изменения повлияют на число сравнений и операций чтения блоков? А что можно сказать про восемь файлов?

3.9. Дополнительные упражнения

1) Сортировка выборкой просматривает список и выбирает наибольший (или наименьший) элемент в нем. Затем этот элемент меняется местами с тем, который стоит в списке последним (или первым). Процесс повторяется со списком на единицу меньшей длины, который не включает элемент, уже стоящий на месте. Такая процедура продолжается, пока список не будет отсортирован целиком.

   а) Напишите формальный алгоритм сортировки выборкой, который отбирает на каждом шаге наибольший элемент.

   б) Проанализируйте наихудший случай алгоритма из задания а).

   в) Проанализируйте средний случай алгоритма из задания а).

2) Сортировку счетом можно применять для списков, не содержащих дубликатов ключей. При сортировке счетом первый ключ в списке сравнивается со всеми ключами (включая самого себя) и при этом подсчитывается число ключей в списке, не превосходящих выбранный. Если обнаружится X ключей, не превосходящих данный, то он должен оказаться на позиции X. Такую процедуру можно повторить для всех остальных ключей, записывая подсчитанные позиции в отдельный массив, длина которого совпадает с длиной исходного. Когда подсчет будет завершен, дополнительный массив будет содержать список позиций, в которых должны находиться элементы исходного массива, и с его помощью можно создать отсортированный массив.

   а) Напишите формальный алгоритм сортировки счетом.

   б) Проанализируйте наихудший случай алгоритма из задания а).

   в) Проанализируйте средний случай алгоритма из задания а).

3) Иногда при применении алгоритма сортировки нас интересует, что происходит, если в сортируемом списке встречаются дубликаты ключей. Это важно в случае, когда происходит сортировка длинных записей по набору полей и хочется сохранить результаты уже выполненной сортировки. Пусть, например, фамилия и имя человека составляют два отдельных поля записи. Сначала можно отсортировать записи по значению имени, а затем по значению фамилии. Если алгоритм сохраняет порядок записей с одинаковой фамилией, то окончательный список будет правильно отсортирован по паре полей.

   Если всегда при `list [i] =list [j]` $(i < j)$ алгоритм сортировки помещает элемент `list [i]` в позицию i', а элемент `list [j]` в позицию j', такие что $i' < j'$, то такая сортировка называется устойчивой. Другими словами, алгоритм сортировки называется устойчивым, если он сохраняет относительный порядок записей с одинаковым ключом (хотя может и перемещать сами записи).

   Докажите устойчивость следующих алгоритмов:

   а) сортировка вставками;

   б) пузырьковая сортировка;

   в) сортировка Шелла;

г) корневая сортировка;

д) пирамидальная сортировка;

е) сортировка слиянием;

ж) быстрая сортировка.

3.10. Упражнения по программированию

Сложность алгоритма сортировки можно подсчитать следующим образом:

- Реализуйте алгоритм на каком-либо языке программирования.
- Установите глобальные счетчики `compareCount` и `swapCount`.
- Напишите головную программу с циклом, создающим случайный список и сортирующим его. При каждом проходе цикла подсчитывайте максимальное, минимальное и суммарное значения обоих счетчиков compareCount и swapCount. В конце работы все максимальные, минимальные и средние значения счетчиков можно распечатать. Чем больше итераций Вы выполните, тем точнее будут результаты.
- Если Вы сравниваете различные алгоритмы сортировки, то их следует выполнять на одном и том же списке или списках. Проще всего достичь этого, заведя свои счетчики для каждого из алгоритмов сортировки, а затем при генерации случайного списка делать его копии для каждого из алгоритмов. Все сортировки теперь можно прогнать на одном списке, а затем сгенерировать новый.

1) Воспользуйтесь предложенным планом для сравнительного анализа сортировки вставками и пузырьковой сортировки. Сложность обеих равна $O(N^2)$, но не показывает ли Ваш тест какой-нибудь разницы? Как согласуются Ваши результаты с анализом, проведенным в этой главе? Постарайтесь объяснить возникающие расхождения.

2) Воспользуйтесь предложенным планом для сравнительного анализа пирамидальной сортировки, сортировки слиянием и быстрой сортировки. Сложность всех трех в среднем случае равна $O(N\log N)$,

но не показывает ли Ваш тест какой-нибудь разницы? Как согласуются Ваши результаты с анализом, проведенным в этой главе? Постарайтесь объяснить возникающие расхождения.

3) Воспользуйтесь предложенным планом для сравнительного анализа основного варианта пузырьковой сортировки и других ее версий, описанных в упражнениях раздела 3.2.4. Не показывает ли Ваш тест какой-нибудь разницы? Как согласуются Ваши результаты с анализом, проведенным в этой главе и в упражнениях? Постарайтесь объяснить возникающие расхождения.

4) Воспользуйтесь предложенным планом для сравнительного анализа вариантов быстрой сортировки с версиями алгоритма PivotList, предложенными в тексте и в упражнениях раздела 3.7.3. Не показывает ли Ваш тест какой-нибудь разницы? Как согласуются Ваши результаты с анализом, проведенным в этой главе и в упражнениях? Постарайтесь объяснить возникающие расхождения.

5) Воспользовавшись описанным общим планом действий, проанализируйте влияние выбора шага на эффективность сортировки Шелла. Напишите вариант процедуры ShellSort с дополнительным параметром, представляющим собой массив шагов, которые следует использовать в порядке убывания. Вам потребуется также изменить эту функцию так, чтобы она использовала вводимые значения вместо шагов, которые считаются через степени двойки. Поработайте со случайными списками из 250 элементов и проверьте, что каждый из наборов шагов, которые обсуждались в § 3.3, был применен на каждом из списков. Как согласуются Ваши результаты с анализом, проведенным в этой главе? Постарайтесь объяснить возникающие расхождения.

6) Некоторым легче понять работу алгоритма, если ее визуализировать. Числовые ключи можно изображать вертикальными столбиками: левый столбик соответствует первому элементу, правый — последнему. Высота каждого столбика определяется величиной элемента. Если, например, элементы списка меняются от 1 до 500, то на месте, отвечающем элементу 1, будет находиться столбик высотой 1, а на месте, отвечающем элементу 500 — столбик высотой 500 единиц. В случайном списке все столбики будут перемешаны, а упорядоченный список будет выглядеть как треугольник с наибольшим столбиком справа.

Действие алгоритма сортировки можно увидеть, если после каждого прохода выводить на экран такой визуализированный список. В результате наблюдатель сможет проследить, как элементы перемещаются в требуемое положение. Напишите программу (или программы) визуализации сортировки. Размер списка может варьироваться от 250 до 500 в зависимости от возможностей Вашего компьютера. На высокоскоростном компьютере, возможно, придется вставить задержку, чтобы смена экранов происходила не слишком быстро.

а) Сортировка вставками — список выводится по окончании каждого прохода внешнего цикла.

б) Пузырьковая сортировка — список выводится по окончании каждого прохода внешнего цикла.

в) Сортировка Шелла — список выводится после каждого вызова модифицированной сортировки вставками.

г) Пирамидальная сортировка - - список выводится после каждого вызова процедуры FixHeap.

д) Сортировка слиянием — список выводится после каждого вызова процедуры `MergeLists`.

е) Быстрая сортировка — список выводится после каждого вызова процедуры `PivotList`.

Глава 4.

Численные алгоритмы

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- производить простые алгебраические преобразования;
- вычислять значения многочленов;
- описывать скорость и порядок роста функций.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- вычислять значения многочленов по схеме Горнера;
- вычислять значения многочленов путем предварительной обработки коэффициентов;
- анализировать сложность предварительной обработки коэффициентов;
- объяснять умножение матриц;
- трассировать алгоритм Винограда умножения матриц;
- анализировать алгоритм Винограда умножения матриц;
- пользоваться алгоритмом Штрассена умножения матриц.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Протрассируйте метод Горнера и вычислите препроцессированные коэффициенты для многочленов $x^3 + 4x^2 - 3x + 2$ и $x^7 - 8x^6 + 3x^5 + 2x^4 - 4x^3 + 5x - 7$. Протрассируйте стандартный алгоритм умножения матриц, алгоритм Винограда умножения матриц и алгоритм Штрассена умножения матриц для пары матриц

$$\begin{bmatrix} 1 & 4 \\ 5 & 8 \end{bmatrix} \quad \begin{bmatrix} 6 & 7 \\ 3 & 2 \end{bmatrix}.$$

Проследите за ходом выполнения метода Гаусса–Жордана на уравнениях

$$\begin{cases} 3x_1 + 9x_2 + 6x_3 = 21 \\ 5x_1 + 3x_2 + 22x_3 = 23 \\ 2x_1 + 8x_2 + 7x_3 = 26. \end{cases}$$

Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

Математические вычисления лежат в основе большого числа разнообразных программ. Компьютерная графика требует большого объема вычислений с многочленами и матрицами. Эти вычисления обычно делаются для каждой точки экрана, поэтому даже незначительные улучшения алгоритмов могут привести к заметному ускорению. Размер типичного изображения составляет 1024 точки по горизонтали на 1024 точки по вертикали. Даже выигрыш одного умножения на каждой из этих позиций приводит к экономии 1 048 576 умножений на всю картинку. Поэтому, хотя на первый взгляд развиваемая в настоящей главе техника и не приводит к радикальному повышению эффективности, ее многократное использование все же может давать большую экономию.

Иногда разрабатываемым программам приходится решать много сложных уравнений. Такие программы могут, например, считывать показатели внешних устройств и подставлять их в уравнения, решения которых говорят, следует ли применять какие-либо управляющие воздействия. Еще одно приложение - - вычисление тригонометрических функций. Чего не осознают многие программисты — это то, что стандартные процедуры вычисления значений тригонометрических функций типа синуса или косинуса используют их представление степенными

рядами, а начальные отрезки таких рядов представляют собой многочлены и вычисление значений тригонометрических функций сводится к вычислению значений многочленов. Вычислять же значения многочленов следует быстро, и мы начнем с эффективных способов делать это.

Умножение матриц встречается в многочисленных приложениях. Модели физических объектов в компьютерной графике, а также компьютерном проектировании и производстве, часто представляют собой матрицы, а операции над ними являются матричными операциями. В анализе образов свертка матриц используется для повышения качества изображения и для определения границ объекта на большой картине. Манипулируя матрицами, анализ Фурье выражает сложные волновые объекты через более простые синусоидальные колебания.

Общее свойство всех этих применений состоит в том, что матричные операции выполняются часто, поэтому чем быстрее они осуществляются, тем быстрее работают использующие их программы. Объекты преобразуются с помощью 4 x 4-матриц, в свертках используются матрицы размером от 3x3 до 11x11 или больше, однако эти операции выполняются многократно. Например, при свертке матрица умножается на блоки картинки для всех возможных положений блока. Это означает, что при применении шаблона размером 5 x 5 к картинке размером 512×512 точек (около четверти стандартного экрана компьютера) происходит умножение этой матрицы на 258 064 различных блока (508 x 508 = 258064). При использовании стандартного алгоритма умножения матриц такое действие потребует 32 258 000 умножений. Более эффективный алгоритм умножения матриц позволяет сэкономить значительное время.

В этой главе мы будем изучать способы повышения эффективности полиномиальных вычислений и матричного умножения. Нас интересует число сделанных арифметических операций, поэтому мы будем подсчитывать число выполняемых алгоритмом сложений и умножений. При изучении поиска и сортировки параметром эффективности служила длина списка. При анализе численных алгоритмов в качестве параметра будет выбираться либо наивысшая степень многочлена, либо размер перемножаемых матриц.

4.1. Вычисление значений многочленов

Займемся общим многочленом вида

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots + a_2 x^2 + a_1 x + a_0. \qquad (4.1)$$

Мы будем предполагать, что все коэффициенты $a_n, \ldots, a_0$ известны, постоянны и записаны в массив. Это означает, что единственным входным данным для вычисления многочлена служит значение ж, а результатом программы должно быть значение многочлена в точке x.

Стандартный алгоритм вычисления прямолинеен:

```
Evaluate(x)
x    точка, в которой вычисляется значение многочлена

result=a[0]+a[1]*x
xPower=x
for i=2 to n do
   xPower=xPower*x
   result=result+a[i]*xPower
end for
return result
```

Этот алгоритм совершенно прозрачен и его анализ очевиден. В цикле for содержится два умножения, которые выполняются $n - 1$ раз. Кроме того, одно умножение выполняется перед циклом, поэтому общее число умножений равно $2n - 1$. В цикле выполняется также одно сложение, и одно сложение выполняется перед циклом, поэтому общее число сложений равно n.

4.1.1. Схема Горнера

Вычисление по схеме Горнера оказывается более эффективным, причем оно не очень усложняется. Эта схема основывается на следующем представлении многочлена:

$$p(x) = ((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_2)x + a_1)x + a_0. \qquad (4.2)$$

Читатель легко проверит, что это представление задает тот же многочлен, что и (4.1). Соответствующий алгоритм выглядит так:

```
HornersMethod(x)
x   точка, в которой вычисляется значение многочлена

for i=n-1 down to 0 do
   result=result*x
   result=result+a[i]
```

```
end for
return result
```

Цикл выполняется n раз, причем внутри цикла есть одно умножение и одно сложение. Поэтому вычисление по схеме Горнера требует n умножений и n сложений — двукратное уменьшение числа умножений по сравнению со стандартным алгоритмом.

4.1.2. Предварительная обработка коэффициентов

Результат можно даже улучшить, если обработать коэффициенты многочлена до начала работы алгоритма. Основная идея состоит в том, чтобы выразить многочлен через многочлены меньшей степени. Например, для вычисления значения x^{256} можно воспользоваться таким же циклом, как и в функции Evaluate в начале этого параграфа; в результате будет выполнено 255 умножений. Альтернативный подход состоит в том, чтобы положить `result=x*x`, а затем семь раз выполнить операцию result=result*result. После первого выполнения переменная result будет содержать x^4, после второго x^8, после третьего x^{16}, после четвертого x^{32}, после пятого x^{64}, после шестого x^{128}, и после седьмого x^{256}.

Для того, чтобы метод предварительной обработки коэффициентов работал, нужно, чтобы многочлен был унимодальным (т.е старший коэффициент a_n должен равняться 1), а степень многочлена должна быть на единицу меньше некоторой степени двойки (п = 2 ' — 1 для некоторого $k > 1$)[1]. В таком случае многочлен можно представить в виде

$$p(x) = (x^j + b)q(x) + r(x), \qquad \text{где } j = 2^{k-1}. \tag{4.3}$$

В обоих многочленах q и r будет вдвое меньше членов, чем в p. Для получения нужного результата мы вычислим по отдельности $q(x)$ и г(x), а затем сделаем одно дополнительное умножение и два сложения. Если при этом правильно выбрать значение 6, то оба многочлена q и r оказываются унимодальными, причем степень каждого из них позволяет применить к каждому из них ту же самую процедуру. Мы увидим, что ее последовательное применение позволяет сэкономить вычисления.

[1] **Преимущества** описываемого подхода могут оказаться настолько велики, что иногда стоит добавить к многочлену слагаемые, позволяющие его применить, а затем вычесть добавленные значения из окончательного результата. Другими словами, если мы имеем дело с многочленом тридцатой степени, то нужно прибавить к нему x^{31}, найти разложение, а затем вычесть x^{31} из каждого результата вычисления. Алгоритм все равно будет работать быстрее остальных методов.

Вместо того, чтобы вести речь о многочленах общего вида, обратимся к примеру. Рассмотрим многочлен

$$p(x) = x^7 + 4x^6 - 8x^4 + 6x^3 + 9x^2 + 2x - 3.$$

Определим сначала множитель $x^j + b$ в уравнении (4.3). Степень многочлена p равна 7, т.е. $2^3 - 1$, поэтому $k = 3$. Отсюда вытекает, что $j = 2^2 = 4$. Выберем значение b таким образом, чтобы оба многочлена q и r были унимодальными. Для этого нужно посмотреть на коэффициент a_{j-1} в p и положить $b = a_{j-1} - 1$. В нашем случае это означает, что $b = a_3 - 1 = 5$. Теперь мы хотим найти значения q и r, удовлетворяющие уравнению

$$x^7 + 4x^6 - 8x^4 + 6x^3 + 9x^2 + 2x - 3 = (x^4 + 5)q(x) + r(x).$$

Многочлены q и r совпадают соответственно с частным и остатком от деления p на $x^4 + 5$. Деление с остатком дает

$$p(x) = (x^4 + 5)(x^3 + 4x^2 + 0x + 8) + (x^3 - 11x^2 + 2x - 37).$$

На следующем шаге мы можем применить ту же самую процедуру к каждому из многочленов q и r:

$$q(x) = (x^2 - 1)(x + 4) + (x + 12), \qquad r(x) = (x^2 + 1)(x - 11) + (x - 26).$$

В результате получаем

$$p(x) = (x^4 + 5)((x^2 - 1)(x + 4) + (x + 12)) + ((x^2 + 1)(x - 11) + (x - 26)).$$

Посмотрев на этот многочлен, мы видим, что для вычисления x^2 требуется одно умножение; еще одно умножение необходимо для подсчета $x^4 = x^2 \cdot x^2$. Помимо этих двух умножений в вычислении правой части равенства участвуют еще три умножения. Кроме того, выполняется 10 операций сложения. В таблице на рис. 4.1 приведены сравнительные результаты анализа этого метода и других методов вычисления. Экономия не выглядит значительной, однако это объясняется тем, что мы рассматриваем лишь частный случай. Общую формулу для сложности можно вывести, внимательно изучив процедуру. Заметим прежде всего, что в равенстве (4.3) участвуют одно умножение и два сложения. Поэтому для числа умножений $M = M(k)$ и числа сложений $A = A(k)$ мы получаем следующий набор рекуррентных соотношений:

$$M(1) = 0 \qquad A(1) = 0$$
$$M(k) = 2M(k-1) + 1 \text{ при } k > 1 \qquad A(k) = 2A(k-1) + 2 \text{ при } k > 1.$$

Способ	Умножения	Сложения
Стандартный	13	7
Схема Горнера	7	7
Предварительная обработка	5	10

Рис. 4.1. Число операций при вычислении значения многочлена степени 7

Решая эти соотношения, мы заключаем, что число умножений приблизительно равно $N/2$, а число сложений приблизительно равно $(3N-1)/2$. Неучтенными, однако, остались умножения, необходимые для подсчета последовательности значений $x^2, x^4, x^8, \ldots, x^{2^{k-1}}$; на это требуется дополнительно $k-1$ умножение. Поэтому общее число умножений приблизительно равно $N/2+\log_2 N$.

Способ	Умножения	Сложения
Стандартный	$2N-1$	N
Схема Горнера	N	N
Предварительная обработка	$\frac{N}{2}+\log_2 N$	$\frac{3N-1}{2}$

Рис. 4.2. Число операций при вычислении значения многочлена степени N

В таблице на рис. 4.2 приведены результаты сравнительного анализа стандартного алгоритма, схемы Горнера и алгоритма с предварительной обработкой коэффициентов. При сравнении последних двух алгоритмов видно, что нам удалось сэкономить $N/2-\log_2 N$ умножений, но за счет дополнительных $(N-1)/2$ сложений. Во всех существующих вычислительных системах обмен умножений на сложения считается выгодным, поэтому предварительная обработка коэффициентов повышает эффективность.

4.1.3. Упражнения

1) Выпишите разложение многочлена $x^7+2x^6+6x^5+3x^4+7x^3+5x+4$

   а) по схеме Горнера;

   б) с предварительной обработкой коэффициентов.

2) Выпишите разложение многочлена $x^7+6x^6+4x^4-2x^3+3x^2-7x+5$

   а) по схеме Горнера;

   б) с предварительной обработкой коэффициентов.

4.2. Умножение матриц

Матрица - - математический объект, эквивалентный двумерному массиву. Числа располагаются в матрице по строкам и столбцам. Две матрицы одинакового размера можно поэлементно сложить или вычесть друг из друга. Если число столбцов в первой матрице совпадает с числом строк во второй, то эти две матрицы можно перемножить. У произведения будет столько же строк, сколько в первой матрице, и столько же столбцов, сколько во второй. При умножении матрицы размером 3 x 4 на матрицу размером 4×7 мы получаем матрицу размером 3×7. Умножение матриц некоммутативно: оба произведения AB и BA двух квадратных матриц одинакового размера можно вычислить, однако результаты, вообще говоря, будут отличаться друг от друга. (Отметим, что умножение чисел коммутативно, и произведения AB и BA двух чисел A и B равны.)

Для вычисления произведения двух матриц каждая строка первой почленно умножается на каждый столбец второй. Затем подсчитывается сумма таких произведений и записывается в соответствующую клетку результата. На рис. 4.3 приведен пример умножения двух матриц.

Умножение матриц на рис. 4.3 требует 24 умножений и 16 сложений. Вообще, стандартный алгоритм умножения матрицы размером a x b на матрицу размером b x c выполняет abc умножений и $a(b-1)c$ сложений.

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \end{bmatrix} = \begin{bmatrix} aA+bE+cI & aB+bF+cJ & aC+bG+cK & aD+bH+cL \\ dA+eE+fI & dB+eF+fJ & dC+eG+fK & dD+eH+fL \end{bmatrix}$$

Рис. 4.3. Умножение 2 x 3-матрицы на 3 x 4-матрицу

Вот как выглядит стандартный алгоритм умножения матрицы G размером a x 6 на матрицу *H* размером *Ъ* x *c*. Результат записывается в матрицу *R* размером *a x c*.

```
for i=1 to a do
   for j=1 to c do
      R[i,j]=0
      for k=1 to b do
         R[i,j]=R[i,j]+G[i,k]*H[k,j]
      end for k
   end for j
end for i
```

На первый взгляд это минимальный объем работы, необходимый для перемножения двух матриц. Однако исследователям не удалось доказать минимальность, и в результате они обнаружили другие алгоритмы, умножающие матрицы более эффективно.

4.2.1. Умножение матриц по Винограду

Если посмотреть на результат умножения двух матриц, то видно, что каждый элемент в нем представляет собой скалярное произведение соответствующих строки и столбца исходных матриц. Можно заметить также, что такое умножение допускает предварительную обработку, позволяющую часть работы выполнить заранее.

Рассмотрим два вектора $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$. Их скалярное произведение равно

$$V \cdot W = v_1 w_1 + v_2 w_2 + v_3 w_3 + v_4 w_4.$$

Это равенство можно переписать в виде

$$V \cdot W = (v_1 + 102)(v_2 + w_1) + (v_3 + w_4)(v_4 + w_3) - v_1 v_2 - v_3 v_4 - w_1 w_2 - w_3 w_4. \quad (4.4)$$

Читатель без труда проверит эквивалентность двух последних выражений. Кажется, что второе выражение задает больше работы, чем первое: вместо четырех умножений мы насчитываем их шесть, а вместо трех сложений — десять. Менее очевидно, что выражение в правой части последнего равенства допускает предварительную обработку: его

части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй. На практике это означает, что над предварительно обработанными элементами нам придется выполнять лишь первые два умножения и последующие пять сложений, а также дополнительно два сложения.

Вот как выглядит полный алгоритм Винограда для умножения матрицы G размером a x b на матрицу H размером b x c. Результат записывается в матрицу R размером a x c.

```
d=b/2
// вычисление rowFactors для G
for i=1 to a do
   rowFactor[i]=G[i,1]*G[i,2]
   for j=2 to d do
      rowFactor[i]=rowFactor[i]+G[i,2j-1]*G[i,2j]
   end for j
end for i

// вычисление columnFactors для H
for i=1 to c do
   columnFactor[i]=H[1,i]*H[2,i]
   for j=2 to d do
      columnFactor[i]=columnFactor[i]+H[2j-1,i]*H[2j,i]
   end for j
end for i

// вычисление матрицы R
for i=1 to a do
   for j=1 to c do
      R[i,j]=-rowFactor[i]-columnFactor[j]
      for k=1 to d do
         R[i,j]=R[i,j]+(G[i,2k-1]+H[2k,j])*(G[i,2k]+H[2k-1,j])
      end for k
   end for j
end for i

// прибавление членов в случае нечетной общей размерности
if (2*(b/2) /= b) then
   for i=1 to a do
      for j=1 to c do
```

```
            R[i,j]=R[i,j]+G[i,b]*H[b,j]
        end for j
    end for i
end if
```

Анализ алгоритма Винограда

Рассмотрим случай четной общей размерности b. Результаты подсчета сложений и умножений сведены в следующую таблицу[2].

	Умножения	Сложения
Предварительная обработка матрицы G	ad	$a(d-1)$
Предварительная обработка матрицы H	cd	$c(d-1)$
Вычисление матрицы R	acd	$ac(2d+d+1)$
Всего	$(abc+ab+bc)/2$	$(a(b-2)+c(b-2)+ac(3b+2))/2$

4.2.2. Умножение матриц по Штрассену

Алгоритм Штрассена работает с квадратными матрицами. На самом деле он настолько эффективен, что иногда разумно расширить матрицы до квадратных, и при этом он все равно дает выигрыш, превышающий расходы на введение дополнительных элементов.

Для умножения 2 x 2-матриц алгоритм Штрассена использует семь формул. Эти формулы чрезвычайно неестественны и, к сожалению, в оригинальной статье Штрассена не объясняется, как он пришел к ним. Замечательно, что как сами формулы, так и их использование не требуют, чтобы умножение элементов матриц было коммутативным. Это

[2]Слагаемое $2d$ при подсчете числа сложений при вычислении элементов матрицы R возникает из двух сложений внутри произведений, слагаемое d — из необходимости складывать результаты умножения, а слагаемое 1 — из необходимости инициализации результата.

означает, в частности, что сами эти элементы могут быть матрицами, а значит, алгоритм Штрассена можно применять рекурсивно. Вот формулы Штрассена:

$$x_1 = (G_{1,1} + G_{2,2})(H_{1,1} + H_{2,2}); \quad x_5 = (G_{1,1} + G_{2,2})H_{2,2};$$

$$x_2 = (G_{2,1} + G_{2,2})H_{1,1}; \quad x_6 = (G_{2,1} - G_{1,1})(H_{1,1} + H_{1,2});$$

$$x_3 = G_{1,1}(H_{1,2} - H_{2,2}); \quad x_7 = (G_{2,1} - G_{2,2})(H_{2,1} + H_{2,2}).$$

$$x_4 = G_{2,2}(H_{2,1} - H_{1,1});$$

Теперь элементы матрицы R могут вычисляться по формулам

$$R_{1,1} = x_1 + x_4 - x_5 + x_7; \quad R_{1,2} = x_3 + x_5;$$

$$R_{2,1} = x_2 + x_4; \quad R_{2,2} = x_1 + x_3 - x_2 + x_6.$$

При умножении двух 2 x 2-матриц алгоритм выполняет 7 умножений и 18 сложений. Экономия не видна: мы получили 14 сложений в обмен на одно умножение в стандартных алгоритмах. Анализ общего случая показывает, что число умножений при перемножении двух N x N-матриц приблизительно равно $N^{2.81}$, а число сложений $6N^{2.81} - 6N^2$. При умножении двух 16 x 16-матриц алгоритм Штрассена экономит 1677 умножений за счет дополнительных 9138 сложений.

Сводя три результата воедино, мы получаем следующую картину.

	Умножения	Сложения
Стандартный алгоритм	N^3	$N^3 - N^2$
Алгоритм Винограда	$\frac{N^3+2N^2}{2}$	$\frac{3N^3+4N^2-4N}{2}$
Алгоритм Штрассена	$N^{2.81}$	$6N^{2.81} - 6N^2$

На практике алгоритм Штрассена применяется редко: его использование требует аккуратного отслеживания рекурсии. Важность его состоит в том, что это первый алгоритм, умножение матриц с помощью которого требует менее, чем $O(N^3)$ операций. Работа над повышением эффективности матричного умножения и поиском возможных нижних оценок его сложности продолжается.

4.2.3. Упражнения

1) Сколько умножений и сложений выполняет алгоритм Винограда при нечетной общей размерности матриц?

2) Продемонстрируйте работу алгоритма Штрассена при умножении пары матриц

$$\begin{bmatrix} 1 & 9 \\ 7 & 3 \end{bmatrix} \text{ и } \begin{bmatrix} 5 & 2 \\ 4 & 11 \end{bmatrix}.$$

Сравните этот результат с работой стандартного алгоритма.

4.3. Решение линейных уравнений

Система линейных уравнений состоит из N уравнений с N неизвестными. Система общего вида выглядит так:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + a_{N3}x_3 + \cdots + a_{NN}x_N &= b_N. \end{aligned}$$

Такие системы могут иметь самое разное происхождение, однако обычно константы (представленные коэффициентами a) являются некоторыми значениями параметров приборов, показания которых представлены значениями b в правой части равенств. Нас интересует, при каких значениях параметров достигаются такие результаты. Рассмотрим следующий пример:

$$\begin{aligned} 2x_1 + 7x_2 + 1x_3 + 5x_4 &= 70 \\ 1x_1 + 5x2 + 3xз + 2x4 &= 45 \\ 3x_1 + 2x_2 + 5x_3 + 1x4 &= 33 \\ 8x_1 + 1x2 + 5xз + 3x4 &= 56. \end{aligned}$$

Один из способов решения системы линейных уравнений состоит в выполнении подстановок. Например, второе уравнение можно переписать в виде $x_1 = 45 - 5x_2 - 3xз - 2x4$, а затем подставить выражение в правой части вместо x_1 в остальные три уравнения. В результате мы

получим три уравнения с тремя неизвестными. Затем в одном из оставшихся уравнений можно проделать то же самое с неизвестным x_2, получится два уравнения с двумя неизвестными. Еще одно преобразование с переменной жз — и мы получим одно уравнение с одним неизвестным (а именно, x_4). Это уравнение можно решить, получив значение неизвестного Ж4; подставив полученное решение в одно из двух уравнений предыдущего этапа, мы можем найти значение неизвестного x_3. После подстановки найденных значений жз и x_4 в одно из трех уравнений, полученных в результате первой подстановки, мы сможем найти Ж2, и, наконец, любое из исходных уравнений даст нам значение x_1.

Такая процедура очень хорошо работает, однако при ее выполнении производится очень много алгебраических преобразований, в которые легко может вкрасться ошибка. При возрастании числа уравнений и неизвестных время на выполнение этих преобразований быстро растет, а запрограммировать их не так-то легко. Однако именно они лежат в основе метода Гаусса–Жордана, который мы сейчас и опишем.

4.3.1. Метод Гаусса–Жордана

Систему линейных уравнений можно представлять матрицей с N строками и $N + 1$ столбцами. В предыдущем примере матрица имеет вид

$$\begin{bmatrix} 2 & 7 & 1 & 5 & 70 \\ 1 & 5 & 3 & 2 & 45 \\ 3 & 2 & 4 & 1 & 33 \\ 8 & 1 & 5 & 3 & 56 \end{bmatrix}.$$

Теперь со строками этой матрицы можно выполнить некоторые преобразования, которые и приведут к требуемому результату. Если первые N столбцов и строк матрицы представляют собой единичную матрицу, то последний столбец состоит из искомых значений ж:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & Ж1 \\ 0 & 1 & 0 & 0 & x_2 \\ 0 & 0 & 1 & 0 & x_3 \\ 0 & 0 & 0 & 1 & x_4 \end{bmatrix}.$$

В основе плана действий лежит следующее преобразование: нужно разделить первую строку на ее первый элемент, а затем вычесть кратные этой первой строки из всех последующих строк.

В нашем примере из второй строки следует вычесть первую, из третьей — утроенную первую, из четвертой — первую, умноженную на 8. Ясно, что в результате первый столбец приобретет требуемый вид. Вот как выглядит новая матрица:

$$\begin{bmatrix} 1 & 3.5 & 0.5 & 2.5 & 35 \\ 0 & 1.5 & 2.5 & -0.5 & 10 \\ 0 & -8.5 & 2.5 & -6.5 & -72 \\ 0 & -27 & 1 & -17 & -224 \end{bmatrix}.$$

Повторим теперь эти действия для второй строки. После того, как мы разделим эту строку на значение ее второго элемента (т.е. на 1.5), значения вторых элементов первой, третьей и четвертой строк определят, на сколько следует умножить новую вторую строку перед ее вычитанием. Новая матрица (с округленными элементами) имеет вид

$$\begin{bmatrix} 1 & 0 & -5.33 & 3.66 & 11.7 \\ 0 & 1 & 1.6 & -0.33 & 6.67 \\ 0 & 0 & 16.7 & -9.3 & 15.3 \\ 0 & 0 & 46 & -26 & -44 \end{bmatrix}.$$

Повторим теперь этот процесс для третьей строки — получим нужный третий столбец, а затем с четвертой:

$$\begin{bmatrix} 1 & 0 & 0 & 0.68 & 6.76 \\ 0 & 1 & 0 & 0.6 & 8.2 \\ 0 & 0 & 1 & -0.56 & -0.96 \\ 0 & 0 & 0 & -0.24 & -1.68 \end{bmatrix};$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 & 4 \\ 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 7 \end{bmatrix}.$$

Окончательно, получаем искомые значения неизвестных $x_1 = 2$, $X2 = 4$, $x_3 = 3$ и $x_4 = 7$. При реализации этого процесса на компьютере мы, однако, сталкиваемся с проблемой округления. При итерации цикла ошибки округления накапливаются, за счет чего результаты могут оказаться очень неточными. При решении больших систем линейных

уравнений ошибки могут стать значительными. Существуют другие алгоритмы решения линейных уравнений, которые позволяют, по крайней мере, проконтролировать ошибки округления, однако эти алгоритмы естественно изучать в курсе численного анализа, и мы не будем на них останавливаться.

Еще одна проблема возникает, если две строки оказываются кратными. В таком случае мы получим чисто нулевую строку, и алгоритм наткнется на ошибку деления на нуль. Такая ситуация называется «особенностью»; обработка особенностей выходит за рамки нашей книги.

4.3.2. Упражнения

1) Опишите шаги алгоритма Гаусса-Жордана для следующей системы линейных уравнений:

$$3x_1 + 6x_2 + 12x_3 + 9x_4 = 78$$
$$2x_1 + 3x_2 + 5x_3 + 7x_4 = 48$$
$$1x_1 + 7x_2 + 2x_3 + 3x_4 = 27$$
$$4x_1 + 9x2 + 1x_3 + 2x_4 = 45.$$

2) Опишите шаги алгоритма Гаусса-Жордана для следующей системы линейных уравнений:

$$2x_1 + 4x_2 + 5x_3 = 23$$
$$1x_1 + 5x_2 + 3x_3 = 16$$
$$3x_1 + 1x2 + 6x_3 = 25.$$

3) Воспользовавшись описанием в разделе 4.2.1, проанализируйте алгоритм Гаусса–Жордана решения системы из N линейных уравнений с N неизвестными. Результатом Вашего анализа должны служить число выполняемых умножений и число выполняемых сложений.

Глава 5.

Алгоритмы сравнения с образцом

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- создавать конечные автоматы;
- пользоваться символьными строками;
- пользоваться одномерными и двумерными массивами;
- описывать скорость и порядок роста функций.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- формулировать задачу поиска строки по образцу;
- объяснять прямой алгоритм поиска по образцу и проводить его анализ;
- объяснять использование конечных автоматов для поиска по образцу;
- строить и использовать автомат Кнута–Морриса–Пратта;
- строить и использовать массивы сдвигов и прыжков для алгоритма Бойера–Мура;
- объяснять метод приблизительного совпадения строк.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Выполните трассировку поиска образца `abcabc` в тексте abccbaabcabcbccabc для прямого алгоритма и алгоритма Кнута–Морриса–Пратта, а также трассировку алгоритма Бойера–Мура на том же тексте с образцом abcbccabc. Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

Поиск подстроки в длинном куске текста — важный элемент текстовых редакторов. В начале этой главы мы описываем четыре возможных подхода к этой задаче. Поиск по образцу мы будем вести на примере текстовых строк. Однако ту же самую технику можно использовать для поиска битовых или байтовых строк в двоичном файле. Так, например, осуществляется поиск вирусов в памяти компьютера.

В программах обработки текстов обычно имеется функция проверки синтаксиса, которая не только обнаруживает неправильно написанные слова, но и предлагает варианты их правильного написания. Один из подходов к проверке состоит в составлении отсортированного списка слов документа. Затем этот список сравнивается со словами, записанными в системном словаре и словаре пользователя; слова, отсутствующие в словарях, помечаются как возможно неверно написанные. Процесс выделения возможных правильных написаний неверно набранного слова может использовать приблизительное совпадение с образцом.

При обсуждении приблизительных совпадений строк речь идет о поиске подстроки в данном отрезке текста. Ту же самую технику можно использовать и для поиска приблизительных совпадений в словаре.

5.1. Сравнение строк

Наша задача состоит в том, чтобы найти первое вхождение некоторой подстроки в длинном тексте. Поиск последующих вхождений основан на том же подходе. Это сложная задача, поскольку совпадать должна вся строка целиком. Стандартный алгоритм начинает со сравнения первого символа текста с первым символом подстроки. Если они совпадают, то происходит переход ко второму символу текста и подстроки. При совпадении сравниваются следующие символы. Так про-

```
Текст:      there they are
Проход 1:   they
Текст:      there they are
Проход 2:    they
Текст:      there they are
Проход 3:      they
Текст:      there they are
Проход 4:        they
Текст:      there they are
Проход 5:          they
Текст:      there they are
Проход 6:            they
Текст:      there they are
Проход 7:              they
```

Рис. 5.1. Поиск образца they в тексте there they are. При первом проходе три первых символа подстроки совпадают с символами текста. Однако только седьмой проход дает полное совпадение. (Совпадение находится после 13 сравнений символов.)

должается до тех пор, пока не окажется, что подстрока целиком совпала с отрезком текста, или пока не встретятся несовпадающие символы. В первом случае задача решена, во втором мы сдвигаем указатель текущего положения в тексте на один символ и заново начинаем сравнение с подстрокой. Этот процесс изображен на рис. 5.1.

Следующий алгоритм осуществляет стандартное сравнение строк:

subLoc=1 //указатель текущ. сравниваемого символа в подстроке
textLoc=1 **//указатель** текущего сравниваемого символа в тексте
textStart=1 **//указатель** на начало сравнения в тексте

```
while TextLoc<=length(text) and subLoc<=length(substring) do
   if text[textLoc]=substring[subLoc] then
      textLoc=textLoc+1
      subLoc=subLoc+1
   else
      // начать сравнение заново со следующего символа
      textStart=textStart+1
      textLoc=textLoc+1
      subLoc=subLoc+1
```

```
    end if
end while

if (subLoc>length(substring)) then
   return textStart // совпадение найдено
else
   return  0          // совпадение не найдено
end if
```

Из алгоритма ясно, что основная операция — сравнение символов, и именно число сравнений и следует подсчитывать. В наихудшем случае при каждом проходе совпадают все символы за исключением последнего. Сколько раз это может произойти? По одному разу на каждый символ текста. Если длина подстроки равна 5, а длина текста равна T, то, похоже, в наихудшем случае число сравнений будет равно $S(T-S+1)$.[1] Посмотрим теперь, возможна ли в принципе такая ситуация. Будем искать подстроку **XX**...XXY, состоящую из $S-1$ буквы X и одной буквы Y, в тексте **XX**...XXXXX, состоящем из T букв X. Такой набор символов и дает искомое наихудшее поведение. Следует отметить, что в естественных языках таких слов обычно не бывает, поэтому можно ожидать, что производительность приведенного алгоритма на текстах естественного языка оказывается намного выше. Как показывают исследования, на текстах естественного языка сложность приведенного алгоритма оказывается немногим больше T.[2]

Проблема стандартного алгоритма заключается в том, что он затрачивает много усилий впустую. Если сравнение начала подстроки уже произведено, то полученную информацию можно использовать для того, чтобы определить начало следующего сравниваемого отрезка текста. Например, при первом проходе на рис. 5.1 расхождение с образцом осуществляется на четвертом символе, а в первых трех символах произошло совпадение. При взгляде на образец видно, что третий символ встречается в нем лишь однажды, поэтому первые три символа текста можно пропустить и начать следующее сравнение с четвертого, а не со второго символа текста. Сейчас мы опишем способы использования этой возможности.

[1] Число $T - S + 1$ появляется, поскольку выполнение алгоритма можно остановить, если в тексте остается меньше S непроверенных символов.

[2] Поскольку число символов в алфавитах естественных языков ограничено, а сами символы встречаются крайне неравномерно (например, в английском буква e встречается гораздо чаще, чем z или q), производительность алгоритма на текстах естественного языка гораздо выше наших оценок общего характера.

5.1.1. Конечные автоматы

Конечные автоматы используются для решения задачи о том, принадлежит ли данное слово данному языку. Конечный автомат представляет собой простое устройство, описываемое текущим состоянием и функцией перехода. Функция перехода формирует новое состояние автомата по текущему состоянию и значению очередного входного символа. Некоторые состояния считаются принимающими, и если после окончания ввода автомат оказывается в принимающем состоянии, то входное слово считается принятым.

Конечный автомат, настроенный на данный образец, можно использовать для сравнения с образцом; если автомат переходит в принимающее состояние, то это означает, что образец найден в тексте. Использование конечных автоматов очень эффективно, поскольку такой автомат обрабатывает каждый символ однократно. Поэтому поиск образца с помощью конечного автомата требует не более T сравнений. Теперь встает задача создания конечного детерминированного автомата, отвечающего данной подстроке. Это непросто; существующие алгоритмы работают долго. Поэтому конечные автоматы и не дают общепринятого хорошего решения задачи сравнения с образцом.

5.1.2. Алгоритм Кнута–Морриса–Пратта

При построении конечного автомата для поиска подстроки в тексте легко построить переходы из начального состояния в конечное принимающее состояние: эти переходы помечены символами подстроки (см. рис. 5.2). Проблема возникает при попытке добавить другие символы, которые не переводят в конечное состояние.

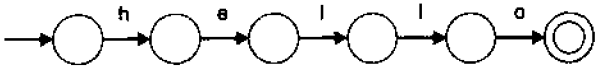


Рис. 5.2 Начало построения автомата для поиска подстроки hello

Алгоритм Кнута-Морриса-Пратта основан на принципе конечного автомата, однако он использует более простой метод обработки неподходящих символов. В этом алгоритме состояния помечаются символами, совпадение с которыми должно в данный момент произойти. Из каждого состояния имеется два перехода: один соответствует успешному сравнению, другой — несовпадению. Успешное сравнение переводит

нас в следующий узел автомата, а в случае несовпадения мы попадаем в предыдущий узел, отвечающий образцу. Пример автомата Кнута–Морриса-Пратта для подстроки ababcb приведен на рис. 5.3.

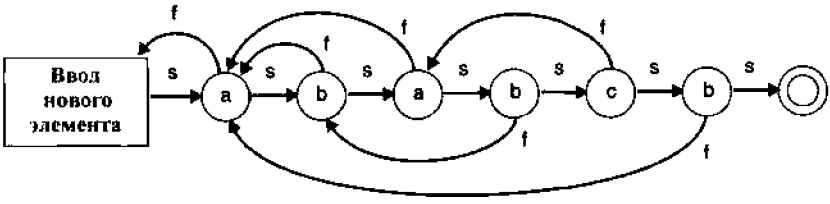


Рис. 5.3. Полный автомат Кнута-Морриса-Пратта для подстроки ababcb

При всяком переходе по успешному сравнению в конечном автомате Кнута–Морриса–Пратта происходит выборка нового символа из текста. Переходы, отвечающие неудачному сравнению, не приводят к выборке нового символа; вместо этого они повторно используют последний выбранный символ. Если мы перешли в конечное состояние, то это означает, что искомая подстрока найдена. Проверьте текст abababcbab на автомате, изображенном на рис. 5.3, и посмотрите, как происходит успешный поиск подстроки.

Вот полный алгоритм поиска:

```
subLoc=1 // указатель тек. сравниваемого символа в подстроке
textLoc=1 // указатель тек. сравниваемого символа в тексте

while textLoc<=length(text) and subLoc<=length(substring) do
   if subLoc=0 or text[textLoc]=substring[subLoc] then
      textLoc=textLoc+1
      subLoc=subLoc+1
   else   // совпадения нет; переход по несовпадению
      subLoc=fail[subLoc]
end while

if (subLoc>length(substring) then
   return textLoc-length(substring)+1 // наиденное совпадение
else
   return 0    // искомая подстрока не найдена
end if
```

Прежде, чем перейти к анализу этого процесса, рассмотрим как задаются переходы по несовпадению. Заметим, что при совпадении ничего особенного делать не надо: происходит переход к следующему узлу. Напротив, переходы по несовпадению определяются тем, как искомая подстрока соотносится сама с собой. Например, при поиске подстроки ababcb нет необходимости возвращаться назад на четыре позиции при необнаружении символа c. Если уж мы добрались до пятого символа в образце, то мы знаем, что первые четыре символа совпали, и поэтому символы ab в тексте, отвечающие третьему и четвертому символам образца, совпадают также с первым и вторым символами образца. Вот как выглядит алгоритм, задающий эти соотношения в подстроке:

```
fail[1]=0
for i=2 to length(substring) do
   temp=fail[i-1]
   while(temp>0) and substring[temp]/=substring[i-1]) do
      temp=fail[temp]
   end while
   fail[i]=temp+1
end for
```

Анализ алгоритма **Кнута–Морриса–Пратта**

Рассмотрим сначала этап создания автомата, т. е. формирование переходов по неудачному сравнению. Цикл while выполняется пока в подстроке не найдется двух совпадающих символов. При беглом взгляде на алгоритм видно, что адрес очередного перехода по несовпадению хранится в переменной temp, и эти адреса идут в убывающем порядке. Это может создать впечатление, что при fc-ой итерации цикла for в цикле while выполняется $k - 1$ сравнение, так что общее число сравнений имеет порядок $S^2/2$. Однако при более внимательном изучении мы найдем лучшее приближение для числа сравнений. Отметим следующие факты:

1) Число неудачных сравнений символов не превосходит $5 - 1$.

2) Все ссылки `fail` меньше своего номера (т. е. `fail`[x] <x для всех x), поскольку они указывают, куда нужно перейти при неудачном сравнении символов.

3) При каждом неудачном сравнении /= значение переменной temp уменьшается в силу утверждения 2.

4) При первом проходе цикла **for** цикл while не выполняется ни разу, поскольку **temp=fail[1]=0**.

5) Последнее утверждение цикла **for**, увеличение переменной *r* и первое утверждение следующей итерации цикла for означают, что переменная temp увеличивается на 1 при всякой последующей итерации цикла **for**.

6) Согласно утверждению 5, в силу того, что число «последующих итераций» цикла **for** равно 5 — 2, переменная temp увеличивается $S - 2$ раз.

7) Поскольку **fail[1]=0**, переменная temp никогда не становится отрицательной.

Мы знаем, что начальное значение переменной temp равно 0 (утверждение 4), и что она увеличивается не более, чем 5 — 2 раз (утверждение 6). Поскольку при всяком неудачном сравнении символов переменная temp уменьшается (утверждение 3) и никогда не становится отрицательной (утверждение 7), общее число неудачных сравнений символов не может превышать 5 — 2. Поэтому суммарное число удачных (утверждение 1) и неудачных сравнений равно 25 — 3. Поэтому построение переходов линейно по длине подстроки.

Рассмотрим теперь алгоритм сравнения. Заметим, что при одном проходе цикла while происходит максимум одно сравнение. При каждом проходе либо увеличиваются обе переменные textLoc и **subLoc**, либо переменная subLoc уменьшается. Начальное значение переменной textLoc равно 1, и ее значение не может превосходить длины T текста, поэтому число увеличений переменных textLoc и subLoc не превосходит T. Начальное значение указателя subLoc также равно 1, этот указатель ни в какой момент не становится отрицательным, и он увеличивается на единицу не более, чем T раз. Поэтому и уменьшаться он может не более, чем T раз. Подводя итог, мы заключаем, что поскольку число увеличений переменных textLoc и subLoc не превосходит T и число уменьшений переменной subLoc не превосходит T, общее число сравнений символов не превосходит $2T$.

Поэтому алгоритм Кнута–Морриса–Пратта выполняет всего $25 + 2T - 3$ сравнений символов, что составляет величину порядка $O(T+S)$ — заметное улучшение по сравнению со стандартным алгоритмом, порядок которого $O(TS)$. Сравнение этих алгоритмов на текстах на естественном языке привело к заключению, что их эффективность

приблизительно одинакова, хотя у алгоритма Кнута–Морриса–Пратта есть некоторое преимущество, которое объясняется тем, что он не осуществляет возврата по тексту.

5.1.3. Алгоритм Бойера-Мура

В отличие от алгоритмов, обсуждавшихся выше, алгоритм Бойера-Мура осуществляет сравнение с образцом справа налево, а не слева направо. Исследуя искомый образец, можно осуществлять более эффективные прыжки в тексте при обнаружении несовпадения.

В примере, изображенном на рис. 5.4 (образец тот же, что и на рис. 5.1), мы сначала сравниваем у с г и обнаруживаем несовпадение. Поскольку мы знаем, что буква r вообще не входит в образец, мы можем сдвинуться в тексте на целых четыре буквы (т. е. на длину образца) вправо. Затем мы сравниваем букву у с h и вновь обнаруживаем несовпадение. Однако поскольку на этот раз h входит в образец, мы можем сдвинуться вправо только на две буквы так, чтобы буквы h совпали. Затем мы начинаем сравнение справа и обнаруживаем полное совпадение кусочка текста с образцом. В алгоритме Бойера–Мура делается 6 сравнений вместо 13 сравнений в исходном простом алгоритме.

Подобное улучшение порождает одну проблему. При сравнении на рис. 5.5 происходит совпадение по буквам k, **n, i,** однако буква h в слове tinkle не совпадает с буквой t в слове think. Если мы ограничимся предложенным улучшением, то придется сдвинуться вправо по тексту всего на один символ, несмотря на то, что совпадение подстрок ink означает, что мы немедленно после сдвига наткнемся на несовпадение, и это несовпадение можно предугадать.

Алгоритм Бойера–Мура обрабатывает образец двумя способами. Во-первых, мы можем вычислить величину возможного сдвига при не-

```
Текст:     there they are
Проход1:   they
Текст:     there they are
Проход2:       they
Текст:     there they are
Проход3:           they
```

Рис. 5.4. Поиск образца they в тексте there they are (совпадение находится после 6 сравнений символов)

```
Текст:    the tinkle of a bell
Образец: think
Текст:    the tinkle of a bell
Образец:      think
```

Рис. 5.5. Проблема со сдвигом

совпадении очередного символа. Во-вторых, мы вычислим величину прыжка, выделив в конце образца последовательности символов, уже появлявшиеся раньше. Прежде, чем заняться подсчетом величины прыжка, посмотрим, как используются результаты этого подсчета.

Алгоритм сравнения

Мы дали общее описание того, как будут использоваться массивы сдвигов и прыжков. Массиве сдвигов содержит величины, на которые может быть сдвинут образец при несовпадении очередного символа. В массиве прыжков содержатся величины, на которые можно сдвинуть образец, чтобы совместить ранее совпавшие символы с вновь совпадающими символами строки. При несовпадении очередного символа образца с очередным символом текста может осуществиться несколько возможностей. Сдвиг в массиве сдвигов может превышать сдвиг в массиве прыжков, а может быть и наоборот. (Совпадение этих величин — простейшая возможная ситуация.) О чем говорят эти возможности? Если элемент массива сдвигов больше, то это означает, что несовпадающий символ оказывается «ближе» к началу, чем повторно появляющиеся завершающие символы строки. Если элемент массива прыжков больше, то повторное появление завершающих символов строки начинается ближе к началу образца, чем несовпадающий символ. В обоих случаях нам следует пользоваться большим из двух сдвигов, поскольку меньший сдвиг неизбежно опять приводит к несовпадению из-за того, что мы знаем о втором значении. Так, например, если значение сдвига равно 2, а значение прыжка 4, то сдвиг на два символа не позволит найти соответствие образцу: несовпадающий символ все равно окажется невыровненным. Однако, если сдвинуть на четыре символа, то под ранее несовпадавшим символом окажется подходящий символ образца, и при этом сохраняется возможность того, что завершающие символы образца будут совпадать с новыми соответствующими символами текста.

Поскольку речь идет только о большем из двух значений, алгоритм имеет следующий вид:

```
textLoc=length(pattern)
patternLoc=length(pattern)
while (textLoc<=length(text)) and (patternLoc>0) do
   if text[textLoc]=pattern[patternLoc] then
      textLoc=textLoc-1
      patternLoc=patternLoc-1
   else
   textLoc=textLoc+MAX(slide[text[textLoc]],jump[patternLoc])
      patternLoc=length(pattern)
   end if
end while

if patternLoc=0 then
   return textLoc+1 // совпадение найдено
else
   return 0
end if
```

Массив сдвигов

Обратимся теперь к массиву сдвигов. На рис. 5.6 (а) мы начинаем сравнение при textLoc=6 и patternLoc=6. Поскольку проверяемые символы совпадают, значения обеих переменных textLoc и patternLoc уменьшаются; следующие два символа тоже совпадают, поэтому уменьшение происходит еще один раз, и значения обеих переменных будут равны 4. В следующем символе происходит несовпадение. Мы хотим сдвинуть образец таким образом, чтобы очередной символ b в тексте совпал с символом b образца, как показано на рис. 5.6 (б). Затем мы заново начинаем процесс сравнения с правого конца образца. Для этого переменной patternLoc следует вновь присвоить значение длины образца, а переменную textLoc следует увеличить на 4 — действительную величину сдвига образца.

Для реализации этих действий определим увеличение переменной textLoc при несовпадении очередных символов. Воспользуемся массивом slide длины, равной числу символов, которые могут появиться в тексте. Начальные значения всех сдвигов полагаем равными длине образца, поскольку обнаружение в тексте символа, отсутствующего в

```
            . textLoc
Текст:   baabacacbacb
Образец: abacac
            . patternLoc
```

(а)

```
                . textLoc
Текст:   baabacacbacb
Образец:   abacac
                . patternLoc
```

(б)

Рис. 5.6. Выбор величины сдвига из массива сдвига

образце, должно привести к сдвигу указателя в тексте на всю длину образца. Затем происходит замена значения для всех символов образца. Если символ появляется в образце неоднократно, то величина сдвига должна быть такой, чтобы совместить с символом в тексте последнее вхождение символа в образце. Совмещение с предыдущими вхождениями будет осуществляться с помощью прыжков, которые мы обсудим ниже. Вот как вычисляются значения сдвига:

```
for каждого символа ch в алфавите do
   slide[ch]=length(pattern)
end for

for i=1 to length(pattern) do
   slide[pattern[i]]=length(pattern)-i
end for
```

Выполнив этот алгоритм на образце datadata, Вы получите slide [d] =3, slide [a]=0 и slide [t]=1; для всех остальных букв алфавита значение сдвига будет равно 8.

Массив прыжков

Массив jump, размер которого совпадает с длиной образца, описывает взаимоотношение частей образца. Этот массив позволяет, напри-

мер, при несовпадении символа h образца с символом t текста на рис. 5.5 сдвинуть образец целиком за сравниваемый символ. Этот новый массив также отслеживает повторение символов в конце образца, которые могут заменять сравниваемые символы. Пусть, например, образец имеет вид abcdbc, и в процессе сравнения два последних символа образца совпали с символами текста, а в третьем символе обнаружилось расхождение. Тогда массив jump говорит, насколько следует сдвинуть образец, чтобы символы bc в позициях 5 и 6 совпали с символами bc в позициях 2 и 3. Таким образом, массив jump содержит информацию о наименьшем возможном сдвиге образца, который совмещает уже совпавшие символы с их следующим появлением в образце.

Предположим, что мы имеем дело с образцом, в котором несовпадение очередного символа означает, что образец следует сдвинуть целиком за место начало сравнения. На рис. 5.7. изображены отрезок текста и образец. Символы X в образце могут быть произвольными; они призваны проиллюстрировать процедуру.

```
Текст:    abcdefghijklmn
Образец: XXXXX
```

Рис. 5.7. Определение величины прыжка

Если образец нужно сдвинуть целиком на всю длину, то символы X должны соотнестись с символами от f до j, т. е. новое значение переменной textLoc будет равно 10. Если несовпадение произошло на символе e, когда textLoc=5, то для сдвига образца нужно к textLoc прибавить 5.

Если же несовпадение произошло на символе d, т. е. при textLoc=4, то для сдвига нужно увеличить textLoc на 6. При несовпадении на символах c, Ъ или a увеличение составит соответственно 7, 8 или 9. В общем случае при несовпадении последнего символа увеличение составляет длину образца, а при несовпадении первого символа — удвоенную длину без единицы. Эти соображения и служат основой для инициализации массива jump.

Посмотрим теперь на образец, в котором некоторые наборы символов с конца образца повторяются. Мы хотим узнать, насколько следует увеличивать указатель textLoc для правильного сдвига образца (не принимая во внимание возможностей, обеспечиваемых массивом slide). Представим для этого, что мы сравниваем образец сам с собой.

Рассмотрим вновь образец abcdbc. Если последний символ в нем не совпадет с соответствующим символом текста, мы можем просто увеличить переменную textLoc на единицу и начать проверку заново. Если последний символ совпал, а следующий с конца нет, то можно сдвинуться на всю длину образца, поскольку каждому вхождению символа b в образец предшествует вхождение символа c. Если же совпали последние два символа, а третий с конца не совпал, то переменную textLoc можно увеличить на 5, чтобы совместить символы **bc**, совпавшие с последними двумя буквами образца, со вторым и третьим символами образца.

Следующий алгоритм вычисляет элементы массива **jump**. Первый цикл инициализирует массив jump. Второй цикл изменяет элементы массива в зависимости от повторений последних символов. Третий и четвертый циклы исправляют максимальные значения сдвигов в начале (соответственно, в конце) массива jump в зависимости от найденных совпадений подстрок образца.

```
// установить максимально возможные значения прыжка
for i=1 to length(pattern) do
   jump[i]=2*length(pattern)-i
end for

// сравнение окончания образца с предыдущими его символами
test=length(pattern)
target=length(pattern)+1
while test>0 do
  link[test]=target
  while target<=length(pattern) and
           pattern[test]/=pattern[target] do
    jump[target]=MIN(jump[target],length(pattern)-test)
    target=link[target]
  end while
  test=test-1
  target=target-1
end while
for i=1 to target do
  jump[i]=MIN(jump[i],length(pattern)+target-i)
end for

temp=link[target]
while target<=length(pattern) do
  while target<=temp do
```

```
    jump[target]=MIN(jump[target],temp-target+length(pattern))
    target=target+1
  end while
  temp=link[temp]
end while
```

На рис. 5.8 приведены состояния массивов jump и link после завершения каждого цикла алгоритма на образце datadata.

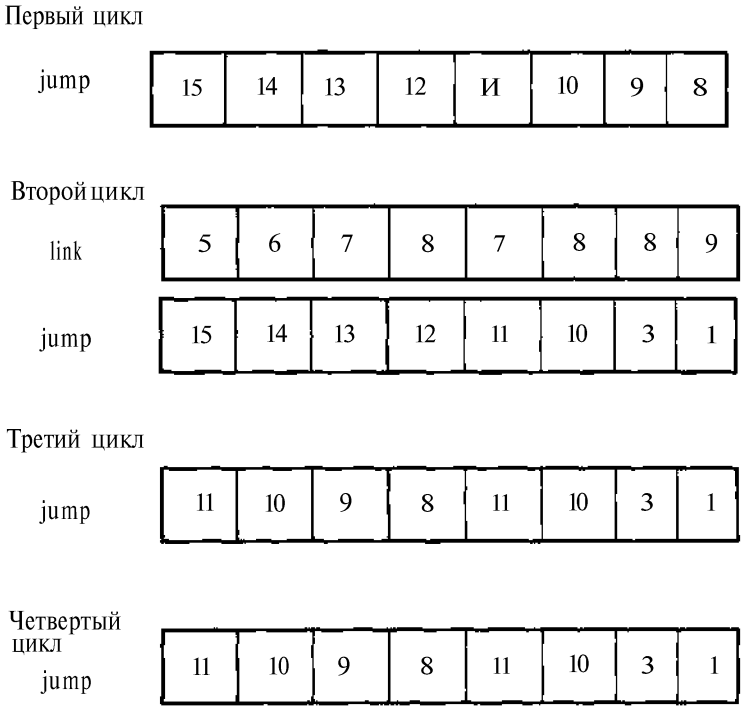


Рис. 5.8. Вычисление элементов массива jump на образце datadata

Анализ

Ниже мы обозначаем через P число символов в образце, через T — длину текста, а через A — число символов в алфавите.

При подсчете массива сдвигов по одному присваиванию приходится на каждый элемент массива и еще по одному - - на каждый символ образца. Поэтому общее число присваиваний равно $O(A + P)$.

При вычислении массива прыжков в худшем случае каждый символ образца будет сравниваться со всеми последующими его символами. Ва-

шего опыта уже достаточно, чтобы подсчитать, что число сравнений будет $O(P^2)$. Каждое сравнение может привести к одному присваиванию, поэтому число присваиваний может также быть $O(P^2)$.

Подробный анализ числа сравнений в основном алгоритме выходит за рамки настоящей книги. Исследования показывают, что при поиске образца из шести или более букв в тексте на естественном языке каждый символ образца сравнивается с 40% или менее символов текста. При возрастании длины образца этот показатель падает до 25%.

5.1.4. Упражнения

1) Постройте переходы по несовпадению в автомате Кнута–Морриса–Пратта для образцов
   а) ABABBC; б) ABCABC; в) CBCBBACA; г) BBABBCAC.

2) Мы говорили в тексте о том, что наихудшим образцом для стандартного алгоритма служит строка, состоящая из символов X и заканчивающаяся символом Y, причем наихудший случай достигается на тексте, состоящем из одних символов X. Как мы видели, на образце, состоящем из S символов (первые $S - 1$ из них это X, а последний Y), стандартный алгоритм выполняет ST сравнений символов. Как выглядят переходы по несовпадению для строки XXXXY и сколько сравнений нужно для их построения? Как выглядят переходы по несовпадению для общего образца того же вида и сколько сравнений нужно для их построения? Сколько сравнений символов выполнит алгоритм Кнута–Морриса–Пратта при поиске образца в тексте? (Опишите действия, с помощью которых Вы получили ответ.)

3) Подсчитайте значения элементов массива сдвигов в алгоритме Бойера-Мура для приведенных ниже образцов. Предположите для простоты, что Вы работаете в алфавите {A,B,C,D,E}.

   а) ABABBC; б) ABCABC; в) CBCBBACA; г) BBABBCAC.

4) Подсчитайте значения элементов массива прыжков в алгоритме Бойера–Мура для приведенных ниже образцов.

   а) ABABBC; б) **ABCABC**; в) **CBCBBACA**; г) BBABBCAC.

5.2. Приблизительное сравнение строк

Причины, по которым сравнение образца и подстроки текста оказалось неудачным, принято разбивать на классы. Отличие может состоять в том, что соответствующие символы образца и подстроки оказались различными, что в образце есть символы, отсутствующие в тексте. что в тексте есть символы, отсутствующие в образце. Ошибки набора, как правило, относятся к одной из этих трех категорий, причем ошибка, заключающаяся в перестановке двух соседних букв интерпретируется как два отличия первого типа.

Обычно мы будет искать соответствие подстроке с точностью k, где через k обозначено максимальное число отличий, упомянутых в предыдущем абзаце. Нам придется учесть несколько возможностей. Что означает, например, несовпадение первого символа строки-образца и текста? Оно может означать, как несовпадение символов, так и отсутствие нужного символа в строке или тексте. Даже если символы совпадают, может оказаться, что лучшее совпадение строк в целом достигается, если считать первый символ образца или текста пропущенным.

Попробуем, например, сравнить строку ad со строкой read. При сравнении с начала текста мы получаем два возможных 2-приближенных совпадения (либо символ a следует заменить на r, а символ d — на e, либо нужно добавить к образцу пропущенные впереди символы `re`). Кроме того при сравнении с первой позиции есть 3-приближенное совпадение (добавить символ `r` и заменить ad на ea). При сравнении со второй позиции имеется 2-приближенное совпадение (заменить ad на `ea`), а также 1-приближенное совпадение (добавить впереди символ e).

Видно, что число возможностей очень велико. Даже при совпадении нескольких первых символов, за которым следует несовпадение, может оказаться, что лучшее приближение достигается при изменении некоторых символов в образце или в тексте или добавлении тех или иных символов туда и сюда. Как найти все возможности, сохранив при этом разумную структуру данных и алгоритма? Сложность алгоритма, проверяющего все возможности, чересчур высока. Мы предпочитаем упростить алгоритм за счет усложнения структур данных.

Будем решать задачу путем создания матрицы с именем `diffs`, в которой накапливается вся до сих пор полученная информация. Каждому символу образца соответствует строка матрицы, а каждому символу текста — столбец. Значения элементов матрицы показывают, насколько похожи текст и образец к данному моменту. Скажем, если на пере-

сечении пятой строки и 27-го столбца стоит число 4, то это означает, что при сравнении первых пяти символов образца с отрезком текста, кончающимся 27-ым символом, мы обнаружили четыре расхождения.

Число расхождений в произвольной позиции определяется тремя соседними элементами матрицы, находящимися непосредственно сверху, слева и сверху-слева от данного. При использовании верхнего соседа мы предполагаем, что в тексте пропущен элемент образца. При использовании левого соседа предполагается, что в образце пропущена буква из текста. Сосед по диагонали отвечает за совпадение или несовпадение символов. Точнее говоря, клетку diffs [i, j] мы заполняем минимумом из трех значений[3]:

1. `diffs[i-1,j-1]` , если `substring[i]=text[j]` , в противном случае `diffs[i-1,j-1]+1` ;
2. `diffs[i-1,j]+1` (символ `substring[j]` отсутствует в тексте);
3. `diffs [i,j-1]+1` (символ substring [j] отсутствует в образце).

При запуске процесса мы обращаемся к строке $i = 0$, лежащей вне матрицы, и полагаем значения ее элементов равными нулю, а также к лежащему вне матрицы столбцу $j = 0$, и полагаем значения его элементов равными номеру соответствующей строки. Пример заполнения для образца trim и текста try the trumpet приведен на рис. 5.9.

		t	r	y		t	h	e		t	r	u	m	p	e	t
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
t	1	0	1	1	1	0	1	1	1	0	1	1	1	1	1	0
r	2	1	0	1	2	1	1	2	2	1	0	1	2	2	2	1
i	3	2	1	1	2	2	2	2	3	2	1	1	2	3	3	2
m	4	3	2	2	2	3	3	3	3	3	2	2	1	2	3	3

Рис. 5.9. Матрица diffs для образца trim и текста try the trumpet

При взгляде на последний элемент столбца y мы видим число 2; это означает, что выравнивание образца trim так, чтобы он кончался в символе y текста, требует двух изменений обсуждавшихся выше типов. Эти два изменений касаются удаления буквы m образца, отсутствующей в тексте за y, а также несовпадение y с i, *либо* отсутствие i перед y

[3]Обратите внимание на то, что элемент diffs `[i-1,j-1]` находится сверху слева, элемент `diffs[i-1,j]` — непосредственно сверху, а элемент `diffs[i,j-1]` — непосредственно слева от заполняемого.

в тексте и несовпадение y с m. Поэтому в последней строке собраны наилучшие возможные совпадения образца с подстрокой текста, кончающейся данным символом. Из таблицы ясно, что наилучшее совпадение образца с текстом происходит в начале trum слова trumpet: единственное отличие наблюдается между буквами i и и.

При реализации этой процедуры мы должны были бы задать не только образец и текст, но и переменную для хранения минимального числа различий. Алгоритм заполнял бы матрицу столбец за столбцом, пока нижний элемент столбца не превышает установленной границы. Это означает, что алгоритм не обязан хранить в памяти все ST элементов матрицы (где S — длина образца, а T — длина текста), а может обойтись $2S$ ячейками для хранения целых чисел: достаточно хранить вычисляемый столбец и предыдущий, которым он однозначно определяется.

Такой тип алгоритмов относится к «динамическому программированию», к которому мы вернемся в главе 9.

5.2.1. Анализ

Природа матрицы позволяет легко проанализировать алгоритм. Мы делаем по одному сравнению на каждый элемент матрицы. Поэтому число сравнений в наихудшем случае равно ST. Заметим, что даже при реализации всех возможных различий этот процесс выполняется со скоростью непосредственного поиска точного соответствия образцу.

5.2.2. Упражнения

1) Постройте матрицу приблизительного соответствия для образца their и текста hello there friends.

2) Постройте матрицу приблизительного соответствия для образца where и текста were they here.

3) При поиске точного совпадения начальную точку легко обнаружить, поскольку начинать поиск мы должны за S символов от конца текста. При поиске приблизительного соответствия начало отсчета найти не так-то легко, потому что символы могут быть пропущены как в тексте, так и в образце. Дополните изложенную выше процедуру описанием дополнительных структур данных и действий, необходимых для остановки работы, если будет найдено k-приблизительное соответствие. *(Указание:* Например, один из

способов проверить, что структура скобок в выражении регулярна, состоит в том, чтобы завести счетчик, к значению которого добавляется 1 при появлении очередной открывающей скобки, и из которого вычитается 1 при появлении закрывающей скобки; при чтении любого другого символа значение счетчика не изменяется. Можно ли отследить пропущенные символы похожим образом?)

5.3. Упражнения по программированию

Вы можете обзавестись текстом большого объема для проверки решений нижеследующих задач, сохранив написанный Вами семестровый отчет в формате «только текст». Специальные случаи можно проверять на словах, не входящих в текст. Такими словами наверняка окажутся названия растения или цветка, города или цвета. Будьте при этом осторожны в выборе слова: короткое слово может оказаться частью более длинного. Скажем, red входит составной частью в bothered. Можно, например, добавить в середину текста слово banyan, а в конец — слово potato, и искать их.

1) Запрограммируйте алгоритм Кнута–Морриса–Пратта и подсчитайте число сравнений символов для нескольких различных случаев. Не забывайте про сравнения, необходимые при вычислении переходов по несовпадению. Проверьте как длинные, так и короткие образцы. Выходом Вашей программы должны служить номер символа текста (расстояние от начала), где начинается совпадение с образцом, а также сделанное число сравнений. Как соотносятся Ваши результаты с проведенным в книге анализом?

2) Запрограммируйте алгоритм Бойера–Мура и подсчитайте число сравнений символов для нескольких различных случаев. Не забывайте про сравнения, необходимые при вычислении массивов сдвигов и прыжков. Проверьте как длинные, так и короткие образцы. Выходом Вашей программы должны служить номер символа текста (расстояние от начала), где начинается совпадение с образцом, а также сделанное число сравнений. Как соотносятся Ваши результаты с проведенным в книге анализом?

Глава 6.

Алгоритмы на графах

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- описывать множества и отношение принадлежности к множеству;
- пользоваться двумерными массивами;
- пользоваться структурами данных «стеки» и «очереди»;
- пользоваться списками со ссылками;
- описывать скорость и порядок роста функций.

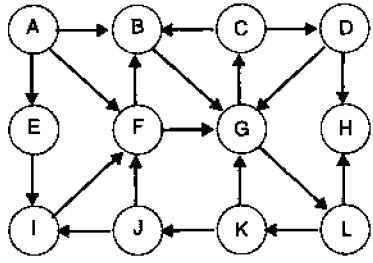
ЦЕЛИ

Освоив эту главу, Вы должны уметь

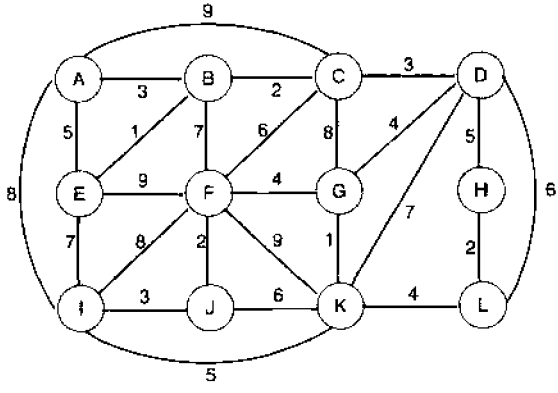
- определять понятия, связанные с графами;
- создавать структуры данных, описывающие графы;
- обходить граф по уровням и в глубину;
- находить минимальное остовное дерево в связном графе;
- находить кратчайший путь между двумя вершинами связного графа;
- находить компоненты двусвязности связного графа.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Научитесь обходить по уровням и в глубину, начиная с вершины A, следующий граф:



Выполните трассировку на следующем графе алгоритма Дейкстры-Прима, строящего минимальное остовное дерево, начиная с вершины A, алгоритма Крускала, также строящего минимальное остовное дерево, и алгоритма Дейкстры для поиска кратчайшего пути:



Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

Графы формально описывают множество близких ситуаций. Самым привычным примером служит карта автодорог, на которой изображены перекрестки и связывающие их дороги. Перекрестки являются вершинами графа, а дороги — его ребрами. Иногда наши графы ориентированы (подобно улицам с односторонним движением) или взвеше-

ны — каждой дороге приписана стоимость путешествия по ней (если, например, дороги платные). Когда мы изучим язык графов подробнее, аналогия с картой автодорог станет еще более глубокой.

После освоения математических аспектов графов мы займемся вопросами их представления в памяти компьютера и алгоритмами их обработки. Мы увидим, что имеется много способов хранения графов, отличающихся по величине накладных расходов; выбор способа может зависеть от самого графа.

Иногда приходится распространять ту или иную информацию среди большой группы людей или по всем компьютерам большой сети. Мы хотели бы, чтобы информация достигла каждого участника группы, но при этом ровно по одному разу. В некоторых группах для этой цели организуется «телефонное дерево», когда каждый из членов группы, получив свежую новость, сообщает ее небольшому числу других участников. Если каждый из членов группы встречается в дереве лишь однажды, а высота дерева не очень велика, то информация очень быстро доходит до всех. Для графов общего вида ситуация оказывается сложнее: в среднем графе гораздо больше связей, чем в дереве. Мы изучим два метода обхода графов: в глубину и по уровням, позволяющих преодолеть эту трудность. Остовное дерево представляет собой связное подмножество графа, не содержащее циклов, включающее в себя все вершины графа и некоторые из его ребер. Минимальное остовное дерево это остовное дерево, имеющее минимально возможную сумму весов ребер. Одно из применений минимальных остовных деревьев — организация внутренней компьютерной сети. Передающие станции устанавливаются в стратегически важных местах некоторой области. Если мы хотим уменьшить суммарную стоимость объединения станций в сеть, то можно нарисовать граф, в котором станции будут служить вершинами, а ребрам, их соединяющим, можно приписать стоимость соединения. Минимальное остовное дерево этого графа указывает, какие станции следует соединить между собой, чтобы любые две станции оказались соединенными, причем общая стоимость соединения была минимально возможной.

Аналогичные приложения имеет и задача поиска кратчайшего пути в графе: умение решать такую задачу помогает планировать путь на автомобиле или посылать сообщение по компьютерной сети.

Важной характеристикой большой компьютерной сети служит ее надежность. Мы хотели бы, чтобы сеть сохраняла работоспособность при выходе из строя одного узла. Говоря проще, станции в сети должны

быть соединены несколькими путями, чтобы при разрушении какого-либо из путей возможность передачи информации сохранялась. В последнем параграфе этой главы мы обсуждаем алгоритм поиска компонент двусвязности. Он ищет вершины, неизбежно находящиеся на всяком пути из одной части графа в другую. В компьютерной сети разрушение таких узлов приводит к нарушению связности сети.

6.1. Основные понятия теории графов

С формальной точки зрения граф представляет собой упорядоченную пару $G = (V, E)$ множеств, первое из которых состоит из вершин, или узлов, графа, а второе — из его ребер. Ребро связывает между собой две вершины. При работе с графами нас часто интересует, как проложить путь из ребер от одной вершины графа к другой. Поэтому мы будем говорить о движении по ребру; это означает, что мы переходим из вершины A графа в другую вершину B, связанную с ней ребром AB (ребро графа, связывающее две вершины, для краткости обозначается этой парой вершин). В этом случае мы говорим, что A примыкает к B, или что эти две вершины соседние.

Граф может быть ориентированным или нет. Ребра неориентированного графа, чаще всего называемого просто графом, можно проходить в обоих направлениях. В этом случае ребро — это неупорядоченная пара вершин, его **концов**[1]. В ориентированном графе, или орграфе, ребра представляют собой упорядоченные пары вершин: первая вершина — это начало ребра, а вторая — его **конец**[2]. Далее мы для краткости будем говорить просто о ребрах, а ориентированы они или нет будет понятно из контекста.

Позже мы будем просто рисовать графы, а не задавать их множествами. Вершины будут изображаться кружочками, а ребра — отрезками линий. Внутри кружочков будут записаны метки вершин. Ребра ориентированного графа будут снабжены стрелками, указывающими допустимое направление движения по ребру.

На рис. 6.1 изображено графическое представление неориентированного (а) и ориентированного (б) графов вместе с их формальным описанием.

[1] Если концы ребра совпадают, то речь идет о петле — ребре, соединяющем вершину с самой собой.

[2] **Начало** и конец ребра в ориентированном графе также могут совпадать.

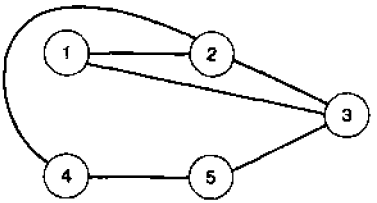


Рис. 6.1 (а). Граф $G = (\{1,2,3,4,5\},\{\{1,2\},\{1,3\},\{2,4\},\{3,5\},\{2,3\},\{4,5\}\})$

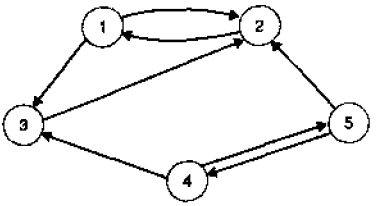


Рис. 6.1 *(б)*. Ориентированный граф $G = (\{1, 2, 3, 4, 5\}, \{(1, 2), (1, 3), (1, 3), (2, 1), (3, 2), (4, 3), (4,5), (5,2), (5,4)\})$

6.1.1. Терминология

Полный граф — это граф, в котором каждая вершина соединена со всеми остальными. Число ребер в полном графе без петель с N вершинами равно $(N^2 - N)/2$. В полном ориентированном графе разрешается переход из любой вершины в любую другую. Поскольку в графе переход по ребру разрешается в обоих направлениях, а переход по ребру в орграфе — только в одном, в полном орграфе в два раза больше ребер, т.е. их число равно N^2 — N.

Подграф (V_s, E_s) графа или орграфа (V, E) состоит из некоторого подмножества вершин, V_s С V, и некоторого подмножества ребер, их соединяющих, E_s С E.

Путь в графе или орграфе — это последовательность ребер, по которым можно поочередно проходить. Другими словами, путь из вершины A в вершину B начинается в А и проходит по набору ребер до тех пор, пока не будет достигнута вершина B. С формальной точки зрения, путь из вершины Vi в вершину Vj это последовательность ребер графа v_iv_{i+1}, $v_{i+1}v_{i+2}$,. . ., $v_{j-1}v_j$.Мы требуем, чтобы любая вершина встречалась на таком пути не более, чем однажды. У всякого пути есть длина — число ребер в нем. Длина пути AB, BC, CD, DE равна 4.

Во взвешенном графе или орграфе каждому ребру приписано число, называемое весом ребра. При изображении графа мы будем записывать вес ребра рядом с ребром. При формальном описании вес будет дополнительным элементом неупорядоченной или упорядоченной пары вершин (образуя вместе с этой парой «триплет»). При работе с ориентированными графами мы считаем вес ребра ценой прохода по нему. Стоимость пути по взвешенному графу равна сумме весов всех ребер пути. Кратчайший путь во взвешенном графе — это путь с минималь-

ным весом, даже если число ребер в пути и можно уменьшить. Если, например, путь P_1 состоит из пяти ребер с общим весом 24, а путь P_2 — из трех ребер с общим весом 36, то путь P_1 считается более коротким.

Граф или орграф называется связным, если всякую пару узлов можно соединить по крайней мере одним путем. Цикл — это путь, который начинается и кончается в одной и той же вершине. В ациклическом графе или орграфе циклы отсутствуют. Связный ациклический граф называется (неукорененным) деревом. Структура неукорененного дерева такая же, что и у дерева, только в нем не выделен корень. Однако каждая вершина неукорененного дерева может служить его корнем.

6.1.2. Упражнения

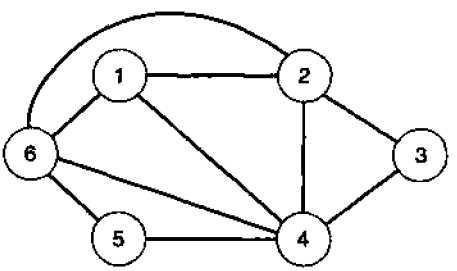
1) Нарисуйте следующий граф:

$$G = (\{1,2,3,4,5,6\}, \{\{1,2\},\{1,4\},\{2,5\},\{2,6\},\{3,4\},\{3,5\},\{3,6\},\{4,5\},\{4,6\},\{5,6\}\}).$$

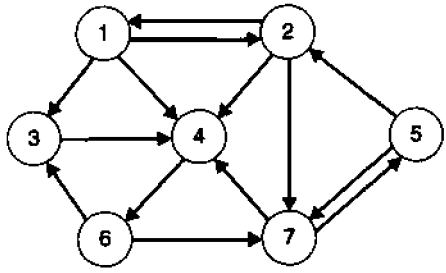
2) Нарисуйте следующий ориентированный граф:

$$G = (\{1,2,3,4,5\}, \{\{1,2\},\{1,4\},\{1,5\},\{2,3\},\{2,4\},\{2,5\},\{3,2\},\{3,4\},\{3,5\},\{4,1\},\{4,2\},\{4,5\},\{5,2\},\{5,3\},\{5,4\}\}).$$

3) Опишите следующий граф:



4) Опишите следующий ориентированный граф:



5) Составьте список всех путей из вершины 1 в вершину 5 в графе из упражнения 3).

6) Составьте список всех путей из вершины 1 в вершину 4 в орграфе из упражнения 4).

7) Составьте список всех циклов с началом в вершине 3 в графе из упражнения 3).

8) Составьте список всех циклов с началом в вершине 7 в орграфе из упражнения 4).

6.2. Структуры данных для представления графов

Информацию о графах или орграфах можно хранить двумя способами: в виде матрицы примыканий или в виде списка примыканий. В настоящем параграфе графы и орграфы представляются одинаково, поэтому мы будем использовать для них общий термин «граф». Мы увидим также, что применяемые методы хранения информации устраняют различие в обработке графов и орграфов.

Матрица примыканий обеспечивает быстрый доступ к информации о ребрах графа, однако если в графе мало ребер, то эта матрица будет содержать гораздо больше пустых клеток, чем заполненных. Длина списка примыканий пропорциональна числу ребер в графе, однако при пользовании списком время получения информации о ребре увеличивается.

Ни один из этих методов не превосходит другой заведомо. В основе выбора подходящего метода должны лежать наши предварительные знания о графах, которые будут обрабатываться алгоритмом. Если в графе много вершин, причем каждая из них связана лишь с небольшим

количеством других вершин, список примыканий оказывается выгоднее, поскольку он занимает меньше места, а длина просматриваемых списков ребер невелика. Если же число вершин в графе мало, то лучше воспользоваться матрицей примыканий: она будет небольшой, и даже потери при хранении в матричном виде разреженного графа будут незначительны. Если же в графе много ребер и он почти полный, то матрица примыканий всегда является лучшим способом хранения графа.

Использование матрицы и списка примыканий подробно описано в нижеследующих разделах.

6.2.1. Матрица примыканий

Матрица примыканий `AdjMat` графа $G = (V, E)$ *с* числом вершин $|V| = N$ записывается в виде двумерного массива размером N x N. В каждой ячейке $[i, j]$ этого массива записано значение 0 за исключением лишь тех случаев, когда из вершины v_i в вершину Vj ведет ребро, и тогда в ячейке записано значение 1. Говоря более строго,

$$\mathrm{AdjMat}[i,j] = \begin{cases} 1, & \text{если } v_i v_j \in E \\ 0, & \text{если } v_i v_j \notin E \end{cases} \quad \text{для всех } i \text{ и } j \text{ от 1 до } N.$$

На рис. 6.2 изображены матрицы примыканий для графа и орграфа, изображенных на рис. 6.1.

Ячейка матрицы примыканий взвешенного графа или орграфа содержит ∞, если соответствующее ребро отсутствует, а во всех осталь-

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

Рис. 6.2 *(а)*. Матрица примыканий для графа с рис. 6.1 (а)

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

Рис. 6.2 *(б)*. Матрица примыканий для ориентированного графа с рис. 6.1 (б)

ных случаях ее значение равно весу ребра. Диагональные элементы такой матрицы равны 0, поскольку путешествие из вершины в нее саму не стоит ничего.

6.2.2. Список примыканий

Список примыканий `AdjList` графа $G = (V, E)$ с числом вершин $\backslash V \backslash = N$ записывается в виде одномерного массива длины N, каждый элемент которого представляет собой ссылку на список. Такой список приписан каждой вершине графа, и он содержит по одному элементу на каждую вершину графа, соседнюю с данной.

На рис. 6.3 изображены списки примыканий для графа и орграфа, изображенных на рис. 6.1.

Элементы списка примыканий для взвешенного графа содержат дополнительное поле, предназначенное для хранения веса ребра.

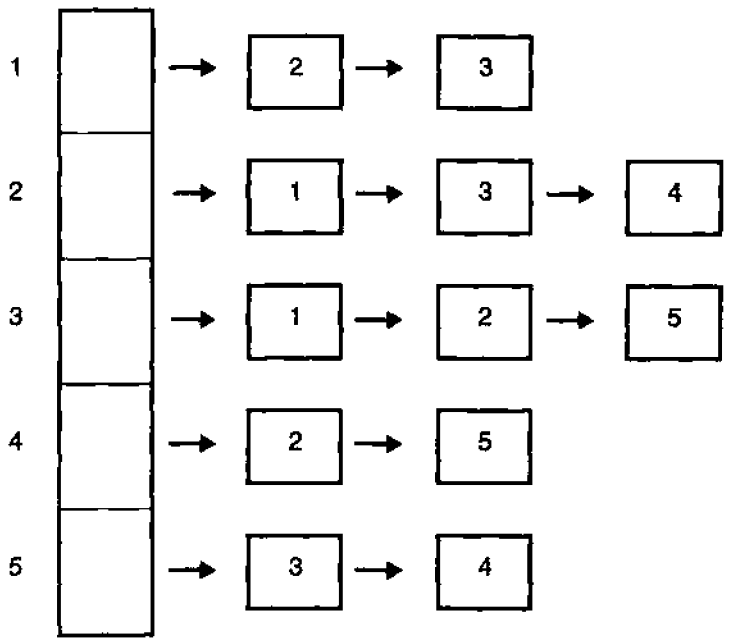


Рис. 6.3 (a). Список примыканий для графа с рис. 6.1 (а)

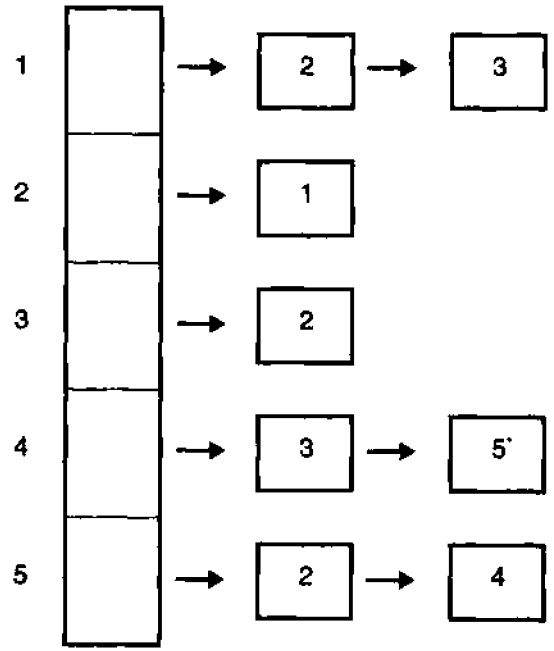


Рис. 6.3 (б). Список примыканий для графа с рис. 6.1 (б)

6.2.3. Упражнения

1) Выпишите матрицу примыканий для графа из упражнения 1) раздела 6.1.2.

2) Выпишите матрицу примыканий для орграфа из упражнения 2) раздела 6.1.2.

3) Выпишите матрицу примыканий для графа из упражнения 3) раздела 6.1.2.

4) Выпишите матрицу примыканий для орграфа из упражнения 4) раздела 6.1.2.

5) Выпишите список примыканий для графа из упражнения 1) раздела 6.1.2.

6) Выпишите список примыканий для орграфа из упражнения 2) раздела 6.1.2.

7) Выпишите список примыканий для графа из упражнения 3) раздела 6.1.2.

8) Выпишите список примыканий для орграфа из упражнения 4) раздела 6.1.2.

6.3. Алгоритмы обхода в глубину и по уровням

При работе с графами часто приходится выполнять некоторое действие по одному разу с каждой из вершин графа. Например, некоторую порцию информации следует передать каждому из компьютеров в сети. При этом мы не хотим посещать какой-либо компьютер дважды. Аналогичная ситуация возникает, если мы хотим собрать информацию, а не распространить ее.

Подобный обход можно совершать двумя различными способами. При обходе в глубину проход по выбранному пути осуществляется настолько глубоко, насколько это возможно, а при обходе по уровням мы равномерно двигаемся вдоль всех возможных направлений. Изучим теперь эти два способа более подробно. В обоих случаях мы выбираем одну из вершин графа в качестве отправной точки. Ниже под «посещением узла» мы понимаем выполнение действия, которое необходимо совершить в каждой вершине. Если, например, идет поиск, то фраза о том, что мы посетили данный узел, означает, что мы проверили его на наличие требуемой информации. Описываемые методы работают без каких-либо изменений как на ориентированных, так и на неориентированных графах. Мы будем иллюстрировать их на неориентированных графах.

С помощью любого из этих методов можно проверить, связен ли рассматриваемый граф. При обходе графа можно составлять список пройденных узлов, а по завершении обхода составленный список можно сравнить со списком всех узлов графа. Если списки содержат одни и те же элементы, то граф связен. В противном случае в графе есть вершины, до которых нельзя дойти из начальной вершины, поэтому он несвязен.

6.3.1. Обход в глубину

При обходе в глубину мы посещаем первый узел, а затем идем вдоль ребер графа, пока не упремся в тупик. Узел неориентированного графа является тупиком, если мы уже посетили все примыкающие к нему узлы. В ориентированном графе тупиком также оказывается узел, из которого нет выходящих ребер.

После попадания в тупик мы возвращаемся назад вдоль пройденного пути пока не обнаружим вершину, у которой есть еще не посещенный сосед, а затем двигаемся в этом новом направлении. Процесс оказывается завершенным, когда мы вернулись в отправную точку, а все примыкающие к ней вершины уже оказались посещенными.

Иллюстрируя этот и другие алгоритмы обхода, при выборе одной из двух вершин мы всегда будем выбирать вершину с меньшей (в числовом или лексикографическом порядке) меткой. При реализации алгоритма выбор будет определяться способом хранения ребер графа.

Рассмотрим граф с рис. 6.4. Начав обход в глубину в вершине 1, мы затем посетим последовательно вершины 2, 3, 4, 7, 5 и 6 и упремся в тупик. Затем нам придется вернуться в вершину 7 и обнаружить, что вершина 8 осталась непосещенной. Однако перейдя в эту вершину, мы немедленно вновь оказываемся в тупике. Вернувшись в вершину 4, мы обнаруживаем, что непосещенной осталась вершина 9; ее посещение вновь заводит нас в тупик. Мы снова возвращаемся назад в исходную вершину, и поскольку все соседние с ней вершины оказались посещенными, обход закончен.

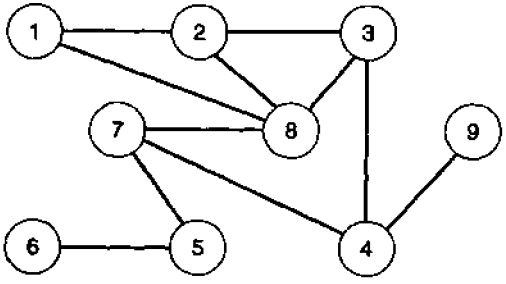


Рис. 6.4. Пример графа

Вот как выглядит рекурсивный алгоритм обхода в глубину:

```
DepthFirstTraversal(G,v)
G   граф
v   текущий узел
Visit(v)
Mark(v)
for каждого ребра vw графа G do
   if вершина w непомечена then
      DepthFirstTraversal(G,w)
   end if
end for
```

Этот рекурсивный алгоритм использует системный стек для отслеживания текущей вершины графа, что позволяет правильно осуществить возвращение, наткнувшись на тупик. Вы можете построить аналогичный нерекурсивный алгоритм самостоятельно, воспользовавшись стеком для занесения туда и удаления пройденных вершин графа.

6.3.2. Обход по уровням

При обходе графа по уровням мы, после посещения первого узла, посещаем все соседние с ним вершины. При втором проходе посещаются все вершины «на расстоянии двух ребер» от начальной. При каждом новом проходе обходятся вершины, расстояние от которых до начальной на единицу больше предыдущего. В графе могут быть циклы, поэтому не исключено, что одну и ту же вершину можно соединить с начальной двумя различными путями. Мы обойдем эту вершину впервые, дойдя до нее по самому короткому пути, и посещать ее второй раз нет необходимости. Поэтому, чтобы предупредить повторное посещение, нам придется либо вести список посещенных вершин, либо завести в каждой вершине флажок, указывающий, посещали мы ее уже или нет.

Вернемся к графу с рис. 6.4. Начав обход с вершины 1, на первом проходе мы посетим вершины 2 и 8. На втором проходе на нашем пути окажутся вершины 3 и 7. (Вершины 2 и 8 также соединены с исходной вершиной путями длины два, однако мы не вернемся в них, поскольку уже побывали там при первом проходе.) При третьем проходе мы обойдем вершины 4 и 5, а при последнем — вершины 6 и 9.

В основе обхода в глубину лежала стековая структура данных, а при обходе по уровням мы воспользуемся очередью. Вот как выглядит алгоритм обхода по уровням:

```
BreadthFirstTraversal(G,v)
G   граф
v   текущий узел
Visit(v)
Mark(v)
Enqueue(v)
while очередь непуста do
   Dequeue(x)
   for каждого ребра xw в графе G do
      if вершина w непомечена then
         Visit(w)
         Mark(w)
         Enqueue(w)
      end if
   end for
end while
```

Этот алгоритм заносит в очередь корень дерева обхода по уровням, но затем немедленно удаляет его из очереди. При просмотре соседних с корнем вершин он заносит их в очередь. После посещения всех соседних с корнем вершин происходит возвращение к очереди и обращение к первой вершине оттуда. Обратите внимание на то, что поскольку узлы добавляются к концу очереди, ни одна из вершин, находящихся на расстоянии двух ребер от корня, не будет рассмотрена повторно, пока не будут обработаны и удалены из очереди все вершины на расстоянии одного ребра от корня.

6.3.3. Анализ алгоритмов обхода

При разработке алгоритмов обхода мы стремились к тому, чтобы посещать каждую вершину связного графа в точности один раз. Посмотрим сначала, удалось ли нам этого достичь. При обходе по уровням мы начинали с исходной вершины и шли по всем доступным ребрам, если только вершина на втором конце ребра не оказывалась уже посещенной. Приводит ли это к каким-либо трудностям? Любой из узлов, в который можно попасть из данного, окажется посещенным — но вдоль самого прямого пути. Вернувшись к графу на рис. 6.4, мы замечаем, что мы не возвращались к узлу 8 после посещения узлов 2 и 3. Однако все те узлы, до которых можно добраться из вершины 8, оказываются посещенными, поскольку мы дошли до этой вершины на более раннем

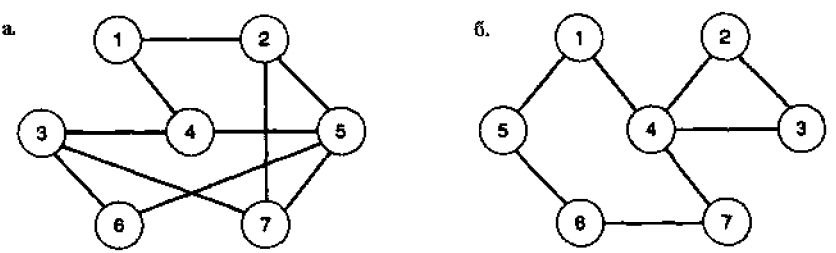
проходе. С другой стороны, может ли в связном графе оказаться узел, до которого мы не дошли? Поскольку при каждом проходе мы делаем один шаг от вершин, посещенных на предыдущем проходе, узел может оказаться непосещенным только, если он не является соседом никакого из посещенных узлов. Но это означало бы, что граф несвязен, что противоречит нашему предположению; значит мы посетим все узлы.

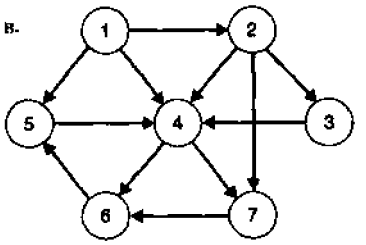
Аналогичное рассуждение справедливо и для обхода в глубину. В этом случае мы заходим вглубь графа пока не наткнемся на тупик. Посетим ли мы, возвращаясь, все остальные узлы? Может ли так получиться, что в какую-то из вершин графа мы не зайдем? При каждом попадании в тупик мы возвращаемся до первого узла, из которого есть проход к еще не посещенному узлу. Тогда мы следуем этим проходом и снова идем по графу в глубину. Встретив новый тупик, мы вновь возвращаемся назад, пока не наткнемся на первую вершину с еще не посещенным соседом. Чтобы одна из вершин графа осталась непосещенной, нужно, чтобы она не была соседней ни для одной из посещенных вершин. А это и означает, что граф несвязен вопреки нашим предположениям. При обходе в глубину мы также должны посетить все узлы.

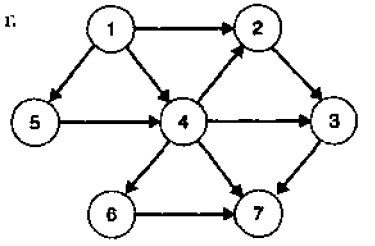
Что можно сказать про эффективность такого алгоритма? Будем предполагать, что основная часть всей работы приходится на посещение узла. Тогда расходы на проверку того, посещали ли мы уже данный узел, и на переход по ребру можно не учитывать. Поэтому порядок сложности алгоритма определяется числом посещений узлов. Как мы уже говорили, каждая вершина посещается в точности один раз, поэтому на графе с N вершинами происходит N посещений. Таким образом, сложность алгоритма равна $O(N)$.

6.3.4. Упражнения

1) Для следующих графов укажите порядок посещения вершин при обходе в глубину начиная с вершины 1.







2) Для графов из упражнения 1) укажите порядок посещения вершин при обходе по уровням начиная с вершины 1.

3) Напишите подробный алгоритм обхода в глубину, использующий представление графа с помощью матрицы примыканий; при посещении очередной вершины алгоритм должен просто печатать ее метку. Проверьте алгоритм на примерах графов, приведенных в этом разделе, и убедитесь, что он дает те же результаты.

4) Напишите подробный алгоритм обхода по уровням, использующий представление графа с помощью матрицы примыканий; при посещении очередной вершины алгоритм должен просто печатать ее метку. Проверьте алгоритм на примерах графов, приведенных в этом разделе, и убедитесь, что он дает те же результаты.

5) Напишите подробный алгоритм обхода в глубину, использующий представление графа с помощью списка примыканий; при посещении очередной вершины алгоритм должен просто печатать ее метку. Проверьте алгоритм на примерах графов, приведенных в этом разделе, и убедитесь, что он дает те же результаты.

6) Напишите подробный алгоритм обхода по уровням, использующий представление графа с помощью списка примыканий; при посещении очередной вершины алгоритм должен просто печатать ее метку. Проверьте алгоритм на примерах графов, приведенных в этом разделе, и убедитесь, что он дает те же результаты.

7) Докажите, что в связном графе каждое ребро либо входит в дерево обхода в глубину, либо указывает на предшественника в дереве.

8) Докажите, что в связном графе каждое ребро либо входит в дерево обхода по уровням, либо указывает на вершину, которая не является ни предшественником, ни потомком данной вершины в дереве.

6.4. Алгоритм поиска минимального остовного дерева

Минимальным остовным деревом (МОД) связного взвешенного графа называется его связный подграф, состоящий из всех вершин исходного дерева и некоторых его ребер, причем сумма весов ребер минимально возможная. Если исходный граф несвязен, то описываемую ниже процедуру можно применять поочередно к каждой его компоненте связности, получая тем самым минимальные остовные деревья для этих компонент.

МОД для связного графа можно найти с помощью грубой силы. Поскольку множество ребер МОД является подмножеством в множестве ребер исходного графа, можно просто перебрать все возможные подмножества и найти среди них МОД. Нетрудно видеть, что это чрезвычайно трудоемкий процесс. В множестве из N ребер имеется 2^N подмножеств. Для каждого из этих подмножеств мы должны будем проверить, что оно задает связный подграф, охватывающий все вершины исходного, и не содержит циклов, а затем сосчитать его вес. Процесс можно ускорить после того, как найдено первое остовное дерево. Никакое подмножество ребер, полный вес которого больше, чем у уже найденного наилучшего остовного дерева, не может нам подойти, поэтому нет необходимости проверять связность, ацикличность и охват всех вершин. Однако даже и при этом улучшении сложность метода грубой силы будет порядка $O(2^N)$.

6.4.1. Алгоритм Дейкстры–Прима

В конце 1950-х годов Эдгар Дейкстра и Прим, работая и публикуя свои результаты независимо друг от друга, предложили следующий алгоритм построения МОД.

Воспользуемся для поиска МОД так называемым «жадным» алгоритмом. Жадные алгоритмы действуют, используя в каждый момент лишь часть исходных данных и принимая лучшее решение на основе этой части. В нашем случае мы будем на каждом шаге рассматривать множество ребер, допускающих присоединение к уже построенной части остовного дерева, и выбирать из них ребро с наименьшим весом. Повторяя эту процедуру, мы получим остовное дерево с наименьшим весом.

Разобьем вершины графа на три класса: вершины, вошедшие в уже построенную часть дерева, вершины, окаймляющие построенную часть,

и еще не рассмотренные вершины. Начнем с произвольной вершины графа и включим ее в остовное дерево. Поскольку результатом построения будет некорневое дерево, выбор исходной вершины не влияет на окончательный результат (конечно, если МОД единственно). Все вершины, соединенные с данной, заносим в кайму. Затем выполняется цикл поиска ребра с наименьшим весом, соединяющего уже построенную часть остовного дерева с каймой; это ребро вместе с новой вершиной добавляется в дерево и происходит обновление каймы. После того, как в дерево попадут все вершины, работа будет закончена.

Вот как выглядит алгоритм:

```
выбрать начальный узел
сформировать начальную кайму, состоящую из вершин,
                соседних с начальным узлом
while в графе есть вершины, не попавшие в дерево do
    выбрать ребро из дерева в кайму с наименьшим весом
    добавить конец ребра к дереву
    изменить кайму, для чего
        добавить в кайму вершины, соседние с новой
        обновить список ребер из дерева в кайму так,
                чтобы он состоял из ребер наименьшего веса
end while
```

На рис. 6.5 описано исполнение алгоритма на конкретном примере. В начале процесса мы выбираем произвольный узел A. Как мы уже говорили, при другом выборе начальной вершины результат будет тем же самым, если МОД единственно. Исходный граф изображен на рис. 6.5 (а) и, как уже упоминалось, мы начинаем построение МОД с вершины A. Все вершины, непосредственно связанные с A, образуют исходную кайму. Видно, что ребро наименьшего веса связывает узлы A и B, поэтому к уже построенной части МОД добавляется вершина B вместе с ребром AB. При добавлении к дереву вершины B (рис. 6.5 (в)) мы должны определить, не следует ли добавить к кайме новые вершины. В результате мы обнаруживаем, что это необходимо проделать с вершинами E и G. Поскольку обе эти вершины соединены только с вершиной B уже построенной части дерева, мы добавляем соединяющие ребра к списку рассматриваемых на следующем шаге. Здесь же необходимо проверить, являются ли ребра, ведущие из вершины A в C, D и F, кратчайшими среди ребер, соединяющих эти вершины с деревом, или есть более удобные ребра, исходящие из B. В исходном графе

вершина B не соединена непосредственно ни с C, ни с F, поэтому для них ничего не меняется. А вот ребро BD короче ребра AD, и поэтому должно его заменить. Наименьший вес из пяти ребер, идущих в кайму, имеет ребро BE, поэтому к дереву нужно добавить его и вершину E (рис. 6.5 (г)). Вес ребра EG меньше веса ребра BG, поэтому оно замещает последнее. Из четырех ребер, ведущих в кайму, наименьший вес имеет AC, поэтому следующим к дереву добавляется оно. Добавление к остовному дереву вершины C и ребра AC (рис. 6.5 (д)) не приводит к изменению списка ребер.

Затем мы выбираем ребро AF и добавляем его к дереву вместе с вершиной F. Кроме того, мы меняем список связей, поскольку вес ребра FD меньше веса ребра BD, а вес ребра FG меньше веса ребра EG. В получившейся кайме (рис. 6.5 (е)) ребро FD имеет наименьший вес среди оставшихся, и оно добавляется следующим.

Теперь осталось добавить к дереву всего один узел (рис. 6.5 (ж)). После этого процесс завершается, и мы построили МОД с корнем в вершине A (рис. 6.5 (з)).

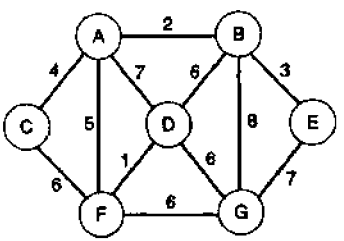


Рис. 6.5 (а). Исходный граф

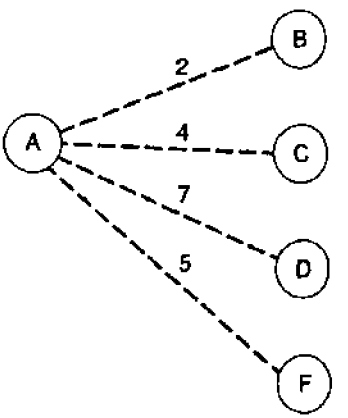


Рис. 6.5 *(б)*. Добавление первой вершины. (Пунктирные отрезки ведут к вершинам каймы.)

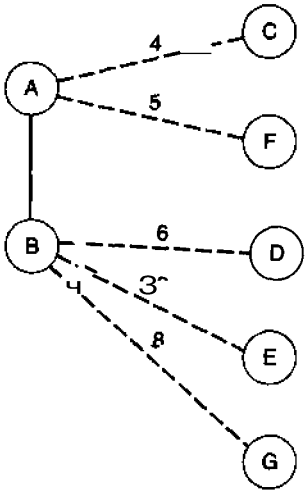


Рис. 6.5 (в). Вторая вершина добавлена; обновлены ребра, ведущие в вершины D, E и G; ребра из МОД изображены сплошными отрезками

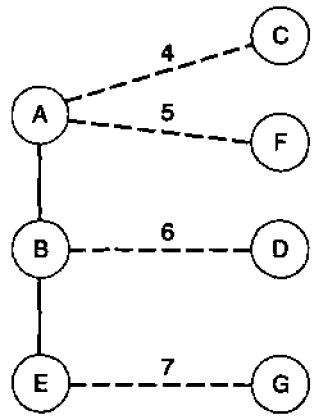


Рис. 6.5 (г). Добавлена третья вершина, изменено ребро, ведущее в вершину E

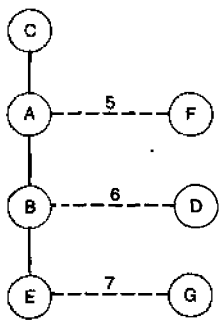


Рис. 6.5 *(д)*. К дереву добавлена вершина C

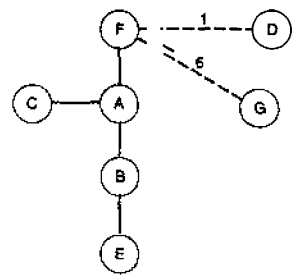


Рис. 6.5 *(е)*. К дереву добавлена вершина F, а ребра, ведущие в вершины D и G обновлены

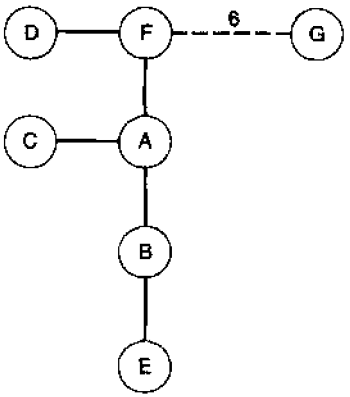


Рис. 6.5 (*ж*). В кайме осталась всего одна вершина

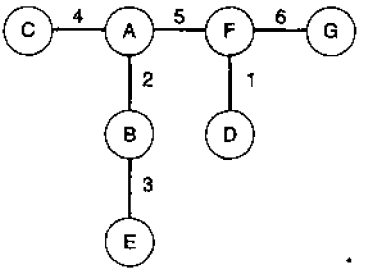


Рис. 6.5 (з). Полное минимальное остовное дерево с корнем в вершине A

6.4.2. Алгоритм Крускала

Алгоритм Дейкстры–Прима начинает построение МОД с конкретной вершины графа и постепенно расширяет построенную часть дерева; в отличие от него алгоритм Крускала делает упор на ребрах графа.

В этом алгоритме мы начинаем с пустого дерева и добавляем к нему ребра в порядке возрастания их весов пока не получим набор ребер, объединяющий все вершины графа. Если ребра закончатся до того, как все вершины будут соединены между собой, то это означает, что исходный граф был несвязным, и полученный нами результат представляет собой объединение МОД всех его компонент связности.

Мы работаем с тем же графом (см. рис. 6.6 (а)), что и при описании алгоритма Дейкстры-Прима. Начнем с ребра наименьшего веса, т.е. с ребра DF. Начальная ситуация изображена на рис. 6.6 (б).

Следующим добавляется ребро веса 2, соединяющее вершины A и B (рис. 6.6 (в)), а затем - - ребро веса три, и мы получаем ситуацию с рис. 6.6 (г).

К результирующему графу последовательно добавляются ребра весов 4 и 5 (рисунки 6.6 (д) и (е)). Неприсоединенной осталась лишь вершина G. Следующими мы должны рассмотреть ребра веса 6.

Два из четырех ребер веса 6 следует отбросить, поскольку их добавление приведет к появлению цикла в уже построенной части МОД. Присоединение ребра CF создало бы цикл, содержащий вершину A, а присоединение ребра BD - - цикл, содержащий вершины A и F. Обе

оставшиеся возможности подходят в равной степени, и, выбирая одну из них, мы получаем либо МОД с рис. 6.6 (ж), либо МОД с рис. 6.6 (з).

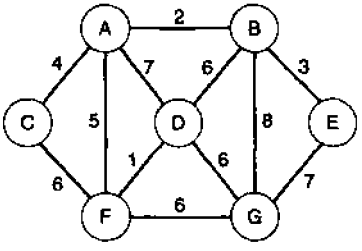


Рис. 6.6 (а). Исходный граф

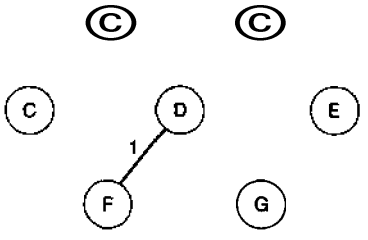


Рис. 6.6 (*б*). Первое ребро добавлено

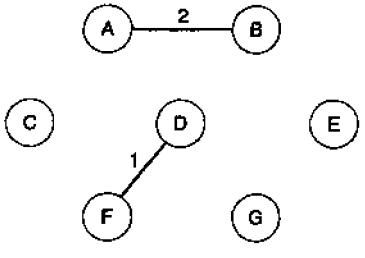


Рис. 6.6 *(в)*. Добавлено второе ребро

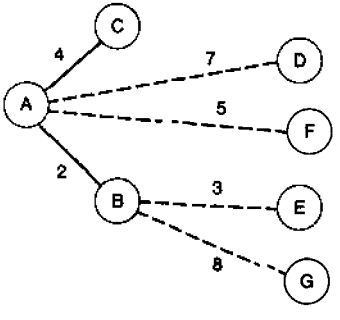


Рис. 6.6 (г). Добавлено третье ребро

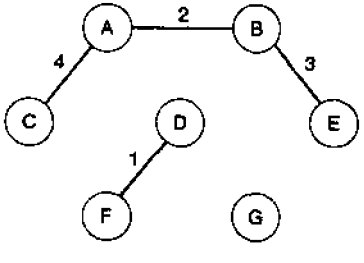


Рис. 6.6 *(д)*. Добавлено четвертое ребро

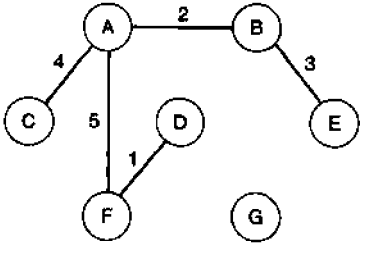


Рис. 6.6 *(е)*. Добавлено пятое ребро

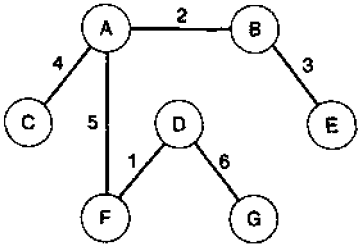


Рис. 6.6 *(ж)*. Минимальное остовное дерево

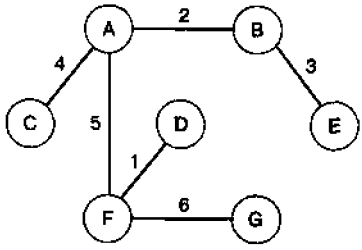


Рис. 6.6 *(з)*. Другое минимальное остовное дерево

Вот общий алгоритм, реализующий эту процедуру (число ребер в графе обозначено через E, а число вершин — через N):

```
отсортировать ребра в порядке возрастания весов
инициализировать структуру разбиений
edgeCount=1
while edgeCount<=E and includedCount<=N-1 do
   parent1=FindRoot(edge[edgeCount].start)
   parent2=FindRoot(edge[edgeCount].end)
   if parent1/=parent2 then
      добавить edge[edgeCount] в остовное дерево
      includedCount=includedCount=1
      Union(parent1,parent2)
   end if
   edgeCount=edgeCount+1
end while
```

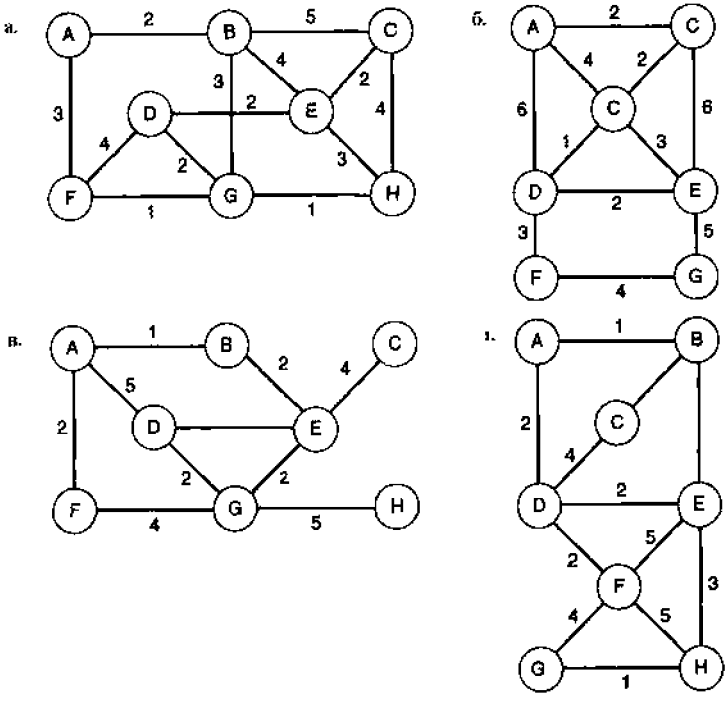
Главный цикл будет выполняться до тех пор, пока значение переменной edgeCount не совпадет с числом ребер в графе или пока значение переменной `includedCount` не покажет нам, что мы добавили достаточно ребер, чтобы образовать остовное дерево. Заметьте, что остовное дерево в графе с N вершинами должно иметь $N - 1$ ребро.

Внутри цикла мы в первую очередь находим родителей вершин, соединяемых очередным добавляемым ребром. Если эти вершины принадлежат разбиениям с различными корнями, то добавление ребра между ними не приведет к появлению цикла, а два разбиения сольются в одно с общим корнем. Подпрограммы FindRoot и Union будут подробно описаны в § 6.7.

Сложность этого алгоритма совпадает со сложностью используемого алгоритма сортировки, поскольку число операций в цикле while линейно по числу ребер. Поэтому сложность алгоритма Крускала поиска МОД равна $O(E \log E)$.

6.4.3. Упражнения

1) Найдите минимальное остовное дерево с помощью алгоритма Дейкстры–Прима в каждом из следующих графов, начиная с вершины A. Выпишите все шаги.



2) Найдите минимальное остовное дерево в графах из упражнения 1) с помощью алгоритма Крускала.

3) Выполните анализ алгоритма Дейкстры–Прима, подсчитав, сколько раз рассматривается каждое ребро при добавлении вершин к кайме, при обновлении списка ребер, ведущих в кайму, и при перенесении вершины из каймы в МОД.

4) Докажите, что ребро, вес которого строго меньше весов всех остальных ребер, наверняка входит в любое МОД.

5) Докажите, что если в связном графе веса всех ребер различны (или, другими словами, в нем нет ребер с одинаковыми весами), то в нем есть лишь одно минимальное остовное дерево.

6) Может ли так случиться, что порядок добавления ребер к МОД при исполнении алгоритма Дейкстры-Прима совпадает с порядком их появления при обходе дерева по уровням? Или так происходит всегда? Или такое вообще невозможно? Обоснуйте свой ответ.

7) Может ли так случиться, что порядок добавления ребер к МОД при исполнении алгоритма Дейкстры–Прима совпадает с порядком их появления при обходе дерева в глубину. Или так происходит всегда? Или такое вообще невозможно? Обоснуйте свой ответ.

6.5. Алгоритм поиска кратчайшего пути

Результатом алгоритма поиска кратчайшего пути является последовательность ребер, соединяющая заданные две вершины и имеющая наименьшую длину среди всех таких последовательностей.

На первый взгляд кажется, что мы можем воспользоваться алгоритмом построения МОД, чтобы отбросить лишние ребра, а затем взять путь, соединяющий заданные вершины в построенном остовном дереве. К сожалению, такие действия не всегда приводят к нужному результату.

Напомним, что алгоритм построения МОД нацелен на поиск дерева с минимальным суммарным весом ребер. Рассмотрим, например, «циклический» граф, т.е. такой граф, в котором первая вершина соединена со второй, вторая — с третьей, и так далее, а последняя вершина в свою очередь соединена с первой. Такой граф представляет собой просто кольцо, каждая вершина в котором соединена с двумя другими. Пример такого графа с шестью вершинами приведен на рис. 6.7 (а).

Обратите внимание на то, что вес каждого ребра равен 1 за исключением ребра, соединяющего вершины A и B, вес которого равен 2. Алгоритм построения минимального остовного дерева выберет все ребра веса 1, отбросив единственное ребро веса 2. Это означает, однако, что путь от A к B в минимальном остовном дереве должен проходить

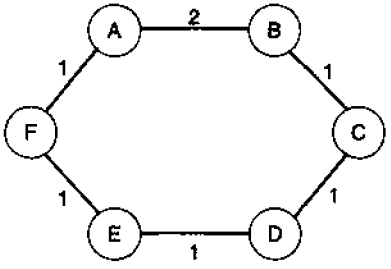


Рис. 6.7 (*а*). Кольцевой граф

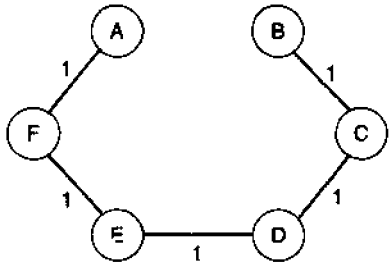


Рис. 6.7 *(б)*. Его минимальное остовное дерево

через все остальные вершины, а его длина равна 5 (рис. 6.7 (б)). Ясно, что он не является кратчайшим, поскольку в исходном графе вершины A и B соединены ребром длины 2.

6.5.1. Алгоритм Дейкстры

Жадный алгоритм построения минимального остовного дерева непригоден для поиска кратчайшего пути между двумя вершинами, поскольку на каждом проходе он учитывает длину лишь одного ребра. Если же изменить его так, чтобы при выборе ребра, ведущего в кайму, он выбирал ребро, являющееся частью кратчайшего в целом пути из начальной вершины, то мы получим требуемый результат. Точнее говоря, вот измененный алгоритм:

```
выбрать начальную вершину
создать начальную кайму из вершин, соединенных с начальной
while вершина назначения не достигнута do
    выбрать вершину каймы с кратчайшим расстоянием до начальной
    добавить эту вершину и ведущее в нее ребро к дереву
    изменить кайму путем добавления к ней вершин,
        соединенных с вновь добавленной
        for всякой вершины каймы do
            приписать к ней ребро, соединяющее ее с деревом и
                    завершающее кратчайший путь к начальной вершине
        end for
end while
```

На рис. 6.8 приведен пример исполнения этого алгоритма. Алгоритм выполняется на том же графе (рис. 6.8 (а)), что и алгоритм построения минимального остовного дерева, а искать мы будем кратчайший путь от A до G.

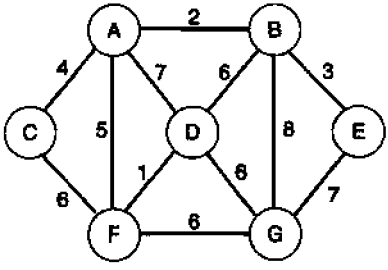


Рис. 6.8 (а). Исходный граф

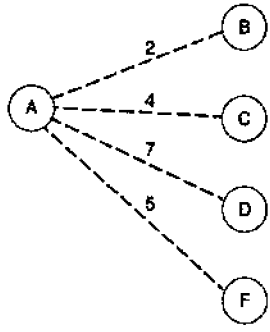


Рис. 6.8 *(б)*. Кратчайший путь в вершину B

В начале пути из вершины A у нас есть четыре возможных ребра. Из этих четырех ребер ребро AB является кратчайшим. Поэтому мы добавляем к дереву вершину B (рис. 6.8 (в)) и смотрим, как следует обновить набор путей. С уже построенным деревом соединены теперь вершины E и G, поэтому их следует добавить к кайме. Кроме того, мы должны посмотреть на вершину D и сравнить прямой путь из нее в A, длина которого равна 7, с окольным путем через вершину B, длина которого равна 8. Прямой путь короче, поэтому приписанное к D ребро менять не следует. Изучив теперь имеющиеся возможности, мы видим, что кратчайшим является путь из A в C длины 4. Ребро BE короче, однако мы рассматриваем полную длину пути из A, а такой путь, ведущий в E, имеет длину 5. Теперь к дереву кратчайших путей добавим вершину C (рис. 6.8 (г)). Посмотрев на граф, мы обнаруживаем, что можем пройти в вершину F через вершину C, однако длина этого пути будет равна 10 — больше, чем у прямого пути из A в F, поэтому изменения в наборе путей не производятся.

На рис. 6.8 (г) мы можем выбрать теперь либо путь из A в F, либо путь из A в E, проходящий через вершину B: оба они имеют длину 5. Какой из путей будет выбран при исполнении программы, зависит от способа хранения данных в ней. Мы же, встретившись с необходимостью добавить вершину, будем всегда выбирать вершину, метка которой первая в лексикографическом порядке.

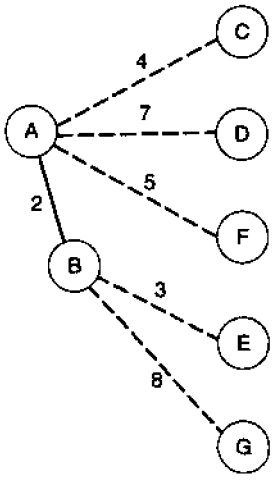


Рис. 6.8(*в*). Путь длины 4 в вершину C — кратчайший из вариантов

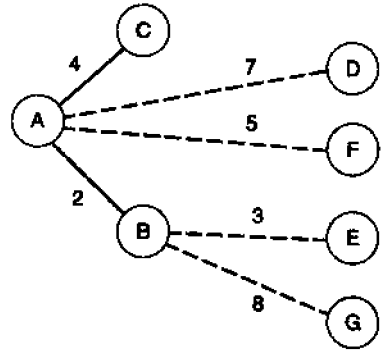


Рис. 6.8 (г). Кратчайшим является любой из путей длины 5 в вершину E **или** F

В результате получим ситуацию с рис. 6.8 (д). Добавление к дереву вершины E не меняет остальных связей, поэтому теперь мы можем добавить вершину F и получим картинку с рис. 6.8 (е). Обратите внимание на то, что, хотя добавление вершины F и привело к изменению ребра, ведущего из D, если бы мы начали с F, все равно на очередном шаге мы должны были бы добавить E.

На рис. 6.8 (е) видно, что путь в вершину D короче пути в вершину G. Поэтому мы добавляем к дереву вершину D и получаем ситуацию, изображенную на рис. 6.8 (ж). Осталось добавить только вершину G, и в результате мы получаем дерево кратчайшего пути с рис. 6.8 (з). Длина кратчайшего пути из A в G равна 10. Вернувшись к рис. 6.5 (з), мы находим на нем еще один пример минимального остовного дерева, в котором путь между A и G не является кратчайшим: его длина равна 11.

На примере с рис. 6.8 мы имеем дело с полным деревом кратчайших путей, поскольку целевая вершина G была добавлена к дереву последней. Если бы мы добрались до нее раньше, алгоритм бы тотчас завершил свою работу. Могут возникнуть ситуации, в которых нас интересуют кратчайшие пути из данной вершины во все остальные. Если, например, мы имеем дело с небольшой компьютерной сетью, скорости передачи данных между узлами которой приблизительно постоянны, то

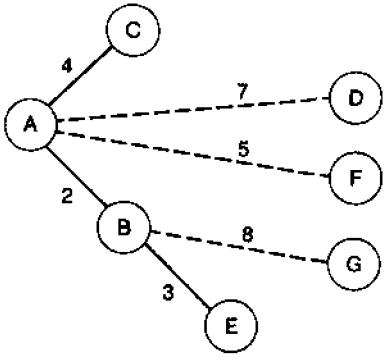


Рис. 6.8 (*д*). Следующим является другой путь длины 5 в вершину F

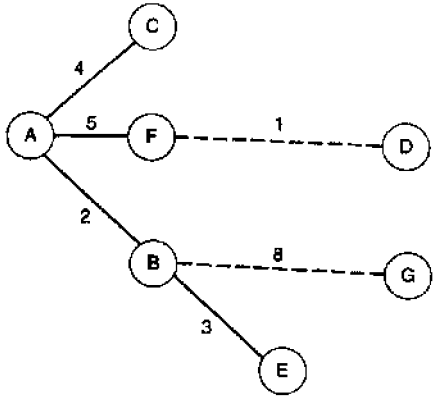


Рис. 6.8 *(е)*. Путь длины 6 в вершину D короче пути в вершину G

для каждого из компьютеров сети мы можем выбрать кратчайший путь до каждого из остальных компьютеров. Поэтому при необходимости переслать сообщение единственное, что нам потребуется, это воспользоваться заранее рассчитанным наиболее эффективным путем.

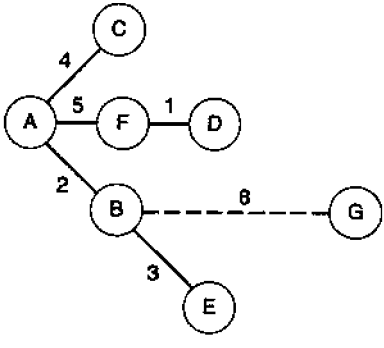


Рис. 6.8 *(ж)*. Остался лишь путь в вершину G

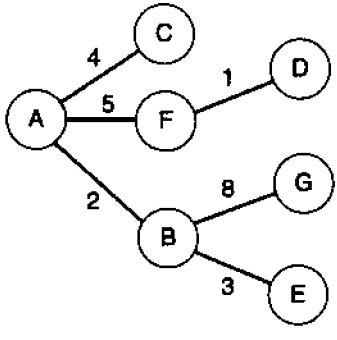
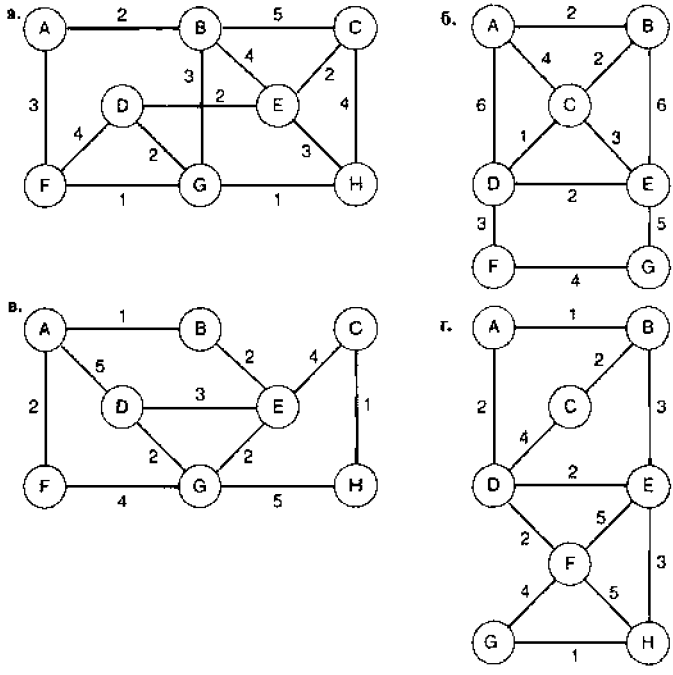


Рис. 6.8 (*з*). Полное дерево кратчайшего пути из вершины A

6.5.2. Упражнения

1) Выполните алгоритм поиска кратчайшего пути из вершины A во все остальные вершины в каждом из следующих примеров. Подсчитайте количество просмотренных Вами ребер (если Вы обра-

щаетесь к ребру несколько раз, то всякий раз его нужно учитывать заново).



2) Измените приведенный в последнем разделе алгоритм таким образом, чтобы он эффективно определял кратчайший путь из данной вершины графа во все остальные его вершины, а не только в одну указанную.

3) Выполните анализ алгоритма поиска кратчайшего пути, подсчитав, сколько раз рассматривается каждое ребро при добавлении вершин к кайме, при обновлении списка ребер, ведущих в кайму, и при перенесении вершины из каймы в МОД.

4) Докажите, что обход по уровням строит кратчайшие пути в графе без весов.

5) Может ли так случиться, что порядок добавления ребер к дереву кратчайшего пути совпадает с порядком их появления при обходе дерева по уровням? Или так происходит всегда? Или такое вообще невозможно? Обоснуйте свой ответ.

6) Может ли так случиться, что порядок добавления ребер к дереву кратчайшего пути совпадает с порядком их появления при обходе дерева в глубину? Или так происходит всегда? Или такое вообще невозможно? Обоснуйте свой ответ.

6.6. Алгоритм определения компонент двусвязности

Компонентой двусвязности графа называется такое максимальное подмножество из трех или более его вершин, в котором любые две вершины соединены по крайней мере двумя путями, не имеющими общих ребер. Кроме того компонента двусвязности может представлять собой просто две вершины, соединенные одним ребром. Компонента двусвязности — устойчивая часть графа: если в ней удалить вершину и все примыкающие к ней ребра, то любые две из оставшихся вершин попрежнему оказываются соединенными между собой. На рис. 6.9 приведен пример графа с тремя компонентами двусвязности. Вершины первой компоненты имеют метки A, B, C, D; вершины второй — метки D, E, F, G; третья компонента содержит вершины H и I.

Анализ компонент двусвязности компьютерной сети показывает, насколько она устойчива при разрушениях отдельных узлов. Таким образом, если граф, образованный компьютерами сети, двусвязен, то сеть сохранит способность функционировать даже при выходе из строя одного из компьютеров. При планировании авиаперевозок двусвязность компоненты графа гарантирует возможность переправить пассажиров по другому маршруту при закрытии аэропорта по погодным условиям.

Точками сочленения в связном графе служат такие вершины, удаление которых превращает граф в несвязный. Точки сочленения — это такие вершины, которые принадлежат сразу двум компонентам двусвязности. На рис. 6.9 это вершины D и H. Определение точек сочленения и компонент двусвязности тесно связаны между собой.

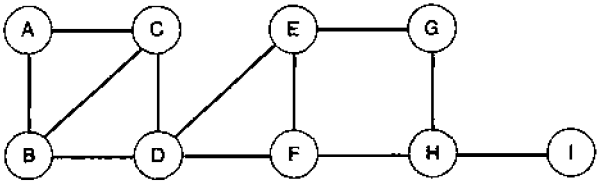


Рис. 6.9. Связный граф с тремя компонентами двусвязности

Точки сочленения можно обнаружить с помощью грубой силы, удаляя из графа поочередно каждую вершину и пользуясь одним из наших алгоритмов обхода графа для проверки связности оставшейся части. Если оставшийся граф связен, то удаленная вершина не является точкой сочленения. При таком подходе мы должны выполнить N обходов графа с N вершинами, что потребует $O(N^2)$ операций. Однако сохраняя незначительное количество дополнительной информации при обходе графов, можно обнаружить все точки сочленения и компоненты двусвязности при единственном обходе.

Рассмотрим пути в графе с рис. 6.9, начинающиеся в вершине F. Видно, что вне зависимости от порядка прохождения узлов всякий путь из F в любую из вершин A, B или C должен проходить через вершину D. Это означает, что вершина D является точкой сочленения, а подграф, включающий вершины A, B, C и D — компонентой двусвязности.

Мы будем строить алгоритм на обходе графа в глубину. Из раздела 6.3.1 Вы знаете, что при обходе в глубину мы перебираем ребра, пока не встретится тупик. При попадании в тупик происходит возвращение вдоль пройденного пути, однако теперь мы потребуем, чтобы алгоритм сообщал информацию о том, насколько высоко мы окажемся в дереве обхода, если пойдем за тупик. Эти возвратные ребра в дереве обозначают циклы в графе. Все вершины цикла должны принадлежать одной компоненте двусвязности. Расположение возвратных ребер указывает, насколько далеко мы можем вернуться по дереву прежде, чем встретим точку сочленения.

При реализации алгоритма будем подсчитывать число посещенных узлов графа. Каждому узлу сопоставим индекс — номер, указывающий момент посещения узла. Другими словами, первый посещенный узел будет иметь номер 1, второй — номер 2 и т.д. После попадания в тупик мы просмотрим все соседние с ним вершины (за исключением той, из которой только что пришли) и будем считать наименьший из номеров этих вершин индексом возврата из тупика. Если тупик соединен всего с одной вершиной (той самой, из которой мы только что пришли), то индексом возврата будем считать номер тупика. При возвращении в узел, который не является корнем дерева обхода, мы сравним его номер с возвращенным индексом возврата. Если индекс возврата не меньше номера текущего узла, то только что пройденное поддерево (за вычетом всех ранее найденных компонент двусвязности) является компонентой двусвязности. Всякая внутренняя вершина дерева обхода в глубину

возвращает наименьшее значение среди всех индексов примыкающих вершин и всех возвращаемых ей индексов возврата.

Как будет действовать эта процедура на нашем графе с рис. 6.9? Начав с вершины F, мы присвоим ей номер 1. Затем мы перейдем к вершинам D (номер 2), B (номер 3), A (номер 4) и C (номер 5). Вершина C является тупиком, и из нее идут ребра назад в вершины A, B и D. Номер вершины D является наименьшим, поэтому индекс возврата 2 будет возвращен в вершину A. Поскольку индекс возврата меньше номера вершины A, эта вершина не является точкой сочленения. Пока индекс 2 является наименьшим, и он будет возвращен также в вершину B. Процесс возвращения продолжается, пока мы не вернемся в вершину D и не обнаружим, что номер этой вершины совпадает с индексом возврата, а значит вершины A, B, C и D образуют компоненту двусвязности. Затем мы возвращаемся в корневую вершину F дерева обхода и переходим из нее в вершину E (с номером 6), за которой последуют вершины G (номер 7) и H (номер 8). Затем мы переходим к вершине I (номер 9), и, поскольку она тупиковая и к ней не примыкает никаких вершин, отличных от H, ее номер возвращается в качестве индекса возврата. Когда вершина H получает от вершины I индекс возврата 9, мы обнаруживаем еще одну компоненту двусвязности, содержащую вершины H и I. Теперь в вершине H сравниваются три значения: индекс 1 (задаваемый ребром возврата к вершине F), индекс 9 (возвращенный из вершины I) и индекс 8 (номер самой вершины H). Наименьший из этих индексов возвращается сначала в G, а затем в E. Затем вершина E возвращает полученное значение индекса возврата назад в корневую вершину, и, поскольку мы посетили уже все вершины графа, оставшиеся узлы D, E, F, G и H образуют еще одну, последнюю, компоненту двусвязности. На рис. 6.10 изображен результат выполнения этой процедуры, откуда видно, что точками сочленения в исходном графе служат вершины D и H — единственные вершины, входящие в различные компоненты двусвязности.

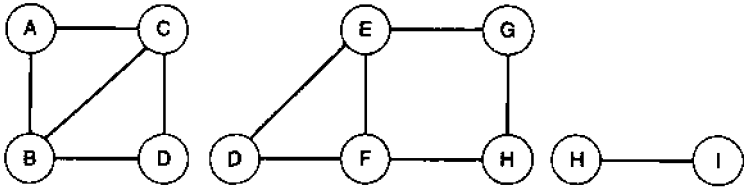
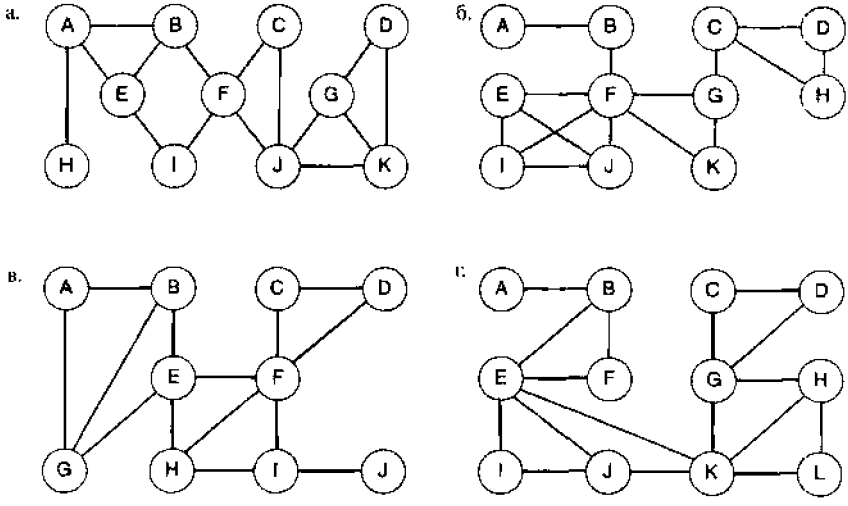


Рис. 6.10. Компоненты двусвязности графа с рис. 6.9

6.6.1. Упражнения

1) Найдите компоненты двусвязности в следующих графах:



2) Напишите алгоритм, определяющий компоненты двусвязности в графе, представленном матрицей смежности.

3) Напишите алгоритм, определяющий компоненты двусвязности в графе, представленном списком смежности.

6.7. Разбиения множеств

Многие алгоритмы используют представление множеств в виде набора непересекающихся подмножеств, которые можно по-разному комбинировать. Для этой цели можно воспользоваться средствами работы с множествами, имеющимися в современных языках программирования, однако их реализации не гарантируют, что различные подмножества действительно будут непересекающимися. Кроме того, средства обработки множеств имеются не во всех языках программирования. В этом параграфе мы рассмотрим способ реализации разбиений множеств на массивах. Подобные алгоритмы могут принести пользу, как мы видели, при реализации алгоритма Крускала построения минимального остовного поддерева.

Начнем с массива Parent, в котором под каждый элемент множества, с которым мы собираемся работать, отведена одна ячейка. Внача-

ле мы полагаем значения всех элементов равными −1. Знак — означает здесь, что каждый элемент является корнем разбиения, а абсолютное значение 1 -- что соответствующая часть разбиения состоит из одного элемента. При работе алгоритма значения ячеек будут меняться, и, скажем, значение −7 указывает на то, что соответствующий элемент является корнем части, состоящей из семи элементов. При добавлении элемента к части значение соответствующей ему ячейки добавляется к значению корня. Если, к примеру, пятый элемент добавляется к части, корень которой — восьмой элемент массива Parent, то в пятую ячейку записывается значение 8, а отрицательное содержимое восьмой ячейки будет увеличено по абсолютной величине, чтобы указать на появление дополнительного элемента. При объединении двух частей разбиения происходит изменение только корневых ячеек этих частей, поэтому каждый элемент массива ссылается на своего непосредственного предка.

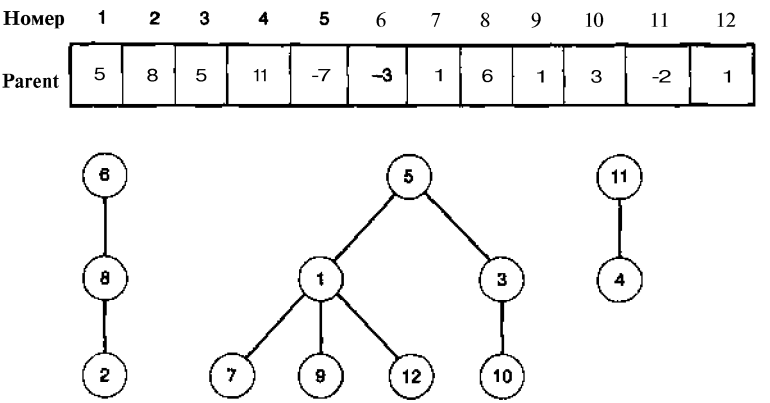


Рис. 6.11. Разбиение множества чисел от 1 до 12 и соответствующий массив Parent

Рассмотрим разбиение, изображенное на рис. 6.11. Видно, что оно состоит из трех частей с корнями 5, 6 и 11. В этих ячейках записаны отрицательные значения, модули которых равны числу элементов в соответствующем разбиении. Во всех остальных ячейках хранятся номера их непосредственных предков. Если мы захотим объединить части с корнями 6 и 11, то нужно в ячейку Parent [11] занести значение 6 — новый корень объединенной части, а в ячейку 6 — значение −5, модуль которого равен числу элементов в объединении двух частей. Вот алгоритмы, реализующие эту структуру разбиений:

```
InitializePartition(N)
N   число элементов в множестве
for i=1 to N do
   Parent[i]=-1
end for
```

Как мы уже говорили, процедура инициализации просто заносит в каждую ячейку массива Parent значение -1, которое указывает, что каждый элемент составляет отдельную часть разбиения.

```
Union(i,j)
i,j   объединяемые части разбиения

totalElements=Parent[i]+Parent[j]
if Parent[i]>=Parent[i]+Parent[j] then
   Parent[i]=j
   Parent[j]=totalElements
else
   Parent[j]=i
   parent[i]=totalElements
end if
```

Функция Union выполняет объединение двух частей в одну. Сначала она подсчитывает общее число элементов в объединенной части. Затем она подсоединяет меньшую часть к большей. Напомним, что размеры частей указываются отрицательными числами, поэтому вариант then добавляет меньшую i-ую часть к большей j-ой части, а вариант else добавляет меньшую j-ую часть к большей i-ой части.

```
FindRoot(s)
s   элемент, для которого мы хотим найти корень части,
    его содержащей

result=s
while Parent[result]>0 do
   result=Parent[result]
end while
return result
```

Процедура **FindRoot** начинает с ячейки Parent [s], в которой хранится номер непосредственного предка элемента s. Если это значение

отрицательно, то s сам является корнем разбиения, и он и возвращается в качестве результата. Если же s не является корнем, то мы заносим в переменную result значение непосредственного предка элемента s и проверяем, не является ли оно корнем. Эта последовательность действий повторяется, пока мы не обнаружим корень. Эффективность этого алгоритма можно повысить, если, вернувшись по пройденному пути, занести во все пройденные ячейки номер найденного корня. На такую работу уходит некоторое время, однако при последующих обращениях корень части, содержащей просмотренные элементы, будет найден быстрее.

6.8. Упражнения по программированию

Для решения формулируемых здесь задач Вам понадобится генерировать полные взвешенные графы на N вершинах. Для этой цели лучше всего использовать матрицу примыканий: Вам придется лишь заполнить все клетки в ней за исключением диагональных, значения в которых должны равняться нулю. Вы будете работать лишь с неориентированными графами, поэтому позаботьтесь о том, чтобы элементы `AdjMat[i,j]` и `AdjMat[j,i]` совпадали при всех `i,j`.

В этих задачах от Вас требуется сравнить два остовных дерева или два пути. Такое сравнение провести легче, если обозначать ребра парой вершин в порядке возрастания их меток. Это возможно, поскольку работать придется с неориентированными графами. Затем ребра можно отсортировать по возрастанию меток ребер. Отсортированные списки ребер двух одинаковых минимальных остовных деревьев или путей должны совпадать в точности.

1) Сгенерируйте полный взвешенный граф с 50 вершинами. Выполните алгоритм Дейкстры-Прима поиска минимального остовного дерева в этом графе, начиная поочередно с каждой из его вершин, и подсчитайте количество найденных таким образом минимальных остовных деревьев. Повторите эту процедуру четырежды, устанавливая максимальное значение случайного веса ребра равным 10, 25, 50 и 100. Сведите результаты экспериментов в отчет и дайте им объяснение. Если для каждого из значений максимума Вы сгенерируете несколько тестовых графов, то результаты будут более обоснованы.

2) Сгенерируйте полный взвешенный граф с 50 вершинами. Выполните алгоритм **Крускала** поиска минимального остовного дерева в этом графе. Встретив два ребра с одинаковым весом, выбирайте добавляемое ребро случайным образом. Сгенерируйте для графа 10 минимальных деревьев и найдите среди них попарно различные. Поскольку выбор весов производится случайным образом, при наличии различных минимальных остовных деревьев они должны обнаружиться. Повторите **эту** процедуру четырежды, устанавливая максимальное значение случайного веса ребра равным 10, 25, 50 и 100. Сведите результаты экспериментов в отчет и дайте им объяснение. **Если** для каждого из значений максимума Вы сгенерируете несколько тестовых графов, то результаты будут более обоснованы.

3) Сгенерируйте полный взвешенный граф с 50 вершинами. Для каждой пары вершин (A и B) проверьте, совпадает ли генерируемый кратчайший путь из A в B с генерируемым кратчайшим путем из B в A и подсчитайте число различных пар. Повторите эту процедуру четырежды, устанавливая максимальное значение случайного веса ребра равным 10, 25, 50 и 100. Сведите результаты экспериментов в отчет и дайте им объяснение. Если для каждого из значений максимума Вы сгенерируете несколько тестовых графов, то результаты будут более обоснованы.

4) Напишите программу, которая будет генерировать полный взвешенный граф. Выполните алгоритм Дейкстры-Прима и Крускала поиска минимального остовного дерева в этом графе. Установите счетчики, подсчитывающие число посещений каждым алгоритмом каждого ребра. Другими словами, подсчитайте число обращений к матрице примыканий. Повторите эту процедуру четырежды, устанавливая максимальное значение случайного веса ребра равным 10, 25, 50 и 100. Сведите результаты экспериментов в отчет и дайте им объяснение. Если для каждого из значений максимума Вы сгенерируете несколько случайных тестовых графов различных размеров, то результаты будут более обоснованы.

Глава 7.

Параллельные алгоритмы

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- читать и создавать алгоритмы;
- анализировать алгоритмы, подобные описанным в главах 2-6.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- объяснять модели PRAM;
- распознавать простые случаи, допускающие использование параллелизма;
- разрабатывать простые параллельные алгоритмы.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Возможно Вы сочтете полезным графическое изображение передачи данных при параллельной обработке. Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

Люди давно уже поняли, что в большинстве случаев два человека способны выполнить работу быстрее, чем каждый из них по отдельности, а трое могут справиться с ней еще быстрее. На практике такое распараллеливание выполняется по-разному. Документы из большого числа папок в офисе легче перебрать, если распределить папки по нескольким сотрудникам. Конвейер ускоряет процесс сборки, поскольку человек, постоянно выполняющий одну и ту же работу, делает ее быстрее: нет необходимости менять инструменты. Ведра с водой быстрее передавать по цепочке, чем бегать с ними от одного места к другому.

В параллельном программировании используются очень похожие средства. Существуют многозадачные системы, в которых каждый процессор выполняет одни и те же операции над различными данными. Имеются конвейерные системы, в которых каждый процессор осуществляет лишь один шаг задания, передавая результаты следующему процессору, который делает очередной шаг. Системы с потоками данных представляют собой последовательность процессоров, сформированную для выполнения определенного задания; для вычисления результата данные передаются от одного процессора к другому.

Эта глава служит введением в параллельное программирование. Параллельные алгоритмы — сложный объект, поэтому подробный разговор о них по меньшей мере удвоил бы объем книги. Мы начнем с обзора общих понятий, связанных со структурой параллельных компьютерных систем, а затем займемся параллельными алгоритмами решения некоторых из задач, которые мы рассматривали в главах 2-6. Предлагаемые нами параллельные алгоритмы не всегда дают наилучшее возможное решение. Их цель — дать представление о методах параллельного решения задач. Разговор о наиболее эффективных методах параллельного программирования выходит далеко за рамки этой книги.

7.1. Введение в параллелизм

В этом параграфе мы вводим основные понятия, используемые в параллельном программировании. Начнем с попытки осмыслить само понятие компьютерной обработки данных. Затем обратимся к архитектурам компьютерных систем. В завершение параграфа мы обсудим принципы, которыми будем руководствоваться при анализе параллельных алгоритмов.

7.1.1. Категории компьютерных систем

Компьютерные системы можно разделить на четыре основных категории. Для этого нужно несколько сменить наше представление о том, как работает программа. С точки зрения центрального процессора программа представляет собой поток инструкций, которые следует расшифровать и выполнить. Данные тоже можно считать поступающими в виде потока. Четыре категории, о которых мы говорим, определяются тем, поступают ли программы и данные одним потоком или несколькими.

Один поток инструкций / Один поток данных

Модель Один поток инструкций / Один поток данных (SISD) представляет собой классическую модель с одним процессором. К ней относятся как компьютеры предыдущих поколений, так и многие современные компьютеры. Процессор такого компьютера способен выполнять в любой момент времени лишь одну инструкцию и работать лишь с одним набором данных. В таких последовательных системах никакого параллелизма нет в отличие от прочих категорий.

Один поток инструкций / Несколько потоков данных

В компьютерах с одним потоком инструкций и несколькими потоками данных (SIMD) имеется несколько процессоров, выполняющих одну и ту же операцию, но над различными данными. SIMD-машины иногда называются также векторными процессорами, поскольку они хорошо приспособлены для операций над векторами, в которых каждому процессору передается одна координата вектора и после выполнения операции весь вектор оказывается обработанным. Например, сложение векторов -- покоординатная операция. Первая ко-

ордината суммы векторов -- сумма первых координат суммируемых векторов, вторая — сумма вторых координат, и т.д. В нашей SIMD-машине каждый процессор получит инструкцию сложить пару координат входных векторов. После выполнения этой единственной инструкции результат будет подсчитан полностью. Заметим, что на векторе из N элементов SISD-машине потребуется выполнить N итераций цикла, а SIMD-машине с не менее, чем N процессорами, хватит одной операции.

Несколько потоков инструкций / Один поток данных

Возможность одновременно выполнять различные операции над одними и теми же данными может поначалу показаться странной: не так уж часто встречаются программы, в которых нужно какое-то значение возвести в квадрат, умножить на 2, вычесть из него 10 и т.д. Однако, если посмотреть на эту ситуацию с другой точки зрения, то мы увидим, что на машинах такого типа можно усовершенствовать проверку числа на простоту[1]. Если число процессоров равно N, то мы можем проверить простоту любого числа между 1 и N^2 на MISD-машине за одну операцию: если число X составное, то у него должен быть делитель, не превосходящий $\sqrt{X}$. Для проверки простоты числа $X < N^2$ поручим первому процессору делить на 2, второму — делить на 3, третьему — на 4, и так до процессора с номером $K - 1$, который будет делить на K, где $K = \lceil\sqrt{X}\rceil$. Если на одном из этих процессоров деление нацело проходит успешно, то число X составное. Поэтому мы достигаем результата за одну операцию. Как нетрудно видеть, на последовательной машине выполнение этого алгоритма потребовало бы по меньшей мере $\lceil\sqrt{X}\rceil$ проходов, на каждом из которых производится одно деление.

Несколько потоков инструкций / Несколько потоков данных

Это наиболее гибкая из категорий. В случае MIMD-систем мы имеем дело с несколькими процессорами, каждый из которых способен выполнять свою инструкцию. Кроме того имеется несколько потоков данных, и каждый процессор может работать со своим набором данных. На практике это означает, что MIMD-система может выполнять на каждом процессоре свою программу или отдельные части одной

[1] Напомним, что простое число — это такое натуральное число, большее 1, которое делится нацело только на само себя и на 1. Например, число 17 простое, поскольку у него нет делителей среди чисел от 2 до 16.

и той же программы, или векторные операции так же, как и SIMD-конфигурация. Большинство современных подходов к параллелизму — включая кластеры компьютеров и мультипроцессорные системы — лежат в категории MIMD.

7.1.2. Параллельные архитектуры

В архитектуре параллельных компьютерных систем основную роль играют два аспекта: как связаны между собой процессоры и их память и как процессоры взаимодействуют. Об этих аспектах мы и будем говорить при обсуждении параллельных алгоритмов, поскольку те или иные решения могут иметь разную эффективность для различных задач.

Слабо и сильно связанные машины

В машинах со слабой связью у каждого процессора есть своя собственная память, а связь между процессорами осуществляется по «сетевым» кабелям. Это архитектура компьютерных кластеров: каждый компьютер кластера представляет собой отдельную компьютерную систему и может работать самостоятельно. Параллелизм осуществляется за счет распределения заданий по компьютерам кластера центральным управляющим компьютером.

В машинах с тесными связями все процессоры пользуются общей центральной памятью. Взаимодействие процессоров осуществляется за счет того, что один из них записывает информацию в общую память, а остальные считывают ее оттуда. Пример такого взаимодействия будет приведен в § 7.3.

Взаимодействие процессоров

Мы говорили о том, что в машинах со слабыми связями взаимодействие процессоров осуществляется по кабелям или проводам. Посмотрим на возможную организацию таких связей. На одном конце спектра расположена полносвязная сеть, каждый процессор в которой соединен со всеми остальными. На другом конце — линейная сеть, в которой процессоры выстроены в цепочку и каждый процессор, за исключением двух концевых, соединен с двумя соседними (у концевых процессоров по одному соседу). Информация в полносвязной сети передается от процессора к процессору очень быстро, однако такая организация требует большой протяженности кабеля. В линейной сети

информация передается медленнее за счет необходимости пропускать ее через промежуточные узлы, и разрыв цепочки в одном месте обрывает поток данных. Мы уже обращались к этой проблеме при обсуждении компонент двусвязности в главе 6.

В качестве альтернативы линейной сети, повышающей устойчивость, можно предложить кольцевую структуру; она устроена так же, как и линейная сеть, но первое и последнее звено цепочки тоже соединены между собой. В такой сети информация распространяется быстрее, поскольку она должна проходить уже не более половины процессоров.

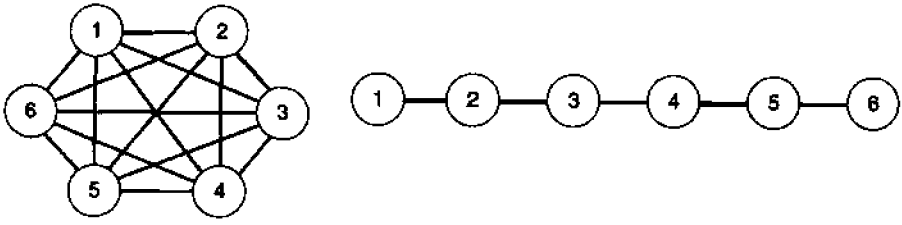


Рис. 7.1. Полносвязная и линейная конфигурации сети

Скажем, в линейной сети на рис. 7.1 сообщение из узла 1 в узел 5 должно пройти через три промежуточных узла, а в кольцевой сети с рис. 7.2 достаточно пропустить его через единственный промежуточный узел 6. В сети с рис. 7.3 процессоры расположены в вершинах двумерной решетки, а кабели соединяют соседей по горизонтали или по вертикали. Такая сеть еще более эффективна и устойчива, чем кольцевая, однако расход кабеля в ней возрастает.

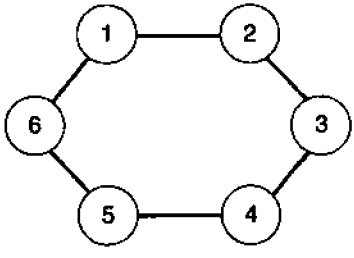


Рис. 7.2. Кольцевая конфигурация сети

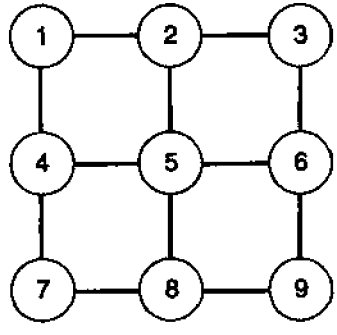


Рис. 7.3. Решетчатая сеть

Есть и другие возможности, которые мы, однако, не собираемся здесь обсуждать. К ним относятся древовидные сети, процессоры в которых образуют бинарные деревья, а также гиперкубы, обобщающие двумерную решетку на три и более измерений.

7.1.3. Принципы анализа параллельных алгоритмов

При работе с параллельными алгоритмами нас будут интересовать два новых понятия: коэффициент ускорения и стоимость. Коэффициент ускорения параллельного алгоритма показывает, насколько он работает быстрее оптимального последовательного алгоритма. Так, мы видели, что оптимальный алгоритм сортировки требует $O(N \log N)$ операций. У параллельного алгоритма сортировки со сложностью $O(N)$ коэффициент ускорения составил бы $O(\log N)$.

Второе интересующее нас понятие — стоимость параллельного алгоритма, которую мы определяем как произведение сложности алгоритма на число используемых процессоров. Если в нашей ситуации алгоритм параллельной сортировки за $O(N)$ операций требует столько же процессоров, каково число входных записей, то его стоимость равна $O(N^{\hat{}})$. Это означает, что алгоритм параллельной сортировки обходится дороже, поскольку стоимость алгоритма последовательной сортировки на одном процессоре совпадает с его сложностью и равна поэтому $O(N \log N)$.

Близким понятием является расщепляемость задачи. Если единственная возможность для нашего алгоритма параллельной сортировки состоит в том, что число процессоров должно равняться числу входных записей, то такой алгоритм становится бесполезным как только число входных записей оказывается достаточно большим. Никаких похожих ограничений в алгоритме последовательной сортировки нет. По большей части нас будут интересовать такие алгоритмы параллельной сортировки, в которых число процессоров значительно меньше потенциального объема входных данных и это число не требует увеличения при росте длины входа.

7.1.4. Упражнения

1) В задаче, над которой Вы работаете, используются суммы величин из некоторой последовательности. Точнее говоря, для последовательности $\{s_1, s_2, s_3, \ldots, s_N\}$ Вам нужны суммы s_1 + S2,

s_1 + S2 + s_3, s_1 + S2 + s_3 + s_4, ..., s_1 + s_2 + s_3 + • • • + s_N. Разработайте параллельный подход к решению этой задачи.

2) Еще одной возможной конфигурацией системы сети является звездочка, в которой единственный центральный процессор соединен со всеми остальными. Нарисуйте звездочку из семи процессоров. Воспользовавшись анализом преимуществ и недостатков линейной сети в этом параграфе, что бы Вы сказали о преимуществах и недостатках звездочки?

7.2. Модель PRAM

При разработке параллельных алгоритмов мы опираемся на одну из четырех рассматриваемых нами моделей компьютерных систем. Одна из проблем параллельных систем состоит в чтении данных из памяти и записи в память. Что произойдет, например, если два процессора попытаются записать данные в одно и то же место общей памяти?

В рассмотренных нами в главах 2-6 алгоритмах предполагалось, что машина, на которой они реализуются, обладает прямым доступом к любой наперед заданной ячейке памяти (RAM). Алгоритмы этой главы будут основаны на параллельном варианте такой машины, называемом PRAM. Процессоры нашей PRAM-машины тесно связаны между собой и пользуются общим блоком памяти. В каждом процессоре есть несколько регистров, в которых может храниться небольшой объем данных, однако основная часть данных содержится в общей памяти.

Помимо требования тесной связи между процессорами мы также будем предполагать, что все они осуществляют один и тот же цикл обработки, состоящий в чтении данных из памяти, выполнении операции над ними и записи результата в память. Это означает, что все процессоры одновременно читают память, одновременно обрабатывают прочитанные данные и одновременно производят запись. Спор за ячейки памяти может возникнуть как при чтении, так и при записи. Требование трехшагового цикла означает, что мы можем не беспокоиться, что произойдет, если процессор Y меняет содержимое ячейки памяти в то время, как процессор X обрабатывает только что прочитанные оттуда данные. Не может также возникнуть и ситуация, в которой один процессор читает содержимое ячейки памяти в тот момент, когда другой пытается что-либо записать в нее.

Споры можно разрешать, предоставляя либо конкурентное, либо исключительное право доступа к памяти. При конкурентном доступе к одной и той же ячейке памяти может обращаться одновременно несколько процессоров. При исключительном доступе к данной ячейке памяти в каждый конкретный момент может обращаться только один процессор, а попытка двух одновременных обращений приводит к появлению сообщения об ошибке. Конкурентный доступ во время чтения не составляет проблемы. Нам, однако, понадобятся и алгоритмы, работающие с исключительным правом чтения. Если при исключительном праве чтения одновременно к одной и той же ячейке памяти обращается несколько процессоров, то возникает ошибка.

При записи также имеется проблема выбора между исключительным и конкурентным доступом. При исключительном доступе право записи в каждую ячейку памяти предоставляется одному процессору, а при попытке записи со стороны нескольких процессоров возникает ошибка. Однако два процессора могут вести запись в разные ячейки памяти одновременно. При конкурентном доступе задача оказывается более сложной: необходимо уметь разрешать возникающие конфликты. В модели с приоритетами каждому процессору приписывается приоритет, и право на запись предоставляется конкурирующему процессору с наивысшим приоритетом. В простейшем варианте этой модели приоритет процессора совпадает с его номером: чем меньше номер, тем выше приоритет. Если, например, в одну ячейку памяти пытаются записывать процессоры номер 4 и номер 7, то предпочтение будет отдано процессору 4. В модели с произвольным доступом будет выбран любой из конкурирующих процессоров -- заранее неизвестно какой. В обычной модели допускается одновременная запись, однако только в случае, когда все записываемые значения совпадают. В комбинированной модели система выполняет некоторые операции над записываемыми значениями: может быть записана их сумма, их произведение, наибольший или наименьший элемент, либо результат логической операции (и, или, исключительное **или**). В различных обстоятельствах любая из этих возможностей может оказаться полезной.

Тем самым у нас есть четыре комбинации возможностей чтения и записи: конкурентное чтение / конкурентная запись (CRCW), конкурентное чтение / исключительная запись (CREW), исключительное чтение / конкурентная запись (ERCW), исключительное чтение / исключительная запись (EREW).

7.2.1. Упражнения

1) Разработайте алгоритм суммирования N чисел в модели CREW PRAM. Насколько быстро работает Ваш алгоритм и какова его стоимость?

2) Разработайте алгоритм поиска наибольшего и наименьшего из чисел в модели CRCW PRAM, обратив особое внимание на механизм разрешения конфликтов. Насколько быстро работает Ваш алгоритм и какова его стоимость?

7.3. Простые параллельные операции

Рассмотрим теперь две элементарные операции: распределение входных данных по процессорам и поиск минимального или максимального элемента списка. Ниже мы обозначаем i-ый процессор через P_i, число процессоров через p, число элементов во входном списке через N, а j-ую ячейку памяти через M_j. Выполняемые параллельно операции записываются в скобках Parallel Start и Parallel End. Если сначала параллельно выполняется один блок операций, а затем другой, то они заключаются в отдельные пары скобок.

7.3.1. Распределение данных в модели CREW PRAM

Напомним, что в модели CREW одну и ту же ячейку памяти могут читать одновременно несколько процессоров. В результате значение можно занести во все процессоры очень быстро:

```
P[1] записывает значение в M[1]
Parallel Start
   for k=2 to p do
      P[k] читает значение из M[1]
   end for
ParallelEnd
```

Распределение совершается в два этапа. На первом значение записывается в память, на втором -- все процессоры читают его. Такая скорость возможна только благодаря конкурентному чтению. Посмотрим теперь, как должна выглядеть эта процедура для модели с исключительным чтением.

7.3.2. Распределение данных в модели EREW PRAM

В модели с исключительным чтением значение, записанное процессором PL, может прочитать лишь один процессор. Если мы просто организуем цикл поочередного чтения остальными процессорами, то получим последовательный алгоритм, нивелирующий все преимущества параллелизма. Однако если мы воспользуемся циклом чтение / обработка / запись на втором процессоре для записи значения в другую ячейку памяти, то на следующем шаге уже два процессора смогут прочитать это значение. Записав его затем еще в две ячейки, мы сможем воспользоваться уже четырьмя процессорами. В результате получается следующий алгоритм.

```
P[1] записывает значение в ячейку M[1]
procLoc=1
for j=1 to log p do
   Parallel Start
      for k=procLoc+1 to 2*procLoc do
         P[k] читает M[k-procLoc]
         P[k] записывает в M[k]
      end for k
   Parallel End
   procLoc=procLoc*2
end for j
```

Сначала алгоритм записывает значение в ячейку M_1. При первом проходе внешнего цикла процессор P2 читает это значение и записывает его в M2, а переменная procLoc принимает значение 2. На втором проходе процессоры P3 и P_4 читают содержимое ячеек M_1 и M_2 и записывают его в ячейки M3 и M_4, а переменная procLoc принимает значение 4. На третьем проходе процессоры с P_5 до P_8 читают содержимое ячеек с M_1 до M_4 и записывают его в ячейки с M_5 до M_8. Ясно, что (в предположении, что p является степенью двойки) на предпоследнем шаге половина процессоров прочитает нужное значение и запишет его в ячейки с M_1 до $M_{p/2}$, после чего оставшаяся половина процессоров сможет его прочитать. Поскольку шаги чтения и записи можно выполнять в одном цикле, как это было сделано в в начале § 7.2, параллельный блок выполняет один цикл, а внешний цикл производит $\log p$ итераций. Поэтому этот параллельный алгоритм распределения выполняет $O(\log p)$ операций.

7.3.3. Поиск максимального элемента списка

Здесь и при других действиях над списками мы будем предполагать, что список содержится в ячейках памяти с M_1 до M_N. Кроме того мы предполагаем, что в нашем распоряжении $p = N/2$ процессоров. (Случай $p < N/2$ мы обсудим после разработки алгоритма.)

На первом проходе процессор *Pi* сравнивает значения в ячейках M_{2i} и M_{2i+1} и записывает в ячейку M_i большее из них. На втором проходе нам нужна только половина процессоров, и с их помощью мы сравниваем пары элементов в ячейках с M_1 до $M_{N/2}$, записывая больший элемент пары в ячейки с M_1 по $M_{N/4}$. В результате получаем следующий алгоритм.

```
count=N/2
for i=1 to log(count)+1 do
   Parallel Start
      for j=1 to count do
         P[j] читает M[2j] в X и M[2j+1] в Y
         if X>Y
            P[j] пишет X в M[j]
         else
            P[j] пишет Y в M[j]
         end if
      end for j
   Parallel End
   count=count/2
end for i
```

Видно, что на каждом проходе цикла число значений, среди которых может находиться максимальное, уменьшается вдвое пока мы не получим единственное число. Это очень похоже на реализацию метода турниров на единственном процессоре. Если $p < N/2$, то мы можем предварительно уменьшить число значений до $2 * p$, а затем алгоритм выполняется так же, как и выше.

Видно, что число проходов цикла равно $\log N$, а его сложность имеет порядок $O(\log N)$. Напомним, о чем шла речь при обсуждении стоимости алгоритма в разделе 7.1.3. Мы определили стоимость как время работы, умноженное на число используемых процессоров. Поэтому стоимость нашего алгоритма равна $N/2 * O(\log N)$, или $O(N \log N)$. Простой последовательный алгоритм из главы 1 выполняет ту же работу за $O(N)$ операций, поэтому он дешевле, однако параллельный алгоритм работает гораздо быстрее.

Если параллельные вычисления действительно могут принести пользу, то должен существовать альтернативный подход, не уступающий по стоимости последовательному. Посмотрев на предыдущий вариант повнимательней, мы увидим, что число процессоров дает слишком большой вклад в стоимость. Можно ли как-то уменьшить это число? Если мы хотим добиться того, чтобы полная стоимость была порядка $O(N)$, а время исполнения параллельного алгоритма — порядка $O(\log \mathrm{TV})$, то число процессоров должно быть порядка $O(N/\log N)$. Это означает также, что на первом проходе каждый процессор должен обрабатывать $N/(N/\log \mathrm{TV}) = \log N$ значений. В результате мы приходим к следующему алгоритму.

```
Parallel Start
   for j = 1 to N/log N do
      P[j] находит максимум значений ячеек с M[1+(j-1)*log N]
           до M[j*log N] последовательным алгоритмом
      P[j] записывает найденный максимум в M[j]
   end for
Parallel End

count=(N/log N)/2
for i=1 to log(count)+1 do
   Parallel Start
      for j=1 to count do
         PCj] читает M[2j] в X и M[2j+1] в Y
         if X>Y
            P[j] пишет X в M[j]
         else
            P[j] пишет Y в M[j]
         end if
      end for j
   ParallelEnd
   count=count/2
end for i
```

В этом варианте имеется этап предварительной обработки, на котором каждый процессор выполняет последовательный алгоритм на списке из $\log N$ элементов; это требует, как Вы помните, $O(\log \mathrm{TV})$ операций. Во второй части алгоритма воспроизведена наша исходная попытка, где число требуемых процессоров уменьшилось до $(\mathrm{TV}/\log N)/2$ (хотя

на этапе предварительной обработки по-прежнему требуется $N/\log N$ процессоров). Поэтому полная стоимость алгоритма равна

Шаг	Стоимость	Время
Предварительная обработка	$(N/\log N) * O(\log N) = O(N)$	$O(\log N)$
Основной цикл	$[(N/\log N)/2] * O(\log N) = O(N)$	$O(\log N)$
Всего	$O(N)$	$O(\log N)$

Стоимость последнего параллельного варианта алгоритма такая же, как и у последовательного алгоритма, однако работает он быстрее.

7.3.4. Упражнения

1) Медиана в наборе чисел — это такое значение, что половина из чисел набора меньше, а другая больше него. Другими словами, если бы значения в списке были отсортированы, то медиана занимала бы в нем в точности среднее положение. Разработайте параллельный алгоритм поиска медианы в модели CREW PRAM. Насколько быстро работает Ваш алгоритм и какова его стоимость?

2) Разработайте параллельный алгоритм поиска медианы в модели CRCW PRAM (определение медианы см. в упражнении 1). Обратите особое внимание на механизм разрешения конфликтов при записи. Насколько быстро работает Ваш алгоритм и какова его стоимость?

7.4. Параллельный поиск

Изучение параллельных методов поиска мы начнем с наивной попытки, в которой число процессоров равно числу элементов списка. Анализ покажет нам, насколько далеко по стоимости такое решение от оптимального. Затем мы попробуем уменьшить стоимость за счет уменьшения числа процессоров, как это было сделано при вычислении максимума. Будем предполагать, что в списке нет дубликатов.

Если число процессоров равно числу элементов списка ($p = N$), то каждый процессор может сравнивать искомое значение со своим элементом списка. Если произошло совпадение, то процессор, обнаруживший

это совпадение, может записать номер ячейки в некотором специальном месте памяти. В следующем алгоритме предполагается, что список расположен в ячейках с M_1 по M_N, искомое значение — в ячейке M_{N+1}, а номер ячейки, в которой оно обнаружено, должен быть записан в M_{N+2}.

```
Parallel Sort
   for j=1 to N do
      P[j] читает M[j] в X и M[N+1] в target
      if X=target then
         записать j в M[N+2]
      end if
   end for
ParallelEnd
```

Мы предполагаем, что во все пустые ячейки памяти первоначально занесено нулевое значение, поэтому по окончании работы алгоритма в ячейке M_{N+2} будет нуль, если искомое значение в списке не обнаружено. Если же значение в списке найдено, то единственный обнаруживший его процессор запишет в M_{N+2} номер содержащей его ячейки.

На каждом из N процессоров этот алгоритм выполняет один цикл чтение / обработка / запись, поэтому общее время работы равно $O(1)$, а стоимость равна $O(N)$. Напомним, что оптимальный последовательный алгоритм поиска — двоичный поиск из главы 2 — имел стоимость $O(\log N)$.

Излагаемый ниже альтернативный параллельный алгоритм позволяет варьировать стоимость и время работы в зависимости от числа имеющихся процессоров.

```
При наличии p<=N процессоров
Parallel Start
   for j=1 to p do
      P[j] выполняет последовательный двоичный поиск
           в ячейках с M[(j-1)*(N/p)+1] по M[j*(N/p)]
           и записывает номер ячейки, содержащей X, в M[N+2]
   end for
ParallelEnd
```

При наличии одного процессора он будет вести поиск в ячейках с M_1 до M_N, т.е. во всем списке. Поэтому на одном процессоре этот алгоритм сводится к последовательному двоичному поиску. Значит его стоимость равна $O(\log N)$, а время выполнения тоже $O(\log N)$.

При использовании N процессоров мы возвращаемся к первому параллельному варианту со стоимостью $O(N)$ и временем выполнения $O(1)$. В промежуточных вариантах мы имеем дело с p списками, содержащими N/p элементов каждый. Время выполнения такого варианта равно $O(\log(N/p))$, а его стоимость $O(p\log(N/p))$. Особо стоит отметить случай p — log TV, когда $\log(N/\log TV) = \log N - \log(\log N)$ и $\log N$. В этом случае время выполнения равно $O(\log N)$ при стоимости $O((\log N) * (\log N)) = O(\log^2 N)$.

Порядок времени выполнения параллельного алгоритма совпадает с порядком последовательного двоичного поиска, однако константа в оценке оказывается меньше, поэтому параллельный алгоритм будет выполняться быстрее. Стоимость $O(\log^2 N)$ превышает стоимость $O(\log N)$ оптимального последовательного поиска, однако не настолько, чтобы сделать параллельный алгоритм бессмысленным.

7.4.1. Упражнения

1) В обсуждавшихся выше алгоритмах поиска предполагалось, что в списке нет дубликатов. Если же это не так и искомое значение встречается в списке несколько раз, то обычно принято возвращать номер его первого появления. Какие изменения следует произвести в обоих алгоритмах поиска в модели CREW PRAM, чтобы обеспечить обработку списков с дубликатами?

2) Разработайте параллельный алгоритм поиска в списке с дубликатами в модели CRCW PRAM. Если искомое значение встречается несколько раз, то обычно принято возвращать номер его первого появления. Обратите особое внимание на механизм разрешения конфликтов при записи. Насколько быстро работает Ваш алгоритм и какова его стоимость?

7.5. Параллельная сортировка

Параллельную сортировку можно осуществлять различными способами. В этом параграфе мы подробно изучим два способа и обсудим в общем другие подходы, которые не вошли в нашу книгу.

7.5.1. Сортировка на линейных сетях

Начнем с метода сортировки, основанного на линейной конфигурации сети. Если число процессоров равно числу сортируемых значений, то сортировку можно осуществить, передавая сети в каждом цикле одно значение. Первый процессор читает поданное значение, сравнивает его с текущим и передает большее значение своему соседу. Остальные процессоры делают то же самое: сохраняют меньшее из двух значений и пересылают большее следующему звену в цепочке. Говоря более формально, мы получили такой алгоритм:

```
for i=1 to N-1 do
   занести следующее значение в M[1]
   Parallel Start
      P[j] читает M[j] в Current
      for k=1 to j-1 do
         P[k] читает M[k] в New
         if Current>New then
            P[k] пишет Current в M[k+1]
            Current=New
         else
            P[k] пишет New в M[k+1]
         end if
      end for k
   Parallel End
   занести следующее значение в M[1]
   Parallel Start
      P[j] читает M[j] в Current
      for k=1 to j do
         P[k] читает M[k] в New
         if Current>New then
            P[k] пишет Current в M[k+1]
            Current=New
         else
            P[k] пишет New в M[k+1]
         end if
      end for k
   Parallel End
Parallel Start
   for j=1 to N-1 do
      P[j] пишет Current в M[j]
```

```
   end for j
ParallelEnd
```

Перед каждым параллельным проходом очередной элемент списка (если в списке еще остались элементы) заносится в первую ячейку памяти. В самом начале это значение просто читается первым процессором. На последующих этапах первый процессор читает следующее значение в переменную New, сравнивает ее со значением переменной Current в процессоре, а затем записывает большее из двух значений во вторую ячейку памяти. Во внешнем цикл **for** имеется два параллельных блока, поскольку при каждом проходе этого цикла к сортировке привлекается очередной процессор; в первом блоке этот процессор читает первое для себя значение, а во втором блоке он вовлекается в процесс сравнения. В самом конце все значения опять записываются в память.

На рис. 7.4 (а), (б) и (в) изображена работа этого алгоритма на входном списке 15, 18, 13, 12, 17, 11, 19, 16, 14. Видно, что на этапе A первый элемент списка заносится в память и его читает первый процессор. На этапе B в память заносится второе значение, оно сравнивается с текущим значением в процессоре P_1, и большее из двух значений заносится в M_2. На этапе C в M_1 заносится третий элемент списка, который сравнивается с текущим значением в процессоре P_1, а процессор P_2 производит первую операцию чтения из M_2. На шаге D сравнение могут делать уже оба процессора $P\backslash$ и P_2. Так, на каждом «нечетном» шаге новый процессор собирается выполнить первое чтение, а на «четном» шаге все активные процессоры выполняют сравнения.

Из рис. 7.4 видно, что сортировка поданного списка требует 16 параллельных шагов, и 1 шаг нужен для записи результата. В общем случае алгоритм выполняет $2 * (N - 1) + 1$, т.е. $O(N)$, действий. В работе принимают участие N процессоров, поэтому стоимость алгоритма равна $O(N\ \hat{})$ — она такая же, как и у самых медленных наших сортировок.

7.5.2. Четно-нечетная сортировка перестановками

Выше нам удавалось снизить стоимость алгоритмов за счет сокращения числа процессоров. Необходимое число процессоров можно уменьшить вдвое, воспользовавшись следующим способом сортировки, который сравнивает соседние значения и при необходимости переставляет их.

```
for j=1 to N/2 do
   Parallel Start
```

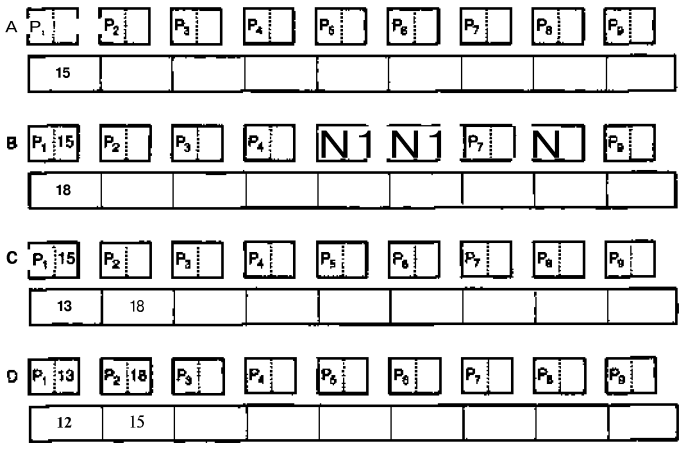


Рис. 7.4 (a). Параллельная сортировка на линейной сети

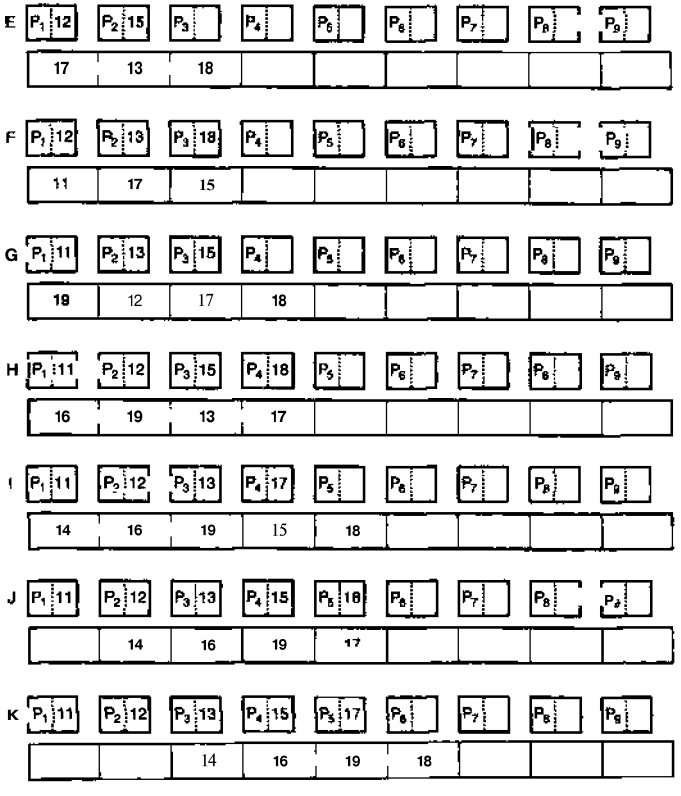


Рис. 7.4 *(б)*. Параллельная сортировка на линейной сети

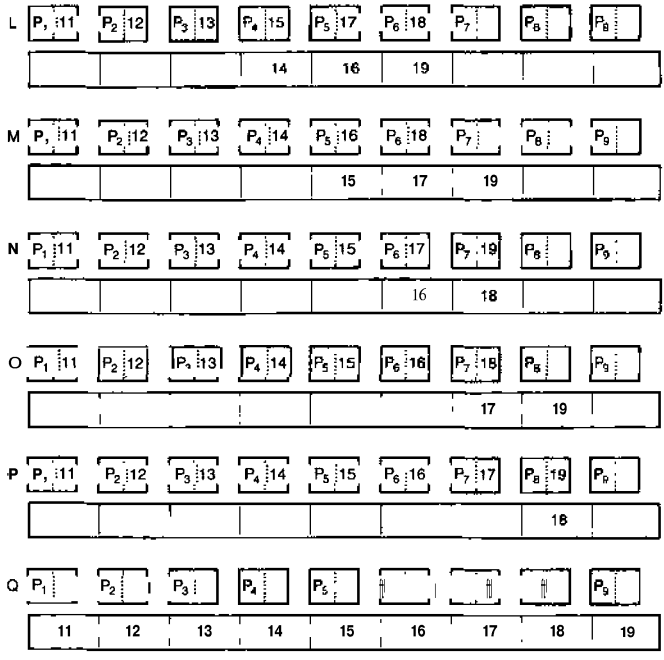


Рис. 7.4 *(в)*. Параллельная сортировка на линейной сети

```
      for k=l to N/2 do
         P[k] сравнивает M[2k-1] и M[2k]
         if их порядок неправильный then
            переставить их
         end if
      end for k
   Parallel End
   Parallel Start
      for k=l to N/2-1 do
         P[k] сравнивает M[2k] и M[2k+1]
         if их порядок неправильный then
            переставить их
         end if
      end for k
   Parallel End
end for j
```

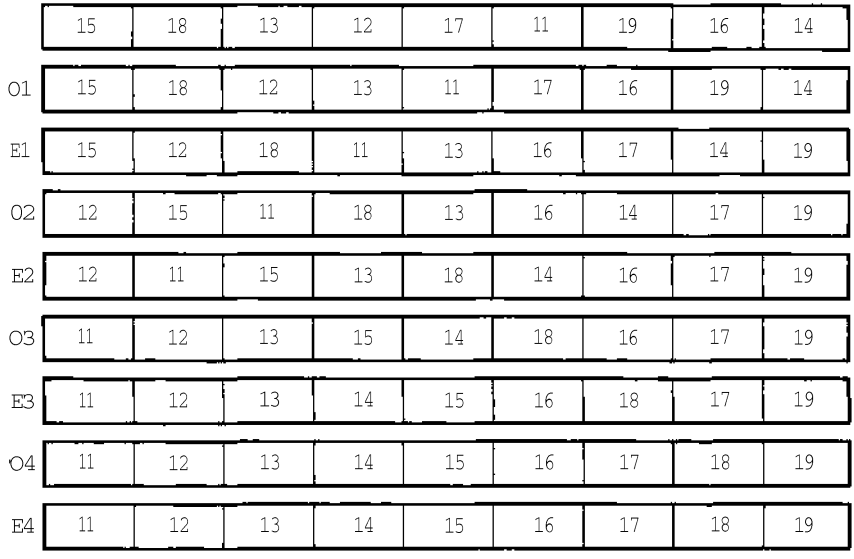


Рис. 7.5. Четно-нечетная параллельная сортировка

При каждом проходе этот алгоритм сначала сравнивает M_1 с M_2, Мз с M_4, ..., M_{N-1} с M_N; затем M_2 сравнивается с Мз, M_4 с M_5, ..., M_{N-2} с M_{N-1},и элементы, стоящие в неправильном порядке, переставляются. Предположим теперь, что наименьший элемент стоит в списке последним. Первое сравнение переведет его в предпоследнюю позицию, второе — во вторую с конца. Каждый проход алгоритма сдвигает минимальный элемент списка на две позиции ближе к требуемому положению. Повторение цикла *N/2* раз сдвигает минимальный элемент на $N - 1$ позицию и ставит его в нужное место.

На рис. 7.5 изображено выполнение этой процедуры сортировки на нашем списке 15, 18, 13, 12, 17, 11, 19, 16, 14. Строчки изображают результат либо нечетного (метка О), либо четного (метка Е) шага.

Все сравнения происходят параллельно, поэтому всякий проход цикла выполняет два сравнения и общее время работы равно $O(N)$. Стоимость алгоритма равна величине $N/2^*$ *O(N)* — несколько меньше, чем у предыдущего алгоритма, но все равно порядка $O(N^2)$.

7.5.3. Другие параллельные сортировки

Список без повторений можно отсортировать также с помощью подсчета. При сравнении каждого элемента списка со всеми остальными

его элементами мы можем подсчитать, сколько в списке значений, меньших него, и определить тем самым правильное его положение: число меньших элементов следует увеличить на 1. В модели CREW PRAM с одним процессором на каждое входное значение местоположение каждого элемента можно определить за $O(N)$ сравнений. Нам требуется N процессоров, поэтому стоимость такого подхода равна $O(N^2)$.

Существуют способы параллельного слияния двух списков, о которых мы не рассказываем, реализующие оптимальную стоимость $O(N)$ при использовании не более чем $N/\log N$ процессоров. Разбив список на $N/\log N$ частей, и отсортировав каждую из них эффективным последовательным алгоритмом сортировки, например Quicksort, можно затем параллельно слить эти части в общий список. Параллельное слияние можно использовать и в сортировке слиянием.

7.5.4. Упражнения

1) Запишите формально алгоритм сортировки подсчетом, описанный в разделе 7.5.3. Проанализируйте этот алгоритм с точки зрения скорости и стоимости.

2) Запишите алгоритм сортировки слиянием, описанный в § 3.6, вставив в него вызов параллельного слияния `ParallelMergeLists(i, j, k, l)`; этот вызов сливает подсписки из ячеек от Mj до Mj и от M_k до M_l. Проанализируйте этот алгоритм с точки зрения скорости и стоимости. Можно предполагать, что ParallelMergeLists выполняется на N процессорах за $\log N + 1$ шагов, где N - - число элементов в слитом списке $(N = j - i + l - k + 2)$.

3) Запишите формально параллельный алгоритм, который разбивает список на $N/\log N$ частей, каждая из которых сортируется алгоритмом Quicksort, а затем сливает отсортированные части. Для выполнения быстрой сортировки подсписка от $M[j]$до $M[k]$ алгоритм осуществляет вызов Quicksort (M, j `,k`). Воспользуйтесь кроме того вызовом ParallelMergeLists, описанным в упражнении 2.

7.6. Параллельные численные алгоритмы

В этом параграфе мы применяем параллельные алгоритмы к решению численных задач. Начав с двух классов параллельных алгоритмов умножения матриц, мы переходим затем к решению систем линейных уравнений.

7.6.1. Умножение матриц в параллельных сетях

Один из способов параллельного матричного умножения пользуется сетью, размер которой связан с величиной матриц. Для умножения I x J-матрицы на J x K-матрицу потребуется двумерная решетка процессоров размером / x K, строчки которой соответствуют строчкам первой матрицы, а столбцы – столбцам второй.

Элементы первой матрицы будут поочередно поступать в строчки сети. Первая строчка попадет туда при первом проходе, вторая — при втором, и т.д.

Аналогичные действия будут выполняться с элементами второй матрицы и колонками сети, начиная с первой колонки матрицы. Запаздывание с поступлением элементов последующих строк и столбцов оказывается как раз таким, что числа, которые должны быть перемножены каждым процессором, поступают в него одновременно. Процессор умножает два числа, поступающие в него в очередном цикле, и прибавляет результат к текущему значению. В конце концов в каждом процессоре окажется одно результирующее значение. Этапы умножения изображены на рис. 7.6 (а)-(и).

Анализ

Этот алгоритм выполняет умножение 2 x 3-матрицы на 3×4-матрицу за семь шагов. Для подсчета числа шагов в общем случае можно вывести формулу. Рассмотрим сначала сам процесс умножения. Один из способов подсчета числа операций состоит в том, чтобы вычислить число шагов, необходимых для полного прогона по сети последнего элемента последней строки первой матрицы. То же самое вычисление работает и для последнего элемента последнего столбца второй матрицы.

Строки первой матрицы сдвигаются на один столбец при каждом цикле. Если в первой матрице I строчек и J столбцов, то сколько циклов потребуется на то, чтобы последний элемент первой строки покинул сеть? Каждый цикл добавляет в сеть одно число, а первое число поступает в сеть на первом цикле, последний элемент первой строки

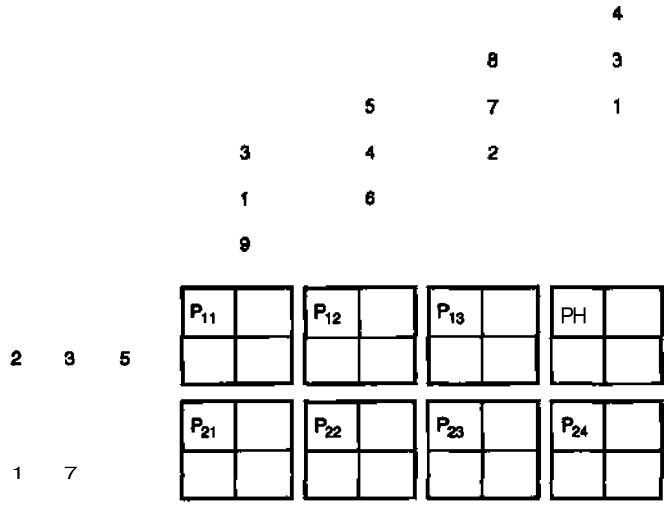


Рис. 7.6 *(а)*. Исходная решетчатая конфигурация сети

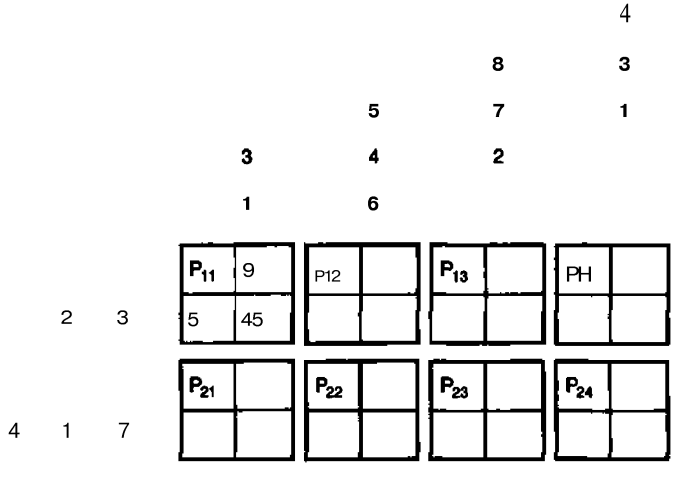


Рис. 7.6 (*б*). Процессор P_{11} перемножает первые два поступивших значения и записывает их в своих регистрах

будет находиться в сети еще J — 1 цикл. Сколько дополнительных циклов потребуется, чтобы этот элемент покинул сеть, когда мы закончим работу с ним? Это число нужно умножить на одно значение из каждого столбца J x K-матрицы, поэтому на то, чтобы оно покинуло сеть, требуется K циклов. Из-за задержки очередной строки и очередного столбца теперь, когда мы поняли, как обрабатывается первая строка, нам необходимо понять, что происходит с последней строкой.

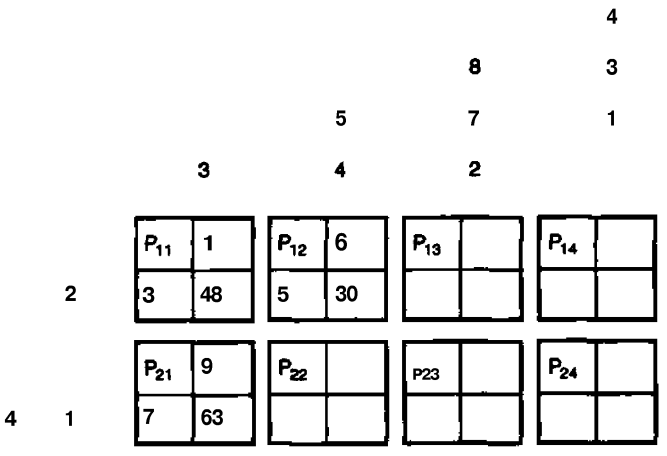


Рис. 7.6 (в). Процессор $Pц$ перемножает следующие два значения и прибавляет произведение к значению регистра; процессоры P_{12} и P_{21} перемножают первые поступившие к ним пары значений

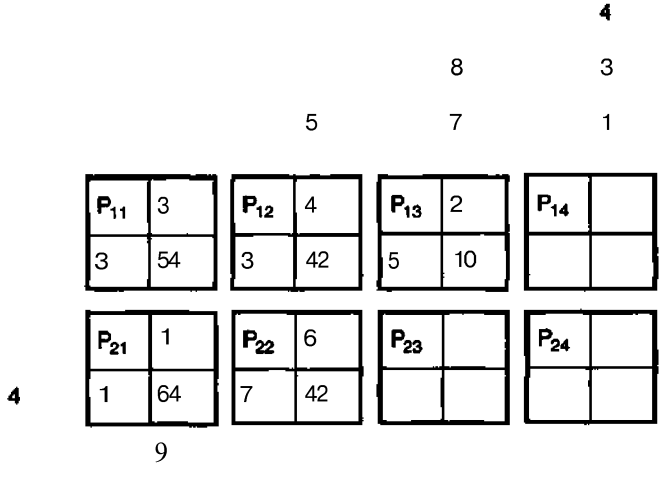


Рис. 7.6 (г). P_{11}, P_{12} и P_{21} обрабатывают очередные пары чисел, а P_{13} и P_{22} присоединяются к процессу обработки

Как мы уже говорили, обработка очередной строки первой матрицы начинается первым циклом после начала обработки предыдущей строки. Вторая строка начинает обрабатываться на втором цикле, третья — на третьем, последняя — на цикле с номером I. Мы уже гово-

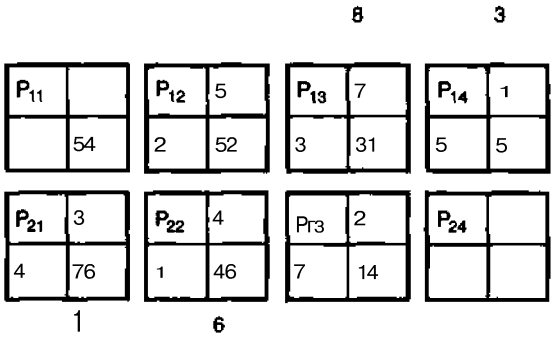


Рис. 7.6 (*д*). P_{11} завершил работу, значение верхнего левого элемента произведения занесено в его регистр; P_{12}, P_{21}, P_{13} и P_{22} обрабатывают очередные пары чисел, а P_{14} и P_{23} присоединяются к процессу обработки

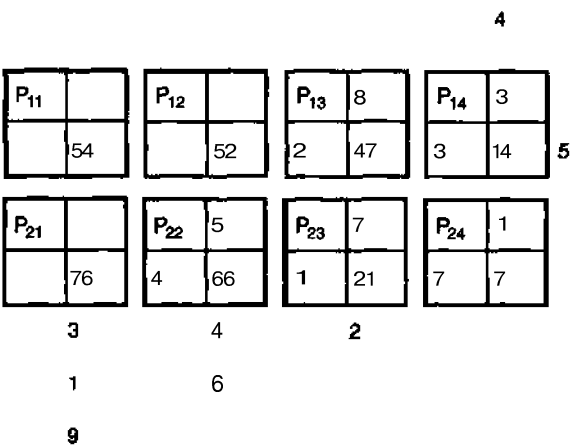


Рис. 7.6 (*е*). P_{12} и P21 завершили работу; P24 получил первую пару чисел, а P_{13},*p22,* P_{14} и P_{23} работают над очередными значениями

рили о том, что последнему элементу строки требуется еще J — 1 цикл, чтобы попасть в сеть, и еще K циклов нужно ему, чтобы выйти из сети. Значит весь процесс завершится за $I + J + K - 1$ цикл. Время выполнения нашего сетевого умножения поэтому имеет порядок $O(N)$, где $N = \max(I, J, K)$. Число используемых процессоров равно $O(N^2)$, поэтому стоимость параллельного алгоритма равна $O(N^3)$, т.е. она совпадает со стоимостью стандартного алгоритма матричного умножения. Основное его достоинство, однако, в том, что время его выполнения значительно меньше, чем у любого из последовательных алгоритмов,

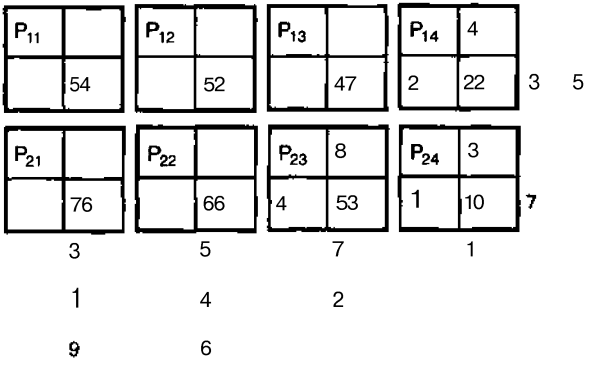


Рис. 7.6 *(ж)*. P_{13} и P_{22} завершили работу, а *Рц*, P_{23} и P_{24} работают над очередными значениями

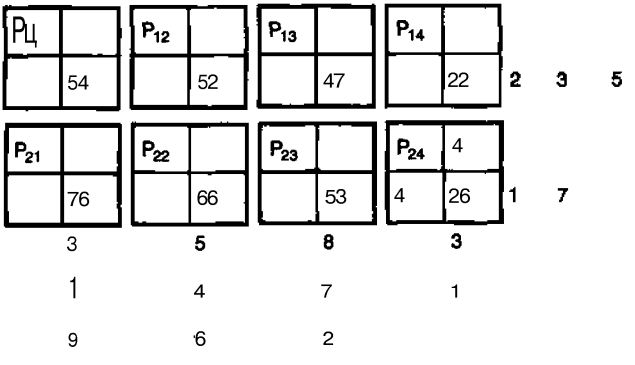


Рис. 7.6 (з). P_{14} и P_{23} завершили работу, P_{24} работает над последними двумя значениями

рассматривавшихся в главе 4. Любой алгоритм, интенсивно использующий умножение матриц, будет работать на двумерной сети гораздо быстрее. Например, в главе 4 мы обсуждали алгоритм свертки, умножающий 5 x 5-матрицу на всякий 5 x 5-участок изображения размером 512×512. Стандартный последовательный алгоритм умножения матриц потребовал бы 32 258 800 умножений, каждое из которых составляет отдельный цикл — всего 32 258 800 циклов. На сети размером 5×5 (25 процессоров) число умножений было бы тем же самым, однако все действия были бы совершены за 3 612 896 циклов — почти 90%-ая экономия времени.

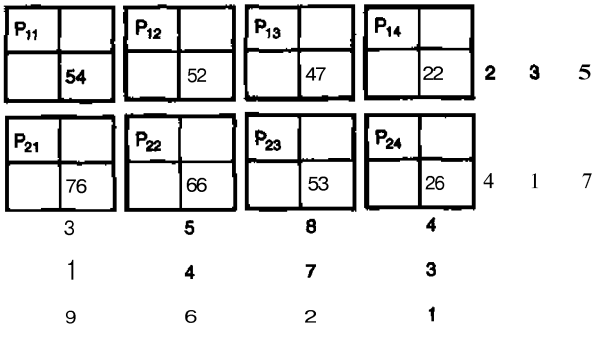


Рис. 7.6 *(и)*. Работа завершена и в процессорах записан результат

7.6.2. Умножение матриц в модели CRCW PRAM

Параллельное умножение матриц в комбинированной модели CRCW PRAM, в которой при конкурентной записи в одну и ту же ячейку памяти записываемые значения складываются, при достаточном числе процессоров выполняется за постоянное время. Умножение *I* x J-матрицы *A* на J x *K*-матрицу *B* потребует *I* * J * *K* процессоров. Каждый из этих процессоров будет выполнять в точности одно из $O(N^3)$ умножений, необходимых для получения окончательного результата:

```
Parallel Start
   for x=1 to I do
      for y=1 to J do
         for z=1 to K do
            P[x,y,z] вычисляет A[x,y]*B[y,z]
                     и записывает результат в M[x,z]
         end for z
      end for y
   end for x
Parallel End
```

Анализ

Как уже упоминалось, каждый из процессоров производит одно умножение и записывает его результат в нужную ячейку памяти. На это уходит один цикл. В каждую ячейку памяти, предназначенную для хранения результата, конкурентную запись производят J процессоров. Мы потребовали, чтобы эта конкурентная модель записи складывала

все записываемые значения, поэтому другая часть стандартного алгоритма умножения матриц — сложение — выполняется при записи.

Этот алгоритм производит умножение матриц за время $O(1)$ и использует $O(N^3)$ процессоров, где $N = \max(I, J, K)$. Поэтому его стоимость равна $O(N^3)$ — такая же, как и у стандартного последовательного алгоритма. Сокращение же времени выполнения даже еще более принципиальное, чем у алгоритма умножения на двумерной решетке. Так, возвращаясь к нашему примеру со сверткой, мы видим, что однократное умножение матриц на 125 процессорах происходит за один цикл. Поэтому свертка всего изображения займет всего 258 064 циклов, что в 125 раз быстрее последовательного варианта.

7.6.3. Решение систем линейных уравнений алгоритмом SIMD

В § 4.3 мы построили последовательный алгоритм решения системы N линейных уравнений с N неизвестными методом Гаусса-Жордана. В этом алгоритме мы записывали систему линейных уравнений в виде матрицы с N строками и $N + 1$ столбцами и выполняли над строками некоторые операции до тех пор пока не получали единичную матрицу в первых N строках и N столбцах. Тогда в последнем столбце матрицы оказывались искомые значения неизвестных. Теперь мы приведем SIMD-алгоритм в модели CREW PRAM, реализующий тот же самый подход. В нижеследующем обсуждении мы предполагаем, что в ячейке памяти *Mij* хранится коэффициент при j-ом неизвестном (j-ый столбец) в i-ом уравнении (i-ая строка).

Прежде, чем изложить параллельный алгоритм, напомним последовательный алгоритм из § 4.3. Процесс начинается с деления первой строки на ее первый элемент. Так, если первый элемент первой строки равен 5, то нужно разделить все элементы первой строки на 5. Затем последовательный алгоритм вычитает из всех остальных строчек первую строку, умноженную на первый элемент этих строчек. Например, если первый элемент второй строки равен 12, то из второй строки нужно вычесть первую, умноженную на 12.

Следующий алгоритм в модели CREW PRAM работает на $N * (N + 1)$ процессорах, каждый из которых обновляет один элемент матрицы. Как и в последовательном алгоритме, реализующем метод Гаусса–Жордана, этот параллельный алгоритм не занимается ошибками округления или матричными особенностями.

```
for x=1 to N do
   Parallel Start
      for y=x to N+1 do
         P[x,y] читает M[x,x] в factor, а M[x,y] в value
         P[x,y] вычисляет value/factor и записывает в M[x,y]
      end for y
   ParallelEnd
   Parallel Start
      for y=x to N+1 do
         for z=1 to N do
            if x/=z
               P[z,y] читает M[z,y] в current,
                      M[z,x] в factor, а M[x,y] в value
               P[x,y] вычисляет current-value/factor
                      и записывает результат в M[z,y]
            end if
         end for z
      end for y
   ParallelEnd
end for x
```

В этом алгоритме внешний цикл итерируется по всем неизвестным. Первый параллельный блок делит элементы текущей строки на требуемый коэффициент. Затем второй параллельный блок вычитает подходящее кратное текущей строки из всех остальных строк. В этом втором блоке внешний цикл обрабатывает те столбцы, которым еще требуется обработка, а внутренний цикл выполняет обработку всех уравнений.

7.6.4. Упражнения

1) Выполните трассировку умножения матриц на плоской решетке аналогично рис. 7.6 для умножения матриц

$$\begin{bmatrix} 2 & 3 \\ 7 & 4 \end{bmatrix} \text{ и } \begin{bmatrix} 5 & 1 \\ 2 & 9 \end{bmatrix}.$$

2) Выполните трассировку умножения матриц на плоской решетке аналогично рис. 7.6 для умножения матриц

$$\begin{bmatrix} 8 & 2 & 1 \\ 3 & 5 & \end{bmatrix} \text{ и } \begin{bmatrix} 3 & 2 & 1 \\ 7 & 4 & \end{bmatrix}.$$

3) Выполните трассировку умножения матриц на плоской решетке аналогично рис. 7.6 для умножения матриц

$$\begin{bmatrix} 1 & 5 \\ 4 & 6 \\ 7 & 2 \end{bmatrix} \text{ и } \begin{bmatrix} 0 & 2 & 3 \\ 5 & 1 & 9 \end{bmatrix}.$$

4) Подсчитайте время выполнения и стоимость параллельного метода Гаусса-Жордана решения системы линейных уравнений. Подсчет должен учитывать как число умножений и делений, так и число сложений и вычитаний. Сопоставьте Ваши результаты с параллельной реализацией метода Гаусса-Жордана.

7.7. Параллельные алгоритмы на графах

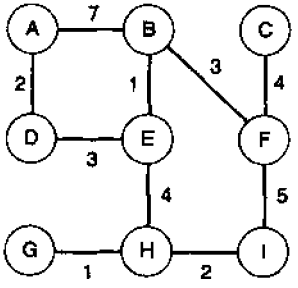
При исследовании параллельных алгоритмов на графах будем представлять графы матрицами примыканий. Мы займемся некоторыми замечательными связями между матричными операциями над матрицами примыканий и свойствами соответствующих графов.

7.7.1. Параллельный алгоритм поиска кратчайшего пути

Мы определили матрицу примыканий взвешенного графа следующим образом:

$$\text{AdjMat}[i,j] = \begin{cases} w_{ij} & \text{при } v_i v_j \in E, \\ 0 & \text{при } i = j, \\ \infty & \text{при } v_i v_j \notin E \end{cases} \qquad \text{для } i,j \in [1, N].$$

На рис. 7.7 изображен взвешенный граф и его матрица примыканий. В матрице примыканий представлены прямые пути между вершинами графа, т.е. пути, состоящие из одного ребра. Нас интересуют кратчайшие пути по графу, состоящие из произвольного числа ребер; попробуем восстановить их по матрице примыканий. В исходной матрице $A = A^1$ содержатся длины кратчайших путей из нуля или одного ребра. В матрице A^2 будут записаны длины кратчайших путей из не более, чем j ребер. Ясно, что в матрице A^{N-1} будут записаны длины всех кратчайших путей с произвольным числом ребер, потому что всякий путь из N и более ребер содержит цикл, а значит кратчайший путь должен содержать не более $N - 1$ ребер.



$$\begin{bmatrix} 0 & 7 & \infty & 2 & \infty & \infty & \infty & \infty & \infty \\ 7 & 0 & \infty & \infty & 1 & 3 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & 4 & \infty & \infty & \infty \\ 2 & \infty & \infty & 0 & 3 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & 3 & 0 & \infty & \infty & 4 & \infty \\ \infty & 3 & 4 & \infty & \infty & 0 & \infty & \infty & 5 \\ \infty & \infty & \infty & \infty & \infty & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & 4 & \infty & 1 & 0 & 2 \\ \infty & \infty & \infty & \infty & \infty & 5 & \infty & 2 & 0 \end{bmatrix}$$

Рис. 7.7. Взвешенный граф и его матрица примыканий

Подумаем, как можно построить матрицу A^2 по матрице A^1. Кратчайший путь из двух ребер между вершинами x и y проходит еще в точности через одну вершину. Например, всякий путь длины два между вершинами A и E проходит либо через вершину B, либо через вершину D. Сравнив сумму весов ребер AB и BE с суммой весов ребер AD и DE, мы видим, что путь через вершину D короче пути через вершину B. В общем случае, если посмотреть на сумму весов ребер A* и E*, где * пробегает все вершины от A до I (за исключением самих вершин A и E), то минимальное значение суммы весов будет давать длину кратчайшего пути из двух или менее ребер. Отсюда получаем

$$A^2_{ij} = \min_{k \in V}(A^1_{ik} + A^1_{kj}).$$

Применив эту процедуру к матрице с рис. 7.7, мы получим матрицу с рис. 7.8.

Матрицу A^3 можно построить по матрицам A^1 и A^2, заметив, что кратчайший путь из трех или менее ребер должен состоять из кратчайшего пути из двух или менее ребер, за которым следует кратчайший путь из одного или менее ребер и наоборот. Матрицу A^4 можно построить либо по матрицам A^1 и A^3, либо только по матрице A^2. Поэтому до конца добраться легче, вычисляя матрицы A^2, A^4, A^8, ... , $A^{N'}$, где N' — первая степень двойки которая превышает число вершин, уменьшенное на 1. Все кратчайшие расстояния в графе с рис. 7.6 мы получим, подсчитав матрицу A^8.

Параллельный подсчет кратчайших расстояний в графе можно реализовать на основе этих матричных операций. Заменив в алгоритме

$$\begin{bmatrix} 0 & 7 & \infty & 2 & 5 & 10 & \infty & \infty & \infty \\ 7 & 0 & 7 & 4 & 1 & 3 & \infty & 5 & 8 \\ \infty & 7 & 0 & \infty & \infty & 4 & \infty & \infty & 9 \\ 2 & 4 & \infty & 0 & 3 & \infty & \infty & 7 & \infty \\ 5 & 1 & \infty & 3 & 0 & 4 & 5 & 4 & 6 \\ 10 & 3 & 4 & \infty & 4 & 0 & \infty & 7 & 5 \\ \infty & \infty & \infty & \infty & 5 & \infty & 0 & 1 & 3 \\ \infty & 5 & \infty & 7 & 4 & 7 & 1 & 0 & 2 \\ \infty & 8 & 9 & \infty & 6 & 5 & 3 & 2 & 0 \end{bmatrix}$$

Рис. 7.8. Матрица A^2 для взвешенного графа с рис. 7.7

умножения матриц сложение операцией взятия минимума, а умножение сложением, мы получим алгоритм, вычисляющий нужные нам матрицы. Применение модифицированного алгоритма матричного умножения к матрицам A^1 и A^1 даст матрицу A^2, *а к* матрицам A^2 и A^2 — матрицу A^4. Теперь параллельный алгоритм подсчета кратчайших расстояний превращается просто в параллельный алгоритм умножения матриц, поэтому анализ последнего применим и в данном случае.

7.7.2. Параллельный алгоритм поиска минимального остовного дерева

Напомним, что алгоритм Дейкстры-Прима порождения минимального остовного дерева (МОД) строит дерево постепенно, добавляя на каждом шаге вершину, соединенную с уже построенной частью дерева ребром минимальной длины. При этом вершины, расположенные рядом с уже построенной частью дерева, образуют «кайму». Располагая большим числом процессоров, мы можем вместо этого рассматривать одновременно все вершины и при каждом проходе выбирать ближайшую из них.

Спроектируем алгоритм для p процессоров, предполагая, что $p < N$; это означает, что каждый процессор будет отвечать з а N/pвершин. Начнем построение МОД с одной из вершин; в начальный момент это ближайшая ко всем остальным вершина дерева, поскольку она единственная. При каждом проходе каждый процессор исследует каждую из вершин и выбирает из них ближайшую к вершинам дерева. Затем процессор передает информацию о выбранной вершине центральному процессору, который выбирает среди них вершину

с абсолютно минимальным расстоянием до построенной части дерева. Эта вершина добавляется к дереву, а информация о ней передается остальным процессорам, и они обновляют свои данные о дереве. Процесс повторяется $N - 1$ раз пока к дереву не будут добавлены все вершины.

В следующем алгоритме множество вершин, за которые отвечает процессор P_j, обозначается через V_j, а через v_k обозначается вершина графа. В каждом из процессоров локально используется два массива. Элемент `closest(v)` первого из них содержит метку вершины построенной части МОД, ближайшей к вершине v, а элемент `distance(v[i],v[j])` - кратчайшее расстояние между вершинами *Vi* и *Vj*. Вот как выглядит этот алгоритм в модели CREW PRAM:

```
v[0] помечается как первая вершина дерева
Parallel Start
   for j=1 to p do
      for каждой вершины в V[j] do
         занести v[0] в closest
      end for
   end for j
Parallel End

for j=1 to N-1 do
   Parallel Start
      for k=1 to p do
         P[k] вычисляет наименьшие расстояния
            от своих вершин до МОД
         P[k] возвращает distance(v,closest(v)), v
            и closest(v)
      end for k
Parallel End

Центральный процессор P находит наименьшее из найденных
   расстояний и добавляет соответствующую вершину v в МОД
Центральный процессор P сообщает вновь добавленную
   вершину v всем остальным процессорам

Parallel Start
```

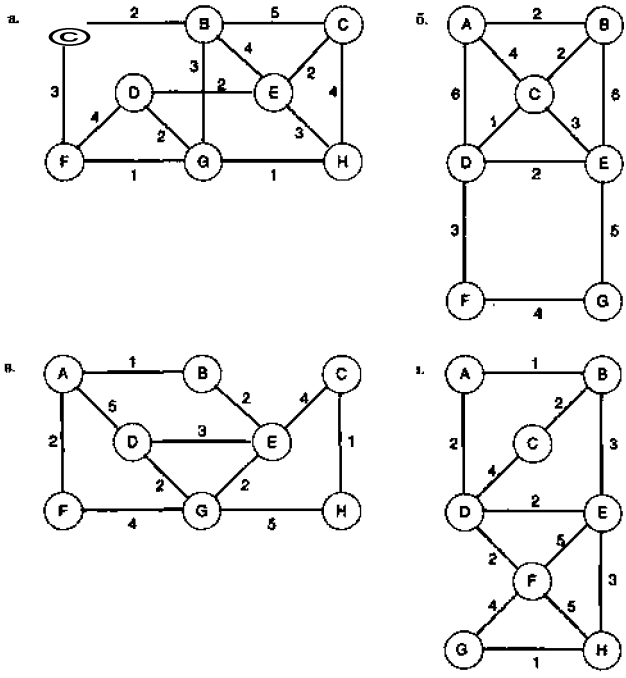
```
for k=1 to p do
   if v лежит в V[k] then
      P[k] помечает, что v занесена в дерево
   end if
   Pk меняет значения closest и distance для всех вершин
      еще не занесенных в дерево учитывая вновь
      добавленную в дерево вершину
   end for k
Parallel End
end for j
```

Анализ

Первый цикл алгоритма устанавливает начальные значения; его выполнение требует N/p шагов, поскольку количество начальных значений, устанавливаемых каждым процессором, пропорционально числу вершин, за которые он отвечает. Первый параллельный блок в главном цикле **for** всякий раз выполняет $N/p - 1$ сравнений: как показано в главе 1, ровно столько сравнений требует последовательный алгоритм поиска максимума или минимума. Следующий шаг состоит в том, чтобы выбрать минимальное расстояние из всех сообщенных p процессорами, на что уходит еще $p - 1$ сравнений. Шаг распределения значений в модели CREW занимает, как было показано, два цикла. В последнем параллельном блоке одно сравнение уходит на то, чтобы проверить, отвечает ли данный процессор за вновь добавленный узел, а исправление массивов closest и distance требует N/pопераций. Итак, при одном проходе главного цикла обработки выполняется $2(N/p)+p+1$ операций, а число проходов равно $N - 1$. Поэтому общее число операций в главном цикле $(N - 1) * (2 * (N/p) + p + 1)$, а общая сложность алгоритма равна $O(N^2/p)$при стоимости $p * O(N^2/p)$, или $O(N^2)$. Как мы уже видели в других случаях, оптимальность достигается при числе процессоров около $N/\log N$.

7.7.3. Упражнения

1) Выпишите взвешенную матрицу примыканий A для каждого из следующих четырех графов и подсчитайте матрицы A^2, A^4 и A^8.



2) Перепишите стандартный последовательный алгоритм умножения матриц, приспособив его для вычисления длины кратчайших путей в графе, как описано в конце раздела 7.7.1. Проверьте работу своего алгоритма на графе с рис. 7.7.

3) Проанализируйте подробно алгоритм подсчета длин кратчайших путей с использованием параллельного умножения матриц. Предполагайте, что матрицы умножаются за $O(N)$ операций при стоимости $O(N^3)$. Результаты анализа будут зависеть от того. сколько раз и на матрицах какого размера вызывается этот алгоритм.

4) Проследите выполнение тремя процессорами параллельного алгоритма построения минимального остовного дерева для графов из упражнения 1, начиная с вершины A. Трассировка должна показывать распределение вершин между процессорами, а также значение, возвращаемое каждым процессором на каждом проходе. Если Вы решили упражнение 1 из раздела 6.4.2, то сравните Ваши результаты для последовательного и параллельного алгоритмов.

Глава 8.

Недетерминированные алгоритмы

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- записывать алгоритмы и объяснять принципы их работы;
- описывать скорость и порядок роста функций.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- определять классы задач P, NP и NP-полный;
- объяснять разницу между задачами принятия решений и оптимизации;
- ставить классические NP задачи и объяснять их важность;
- объяснять причины, по которым задача относится к классу NP;
- объяснять, почему $P \subseteq NP$;
- объяснять, почему проблема «P=NP?» до сих пор остается открытой;
- записывать алгоритм проверки потенциального решения NP задачи.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Особое внимание уделите трассировке предложенных алгоритмов на предложенных входных данных и восстановлению приведенных «результатов». Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

До сих пор все рассматриваемые нами алгоритмы решали поставленные задачи за разумное время. Порядок сложности всех этих алгоритмов был полиномиален. Иногда время их работы оказывалось линейным, как при последовательном поиске: при удлинении списка вдвое алгоритм работает вдвое дольше. Нам встречались и алгоритмы сложности $O(N^2)$ — такую сложность имеют некоторые алгоритмы сортировки: если длину входного списка удвоить, то время работы алгоритма возрастает в 4 раза. Сложность стандартного алгоритма матричного умножения равна $O(N^3)$, и при увеличении размеров матриц вдвое такой алгоритм работает в 8 раз дольше. Хотя это и значительный рост, его все-таки можно контролировать. Соответствующие графики изображены на рис. 1.1 и 1.2.

В этой главе мы обратимся к задачам, сложность решения которых имеет порядок факториала $O(N!)$ или экспоненты $O(x^N)$ $(x > 2)$. Другими словами, для этих задач неизвестен алгоритм их решения за разумное время. Как мы увидим, единственный способ найти разумное или оптимальное решение такой задачи состоит в том, чтобы догадаться до ответа и проверить его правильность.

Несмотря на то, что решение таких задач требует длительного времени, мы не можем просто отмахнуться от них, поскольку у них есть важные приложения. Их необходимо уметь решать при выборе оптимальных путей продаж, составлении разумных расписаний экзаменов и распределении заданий. Эти задачи чрезвычайно важны, а эффективного способа поиска оптимальных путей их решения у нас нет, поэтому в следующей главе мы обратимся к задаче поиска приближенных решений. Рассматриваемые нами задачи образуют класс NP. Мы познакомимся с классическими задачами из этого класса. В нижеследующем параграфе мы обсуждаем, какие свойства задачи позволяют отнести ее к классу NP. В последнем параграфе мы приводим алгоритм проверки априорных решений.

8.1. Что такое NP?

Все алгоритмы, рассматривавшиеся нами в главах 1-7, имели полиномиальную сложность. Более того, сложность их всех была $O(N^3)$,[1] а самым времяемким был алгоритм комплексного умножения. Главное, однако, то, что мы могли найти точное решение этих задач за разумный промежуток времени. Все эти задачи относятся к классу P — классу задач полиномиальной сложности. Такие задачи называются также практически разрешимыми.

Есть и другой класс задач: они практически неразрешимы и мы не знаем алгоритмов, способных решить их за разумное время. Эти задачи образуют класс NP - - недетерминированной полиномиальной сложности. Значение этих слов прояснится к концу параграфа. Отметим только, что сложность всех известных детерминированных алгоритмов, решающих эти задачи, либо экспоненциальна, либо факториальна. Сложность некоторых из них равна 2^N, где N - - количество входных данных. В этом случае при добавлении к списку входных данных одного элемента время работы алгоритма удваивается. Если для решения такой задачи на входе из десяти элементов алгоритму требовалось 1024 операции, то на входе из 11 элементов число операций составит уже 2048. Это значительное возрастание времени при небольшом удлинении входа.

Словосочетание «недетерминированные полиномиальные», характеризующее задачи из класса NP, объясняется следующим двухшаговым подходом к их решению. На первом шаге имеется недетерминированный алгоритм, генерирующий возможное решение такой задачи — что-то вроде попытки угадать решение; иногда такая попытка оказывается успешной, и мы получаем оптимальный или близкий к оптимальному ответ, иногда безуспешной (ответ далек от оптимального). На втором шаге проверяется, действительно ли ответ, полученный на первом шаге, является решением исходной задачи. Каждый из этих шагов по отдельности требует полиномиального времени. Проблема, однако, в том, что мы не знаем, сколько раз нам придется повторить оба эти шага, чтобы получить искомое решение. Хотя оба шага и полиномиальны, число обращений к ним может оказаться экспоненциальным или факториальным.

К классу NP относится задача о коммивояжере. Нам задан набор городов и «стоимость» путешествия между любыми двумя из них. Нужно

[1] Напомним, что функция $д$ принадлежит $O(f)$, если она растет не быстрее, чем f. Поэтому, скажем, x^2 принадлежит классу $O(x^3)$.

определить такой порядок, в котором следует посетить все города (по одному разу) и вернуться в исходный город, чтобы общая стоимость путешествия оказалась минимальной. Эту задачу можно применять, например, для определения порядка эффективного сбора мусора из баков на улицах города или выбора кратчайшего пути распространения информации по всем узлам компьютерной сети. Восемь городов можно упорядочить 40 320 возможными способами, а для десяти городов это число возрастает уже до 3 628 800. Поиск кратчайшего пути требует перебора всех этих возможностей. Предположим, что у нас есть алгоритм, способный подсчитать стоимость путешествия через 15 городов в указанном порядке. Если за секунду такой алгоритм способен пропустить через себя 100 вариантов, то ему потребуется *больше четырех веков,* чтобы исследовать все возможности и найти кратчайший путь. Даже если в нашем распоряжении имеется 400 компьютеров, все равно у них уйдет на это год, а ведь мы имеем дело лишь с 15 городами. Для 20 городов *миллиард* компьютеров должен будет работать параллельно в течение девяти месяцев, чтобы найти кратчайший путь. Ясно, что быстрее и дешевле путешествовать хоть как-нибудь, чем ждать, пока компьютеры выдадут оптимальное решение.

Можно ли найти кратчайший путь, не просматривая их все? До сих пор никому не удалось придумать алгоритм, который не занимается, по существу, просмотром всех путей. Когда число городов невелико, задача решается быстро, однако это не означает, что так будет всегда, а нас как раз интересует решение общей задачи.

Задача о коммивояжере, конечно, очень похожа на задачи про графы, которыми мы занимались в главе 6. Каждый город можно представить вершиной графа, наличие пути между двумя городами — ребром, а стоимость путешествия между ними — весом этого ребра. Отсюда можно сделать вывод, что алгоритм поиска кратчайшего пути из § 6.5 решает и задачу коммивояжера, однако это не так. Какие два условия задачи о коммивояжере отличают ее от задачи о кратчайшем пути? Во-первых, мы должны посетить все города, а алгоритм поиска кратчайшего пути дает лишь путь между двумя заданными городами. Если собрать путь из кратчайших кусков, выдаваемых алгоритмом поиска кратчайших путей, то он будет проходить через некоторые города по нескольку раз. Второе отличие состоит в требовании возвращения в исходную точку, которое отсутствует в поиске кратчайшего пути.

Наше краткое обсуждение того, насколько велико число возможных упорядочиваний вершин, должно было убедить Вас в том, что детер-

минированный алгоритм, сравнивающий все возможные способы упорядочивания, работает чересчур долго. Чтобы показать, что эта задача относится к классу NP, нам необходимо понять, как ее можно решить посредством описанной выше двухшаговой процедуры. В задаче о коммивояжере на первом шаге случайным образом генерируется некоторое упорядочивание городов. Поскольку это недетерминированный процесс, каждый раз будет получаться новый порядок. Очевидно, что процесс генерации можно реализовать за полиномиальное время: мы можем хранить список городов, генерировать случайный номер, выбирать из списка город с этим именем и удалять его из списка, чтобы он не появился второй раз. Такая процедура выполняется за $O(N)$ операций, где N - - число городов. На втором шаге происходит подсчет стоимости путешествия по городам в указанном порядке. Для этого нам нужно просто просуммировать стоимости путешествия между последовательными парами городов в списке, что также требует $O(N)$ операций. Оба шага полиномиальны, поэтому задача о коммивояжере лежит в классе NP. Времяемкой делает ее именно необходимое число итераций этой процедуры.

Здесь следует отметить, что такую двухшаговую процедуру можно было применить к любой из рассматривавшихся нами ранее задач. Например, сортировку списка можно выполнять, генерируя произвольный порядок элементов исходного списка и проверяя, не является ли этот порядок возрастающим. Не относит ли это рассуждение задачу сортировки к классу NP? Конечно, относит. Разница между классом P и классом NP в том, что в первом случае у нас имеется детерминированный алгоритм, решающий задачу за полиномиальное время, а во втором мы такого алгоритма не знаем. Мы вернемся к этой проблеме вновь в § 8.3.

8.1.1. Сведение задачи к другой задаче

Один из способов решения задач состоит в том, чтобы свести, или редуцировать, одну задачу к другой. Тогда алгоритм решения второй задачи можно преобразовать таким образом, чтобы он решал первую. Если преобразование выполняется за полиномиальное время и вторая задача решается за полиномиальное время, то и наша новая задача также решается за полиномиальное время.

Поясним наше рассуждение примером. Пусть первая задача состоит в том, чтобы вернуть значение «да» в случае, если одна из данных булевских переменных имеет значение «истина», и вернуть «нет» в противоположном случае. Вторая задача заключается в том, чтобы найти максимальное зна-

чение в списке целых чисел. Каждая из них допускает простое ясное решение, но предположим на минуту, что мы знаем решение задачи о поиске максимума, а задачу про булевские переменные решать не умеем. Мы хотим свести задачу о булевских переменных к задаче о максимуме целых чисел. Напишем алгоритм преобразования набора значений булевских переменных в список целых чисел, который значению «ложь» сопоставляет число 0, а значению «истина» — число 1. Затем воспользуемся алгоритмом поиска максимального элемента в списке. По тому, как составлялся список, заключаем, что этот максимальный элемент может быть либо нулем, либо единицей. Такой ответ можно преобразовать в ответ в задаче о булевских переменных, возвращая «да», если максимальное значение равно 1, и «нет», если оно равно 0.

Мы видели в главе 1, что поиск максимального значения выполняется за линейное время, а редукция первой задачи ко второй тоже требует линейного времени, поэтому задачу о булевских переменных тоже можно решить за линейное время.

В следующем разделе мы воспользуемся техникой сведения, чтобы кое-что узнать о NP задачах. Однако редукция NP задач может оказаться гораздо более сложной.

8.1.2. NP-полные задачи

При обсуждении класса NP следует иметь в виду, что наше мнение, согласно которому их решение требует большого времени, основано на том, что мы просто не нашли эффективных алгоритмов их решения. Может быть, посмотрев на задачу коммивояжера с другой точки зрения, мы смогли бы разработать полиномиальный алгоритм ее решения. То же самое можно сказать и про другие задачи, которые мы будем изучать в следующем параграфе.

Термин NP-*полная* относится к самым сложным задачам в классе NP. Эти задачи выделены тем, что если нам все-таки удастся найти полиномиальный алгоритм решения какой-либо из них, то это будет означать, что все задачи класса NP допускают полиномиальные алгоритмы решения.

Мы показываем, что задача является NP-полной, указывая способ сведения к ней всех остальных задач класса NP. На практике эта деятельность выглядит не столь уж устрашающе -- нет необходимости осуществлять редукцию для каждой NP задачи. Вместо этого для того, чтобы доказать NP-полноту некоторой NP задачи A, достаточно свести к ней какую-нибудь NP-полную задачу B. Редуцировав задачу B к

задаче A, мы показываем, что и любая NP задача может быть сведена к A за два шага, первый из которых — ее редукция к B.

В предыдущем разделе мы выполняли редукцию полиномиального алгоритма. Посмотрим теперь на редукцию алгоритма, решающего NP задачу. Нам понадобится процедура, которая преобразует все составные части задачи в эквивалентные составные части другой задачи. Такое преобразование должно сохранять информацию: всякий раз, когда решение первой задачи дает положительный ответ, такой же ответ должен быть и во второй задаче, и наоборот.

Гамильтоновым путем в графе называется путь, проходящий через каждую вершину в точности один раз. Если при этом путь возвращается в исходную вершину, то он называется **гамильтоновым** циклом. Граф, в котором есть гамильтонов путь или цикл, не обязательно является полным. Задача о поиске гамильтонова цикла следующим образом сводится к задаче о коммивояжере. Каждая вершина графа — это город. Стоимость пути вдоль каждого ребра графа положим равной 1. Стоимость пути между двумя городами, не соединенными ребром, положим равной 2. А теперь решим соответствующую задачу о коммивояжере. Если в графе есть гамильтонов цикл, то алгоритм решения задачи о коммивояжере найдет циклический путь, состоящий из ребер веса 1. Если же гамильтонова цикла нет, то в найденном пути будет по крайней мере одно ребро веса 2. Если в графе N вершин, то в нем есть гамильтонов цикл, если длина найденного пути равна N, и такого цикла нет, если длина найденного пути больше N.

В 1971 году Кук доказал **NP-полноту** обсуждаемой в следующем параграфе задачи о конъюнктивной нормальной форме. **NP-полнота** большого числа задач была доказана путем редукции к ним задачи о конъюнктивной нормальной форме. В книге Гэри и Джонсона, опубликованной в 1979 году, приведены сотни задач, NP-полнота которых доказана.

Редукция — настолько мощная вещь, что если любую из **NP-полных** задач удастся свести к задаче класса P, то и все NP задачи получат полиномиальное решение. До сих пор ни одна из попыток построить такое сведе́ние не удалась.

8.2. Типичные NP задачи

Каждая из задач, которые мы будем обсуждать, является либо оптимизационной, либо задачей о принятии решения. Целью оптимизационной задачи обычно является конкретный результат, представляющий

собой минимальное или максимальное значение. В задаче о принятии решения обычно задается некоторое пограничное значение, и нас интересует, существует ли решение, большее (в задачах максимизации) или меньшее (в задачах минимизации) указанной границы. Ответом в задачах оптимизации служит полученный конкретный результат, а в задачах о принятии решений — «да» или «нет».

В § 8.1 мы занимались оптимизационным вариантом задачи о коммивояжере. Это задача минимизации, и нас интересовал путь минимальной стоимости. В варианте принятия решения мы могли бы спросить, существует ли путь коммивояжера со стоимостью, меньшей заданной константы C. Ясно, что ответ в задаче о принятии решения зависит от выбранной границы. Если эта граница очень велика (например, она превышает суммарную стоимость всех дорог), то ответ «да» получить несложно. Если эта граница чересчур мала (например, она меньше стоимости дороги между любыми двумя городами), то ответ «нет» также дается легко. В остальных промежуточных случаях время поиска ответа очень велико и сравнимо со временем решения оптимизационной задачи. Поэтому мы будем говорить вперемешку о задачах оптимизации и принятия решений, используя ту из них, которая точнее отвечает нашим текущим целям.

В следующих нескольких разделах мы опишем еще шесть NP задач — как в оптимизационном варианте, так и в варианте принятия решения.

8.2.1. Раскраска графа

Как мы уже говорили в главе 6, граф $G = (V, E)$ представляет собой набор вершин, или узлов, V и набор ребер E, соединяющих вершины попарно. Здесь мы будем заниматься только неориентированными графами. Вершины графа можно раскрасить в разные цвета, которые обычно обозначаются целыми числами. Нас интересуют такие раскраски, в которых концы каждого ребра окрашены разными цветами. Очевидно, что в графе с N вершинами можно покрасить вершины в N различных цветов, но можно ли обойтись меньшим количеством цветов? В задаче оптимизации нас интересует минимальное число цветов, необходимых для раскраски вершин графа. В задаче принятия решения нас интересует, можно ли раскрасить вершины в C или менее цветов.

У задачи о раскраске графа есть практические приложения. Если каждая вершина графа обозначает читаемый в колледже курс, и вершины соединяются ребром, если есть студент, слушающий оба курса,

то получается весьма сложный граф. Если предположить, что каждый студент слушает 5 курсов, то на студента приходится 10 ребер. Предположим, что на 3500 студентов приходится 500 курсов. Тогда у получившегося графа будет 500 вершин и 35 000 ребер. Если на экзамены отведено 20 дней, то это означает, что вершины графа нужно раскрасить в 20 цветов, чтобы ни у одного студента не приходилось по два экзамена *в* день.

Разработка бесконфликтного расписания экзаменов эквивалентна раскраске графов. Однако задача раскраски графов принадлежит к классу NP, поэтому разработка бесконфликтного расписания за разумное время невозможна. Кроме того при планировании экзаменов обычно требуется, чтобы у студента было не больше двух экзаменов в день, а экзамены по различным частям курсам назначаются в один день. Очевидно, что разработка «совершенного» плана экзаменов невозможна, и поэтому необходима другая техника для получения по крайней мере неплохих планов. Приближенные алгоритмы обсуждаются в главе 9.

8.2.2. Раскладка по ящикам

Пусть у нас есть несколько ящиков единичной емкости и набор объектов различных размеров $s_1, s_2, \ldots, s_N$. В задаче оптимизации нас интересует наименьшее количество ящиков, необходимое для раскладки всех объектов, а в задаче принятия решения — можно ли упаковать все объекты в B или менее ящиков.

Эта задача возникает при записи информации на диске или во фрагментированной памяти компьютера, при эффективном распределении груза на кораблях, при вырезании кусков из стандартных порций материала по заказам клиентов. Если, например, у нас есть большие металлические листы и список заказов на меньшие листы, то естественно мы хотим распределить заказы как можно плотнее, уменьшив тем самым потери и увеличив доход.

8.2.3. Упаковка рюкзака

У нас имеется набор объектов объемом $s_1, \ldots, s_N$ стоимости $w_1, \ldots, w_N$. В задаче оптимизации мы хотим упаковать рюкзак объемом K так, чтобы его стоимость была максимальной. В задаче принятия решения нас интересует, можно ли добиться, чтобы суммарная стоимость упакованных объектов была по меньшей мере W.

Эта задача возникает при выборе стратегии вложения денег: объемом здесь является объем различных вложений, стоимостью — пред-

полагаемая величина дохода, а объем рюкзака определяется размером планируемых капиталовложений.

8.2.4. Задача о суммах элементов подмножеств

Пусть у нас есть множество объектов различных размеров $s_1, \ldots, s_N$ и некоторая положительная верхняя граница L. В задаче оптимизации нам необходимо найти набор объектов, сумма размеров которых наиболее близка к L и не превышает этой верхней границы. В задаче принятия решения нужно установить, существует ли набор объектов с суммой размеров L. Это упрощенная версия задачи об упаковке рюкзака.

8.2.5. Задача об истинности КНФ-выражения

Конъюнктивная нормальная форма (КНФ) представляет собой последовательность булевских выражений, связанных между собой операторами AND (обозначаемыми Л), причем каждое выражение является мономом от булевских переменных или их отрицаний, связанных операторами OR (которые обозначаются через V). Вот пример булевского выражения в конъюнктивной нормальной форме (отрицание обозначается чертой над именем переменной):

$$(\text{a V } b) \text{ Л } (\text{a V c}) \text{ Л } (\text{a V 6 V c V } d) \text{ Л } (6 \text{ V c V } \bar{d}) \text{ Л } (\text{a V } b \text{ V c V } \bar{d} \vee \text{e}).$$

Задача об истинности булевского выражения в конъюнктивной нормальной форме ставится только в варианте принятия решения: существуют ли у переменных, входящих в выражение, такие значения истинности, подстановка которых делает все выражение истинным. Как число переменных, так и сложность выражения не ограничены, поэтому число комбинаций значений истинности может быть очень велико.

8.2.6. Задача планирования работ

Пусть у нас есть набор работ, и мы знаем время, необходимое для завершения каждой из них, $t_1, t_2, \ldots, t_N$, сроки $d_1, d_2, \ldots, d_N$, к которым эти работы должны быть обязательно завершены, а также штрафы $p_1, p_2, \cdots, p_N$, которые будут наложены при незавершении каждой работы в установленные сроки. Задача оптимизации требует установить порядок работ, минимизирующий накладываемые штрафы. В задаче принятия решений мы спрашиваем, есть ли порядок работ, при котором величина штрафа будет не больше P.

8.2.7. Упражнения

1) В § 8.1 был предложен двухэтапный недетерминированный процесс решения задач из класса NP. Опишите этот процесс для нижеследующих задач. В ответ должен входить формат выхода недетерминированного шага, причем он должен включать в себя все элементы решения задачи. Например, в задаче о коммивояжере выходные данные должны представлять собой список всех городов в порядке их посещения. Кроме того, Вы должны описать процесс проверки того, что предложенный вариант действительно является решением задачи.

   а) Задача о раскраске вершин графа

   б) Задача о раскладке по ящикам

   в) Задача об упаковке рюкзака

   г) Задача о суммах элементов подмножеств

   д) Задача об истинности КНФ-выражения

   е) Задача планирования работ

2) В § 8.1 был описан процесс преобразования одной задачи в другую, позволяющий доказывать NP-полноту задач. Все задачи из § 8.1 и § 8.2 являются NP-полными, поэтому каждая из них может быть сведена к любой другой. Опишите соответствующее преобразование первой задачи во вторую для каждой из нижеследующих пар задач.

   а) Упаковка рюкзака, раскладка по ящикам

   б) Раскладка по ящикам, планирование работ

   в) Планирование работ, суммы элементов подмножеств

   г) Суммы элементов подмножеств, коммивояжер

   д) Коммивояжер, планирование работ

8.3. Какие задачи относятся к классу NP?

Мы разобрали много задач из класса P и несколько задач из класса NP. Задача принадлежит классу NP, если она разрешима за полиномиальное время недетерминированным алгоритмом. Как упоминалось, процесс сортировки можно реализовать следующим образом:

1) вывести элементы списка в случайном порядке;

2) проверить, что $s_i < s_{i+1}$ для всех i от 1 до $N - 1$.

Это недетерминированный **двухэтапный** процесс. Первый этап не требует сравнений, и его можно выполнить за N шагов — по одному шагу на выходной элемент списка. Второй этап также полиномиален: для его выполнения необходимо сделать $N - 1$ сравнений. Такая процедура подходит под наше определение класса NP, и можно прийти к выводу, что задача сортировки принадлежит как классу P, так и классу NP. То же самое можно проделать с любым полиномиальным алгоритмом, поэтому все задачи класса P лежат и в классе NP, т.е. P является подмножеством в NP. Однако в классе NP есть задачи, для которых мы не знаем полиномиального детерминированного алгоритма их решения.

Сутью этого различия является большое число вариантов, которые необходимо исследовать в NP задачах. Однако это число лишь незначительно превышает число комбинаций входных значений. У нас может быть список из 30 различных элементов или городов, и только один из 30! возможных порядков в этом списке является возрастающим или задает кратчайший путь. Разница же в том, что упорядочить список можно и полиномиальным алгоритмом — некоторым из них для этого понадобится всего 150 сравнений. В алгоритме пузырьковой сортировки при первом проходе по списку по крайней мере наибольший элемент встанет на свое место, отбросив тем самым всего за 29 сравнений по крайней мере 1/30 всех возможных комбинаций. На втором проходе 28 сравнений отбрасывают 1/29 часть оставшихся возможностей. При более внимательном взгляде видно, что число отброшенных возможностей может быть и большим: результатом каждого прохода может быть не только помещение на место наибольшего элемента списка, но и переупорядочивание других элементов.

Но наилучший известный нам способ поиска кратчайшего пути состоит в переборе всех возможных вариантов путей и сравнении их длин. У нас нет алгоритма, позволяющего успешно отбросить значительное количество вариантов[2]. Вот нам и приходится просматривать их все. Даже при просмотре 1 000 000 000 из 30! путей в секунду нам понадобилось бы более 840 миллиардов веков для проверки всех путей. В главе 9

[2]Казалось бы, можно выкинуть дорогостоящую дорогу между двумя городами. Но даже это простое предложение не проходит: не исключено, что выкинутую дорогу придется заменить двумя, суммарная стоимость которых выше, и никакого улучшения мы не получим. Проверка же того, можем ли мы себе позволить выкинуть ребро в графе, возвращает нас к сложному алгоритму.

мы обсудим способы получения ответов, близких к оптимальному. При этом мы не можем оценить, насколько близок оптимальному полученный ответ, поскольку оптимальное значение нам неизвестно. Вместо этого мы можем продолжать исполнять алгоритм поиска приближенного решения: всякое новое решение будет лучше предыдущего. Может быть так мы доберемся и до оптимального решения, но гарантии этого отсутствуют.

Значит задача попадает в класс NP, если для ее решения необходимо рассмотреть огромное количество возможностей и у нас нет эффективного детерминированного алгоритма просеивания этих возможностей в поисках оптимального ответа.

8.3.1. Выполнено ли равенство P=NP?

Казалось бы, предыдущее обсуждение делает смешной саму постановку вопроса, совпадает ли класс P с классом NP. Пример с задачей сортировки, которая решается как полиномиальным детерминированным, так и полиномиальным недетерминированным алгоритмом, подтверждает, что класс P является подклассом в NP. Обсуждение различия между сортировкой и задачей коммивояжера должно, казалось бы, убедить Вас в том, что нам известны задачи из класса NP, не входящие в класс P. Однако это не так. Нам известно только то, что пока не удалось найти детерминированного полиномиального алгоритма для всех задач из класса NP. Это не означает, что такого алгоритма не существует, и исследователи по-прежнему бьются над проблемой совпадения классов. Многие верят, что полиномиальные алгоритмы решения NP-полных задач не существуют, но как можно доказать, что не существует полиномиального алгоритма решения той или иной задачи? Лучше всего исследовать эту задачу и попробовать оценить снизу минимальный объем работы, необходимый для ее решения. Здесь, однако, мы и наталкиваемся на проблему получения нижней оценки, превышающей любой многочлен. Вопрос, выполняется ли равенство P=NP, до сих пор остается предметом исследования по всему миру.

8.3.2. Упражнения

1) Укажите для каждой из следующих задач, лежит ли она в классе P и в классе NP. Для задач, отнесенных Вами к классу P, дайте набросок полиномиального алгоритма их решения. Для остальных

задач объясните, почему Вы считаете, что у них нет полиномиального детерминированного решения.

а) Значением $S(k)$ функции Смарандаша является такое наименьшее число т, что т! делится на k. Например, $5(9) = 6$, так как число $6! = 720$ делится на 9, а никакой меньший факториал на 9 не делится. Вычислите функцию Смарандаша произвольного аргумента.

б) Вам нужно рассадить в аудитории N детей, заняв как можно меньше рядов. Для каждого ребенка имеется список ребят, которых он не любит. Предыдущий опыт показывает, что если ребенок не любит кого-то из своих одноклассников, то оказавшись на одном ряду или на соседних рядах с ним, он будет бросаться в него разными предметами. Такую ситуацию следует предотвратить. Подсчитайте минимальное необходимое количество рядов при наличии таких списков или докажите, что детей нельзя усадить с соблюдением поставленных условий.

в) Фунция Аккермана задается равенствами

$$\begin{aligned} A(0,y) &= y+1, \\ A(x+1,0) &= A(x,1), \\ A(x+1,y+1) &= A(x,A(x+1,y)). \end{aligned}$$

Вычислите функцию Аккермана.

г) Вы находитесь в темноте перед стеной, простирающейся бесконечно в обоих направлениях. Вы знаете, что в стене есть дверь, но Вам неизвестно, слева или справа она от Вас. Имеющийся у Вас маленький фонарик позволяет Вам обнаружить дверь только, если Вы оказываетесь на расстоянии одного шага от нее. Найдите дверь.

8.4. Проверка возможных решений

Описание класса NP предполагает, что у задач этого класса есть решение с недетерминированным первым шагом, на котором генерируется возможный ответ, и детерминированным вторым шагом, который сгенерированный ответ проверяет. Оба эти шага выполняются за полиномиальное время. В этом параграфе мы займемся алгоритмами,

проверяющими полученный ответ в задачах о планировании работ и о раскрашивании графа.

8.4.1. Задача о планировании работ

Напомним, что в задаче о планировании работ задан набор работ, которые необходимо выполнить. Для каждой работы известно, сколько времени необходимо на ее выполнение, срок, к которому ее следует завершить, и штраф, накладываемый в случае ее незавершения к назначенному сроку. Работы выполняются последовательно, а сроки окончания отсчитываются с момента начала первой работы. Для каждой работы нам известна четверка (n, t, d, p), где n — номер работы, t — занимаемое ею время, d — срок окончания, p — штраф. Вот пример списка из пяти работ: $\{(1,3,5,2),(2,5,7,4),(3,1,5,3),(4,6,9,1),(5,2,7,4)\}$.

В задаче принятия решения задается некоторое значение P, и мы хотим узнать, существует ли такой порядок работ, при котором суммарный штраф не превысит P. В задаче оптимизации нас интересует минимальное значение суммарного штрафа. Мы займемся задачей принятия решений: если несколько раз вызвать алгоритм решения этой задачи с возрастающей границей P, пока мы не получим утвердительный ответ, то мы решим заодно и задачу оптимизации. Другими словами, мы спрашиваем, существует ли порядок работ с штрафом 0. Если такого порядка нет, то мы переходим к штрафу 1, и будем увеличивать размер штрафа до тех пор, пока не получим утвердительного ответа. Следующий алгоритм сравнивает с порогом возможное решение задачи планирования работ.

```
PenaltyLess(list,N,limit)
list  упорядоченный список работ
N     общее число работ
limit предельная величина штрафа

currentTime=0
currentPenalty=0
currentJob=1
while (currentJob<=N) and (currentPenalty<=limit) do
   currentTime=currentTime+list[currentJob.time
   if (list[currentJob].deadline<currentTime( then
      currentPenalty=currentPenalty+list[currentJob] .penalty
   end if
```

```
    current Job=current Job+1
end while

if currentPenalty<=limit then
    return да
else
    return нет
end if
```

Принадлежность задачи классу NP требует, чтобы мы могли проверить предложенное решение за полиномиальное время. Видно, что описанный алгоритм действительно подсчитывает суммарный штраф. Анализ временной сложности показывает, что цикл while совершает не более N проходов; максимум достигается, если значение переменной currentPenalty не превышает предельное. Учет всех операций позволяет заключить, что общее число сравнений равно $3N + 1$, а число сложений равно $3N$. Это означает, что сложность алгоритма равна $O(N)$, и значит он полиномиален и удовлетворяет определению класса NP.

8.4.2. Раскраска графа

В задаче о раскраске графа требуется найти способ раскраски вершин графа в несколько цветов (занумерованных целыми числами) таким образом, чтобы любые две соседние вершины были покрашены в различные цвета. Принятие решения заключается в том, чтобы установить, можно ли покрасить вершины в C или меньше цветов с соблюдением указанного требования, а оптимизация — в том, чтобы найти минимально необходимое число цветов.

На недетерминированном этапе генерируется возможное решение, представляющее собой список вершин и приписанных к ним цветов. Именно на этом этапе решается, сколько будет использовано цветов, поэтому этап проверки не несет ответственности за выбранное число цветов. При выборе решения недетерминированный этап попробует обойтись C цветами. В задаче оптимизации он может начать с большого числа цветов, последовательно уменьшая их количество, пока требуемая раскраска еще возможна. Обнаружив, что граф нельзя раскрасить в X цветов, мы заключим, что раскраска в $X + 1$ цвет является минимально возможной.

Нижеследующий алгоритм проверяет, является ли сгенерированная раскраска допустимой. Раскрашиваемый граф представлен в виде списка примыканий, элемент `graph[j]` этого списка описывает j-ую верши-

ну графа, поле `graph[j] . edgeCount` этого элемента — число выходящих из нее ребер, а поле `graph[j] .edge` является массивом, в котором хранятся вершины, соседние с вершиной j.

```
boolean ValidColoring(graph,N,colors)
graph  список примыканий
N      число вершин в графе
colors массив, сопоставляющий каждой вершине ее цвет

for j=1 to N do
   for k=1 to graph[j].edgeCount do
      if (colors[j]=colors[graph[j].edge[k]]) then
         return нет
      end if
   end for k
end for j
return да
```

Видно, что этот алгоритм правильно проверяет допустимость раскраски. Он проходит по всем вершинам, и если вершина непосредственно связана с другой того же цвета, то алгоритм останавливается и возвращает отрицательный ответ. Если же все пары соседних вершин окрашены различными цветами, то ответ утвердительный. Что касается временной сложности этого алгоритма, то внешний цикл `for` осуществляет N проходов. Во внутреннем цикле просматривается каждое ребро, выходящее из данной вершины. Поэтому общее число сравнений будет равно числу ребер, т.е. сложность алгоритма равна $O(\texttt{edges})$ — очевидно полиномиальная оценка, поскольку число ребер в графе не превосходит N^2. Поэтому наши требования оказываются выполненными.

8.4.3. Упражнения

1) Разработайте алгоритмы для проверки сгенерированных ответов при выборе решений в нижеследующих задачах.

   а) Задача раскладки по ящикам

   б) Задача коммивояжера

   в) Задача упаковки рюкзака

   г) Задача о суммах элементов подмножеств

   д) Задача об истинности конъюнктивной нормальной формы

Глава 9.
Другие алгоритмические инструменты

НЕОБХОДИМЫЕ ПРЕДВАРИТЕЛЬНЫЕ ЗНАНИЯ

Приступая к чтению этой главы, Вы должны уметь

- записывать алгоритмы и объяснять принципы их работы;
- описывать класс NP;
- определять скорость и порядок роста функций;
- пользоваться генераторами и таблицами случайных чисел (приложения А и Б);
- пользоваться рекурсивными алгоритмами.

ЦЕЛИ

Освоив эту главу, Вы должны уметь

- объяснять понятие приближенного алгоритма;
- объяснять приближенные алгоритмы для некоторого класса NP задач;
- описывать четыре типа вероятностных алгоритмов;
- повышать производительность алгоритмов с помощью массивов.

СОВЕТЫ ПО ИЗУЧЕНИЮ

Изучая эту главу, самостоятельно проработайте все примеры и убедитесь, что Вы их поняли. Особое внимание уделите трассировке алгоритмов на предложенных входных данных и постарайтесь восстановить описанные результаты. Кроме того, попробуйте отвечать на предложенные вопросы самостоятельно прежде, чем читать ответы на них или подсказки.

* * *

Как обсуждалось в главе 8, задачи из класса NP имеют большое число приложений, поэтому их решение представляет значительный интерес. Поскольку полиномиальные алгоритмы решения этих задач неизвестны, следует рассмотреть альтернативные возможности, дающие лишь достаточно хорошие ответы. Такой алгоритм может иногда дать и оптимальный ответ, однако такую счастливую случайность нельзя предусмотреть. Мы изучим некоторое количество приближенных алгоритмов решения задач из главы 8.

В основе вероятностных алгоритмов лежит идея о том, что иногда лучше попробовать угадать ответ, чем вычислить его. Вероятностные алгоритмы делятся на четыре класса — численные, Монте Карло, Лас Вегас и Шервуд — хотя зачастую все вероятностные алгоритмы называются методами Монте Карло. Общим свойством всех этих категорий является то, что чем дольше алгоритм работает, тем лучший результат он выдает.

В конце этой главы мы показываем, как можно с помощью динамического программирования повысить эффективность рекурсивных алгоритмов и алгоритмов умножения матриц за счет выбора порядка, в котором они умножаются.

9.1. Жадные приближенные алгоритмы

В главе 6 мы описали два жадных алгоритма поиска минимального остовного дерева в графе, а также жадный алгоритм поиска кратчайшего пути между двумя вершинами графа. В этом параграфе мы изучим несколько жадных алгоритмов, приближенно решающих задачи из класса NP.

Как уже обсуждалось, найти точное решение задачи из класса NP трудно, потому что число требующих проверки возможных комбина-

ций входных значений чрезвычайно велико. Для каждого набора входных значений I мы можем создать множество возможных решений PS_I. Оптимальное решение это такое решение $S_{\text{optimal}} \in \mathrm{PS}_I$, что $\mathrm{Value}(S_{\text{optimal}}) < \mathrm{Value}(S')$ для всех $S' \in \mathrm{PS}_I$, если мы имеем дело с задачей минимизации, и $\mathrm{Value}(S_{\text{optimal}}) > \mathrm{Value}(S')$ для всех $S' \in \mathrm{PS}_I$, если мы имеем дело с задачей максимизации.

Решения, предлагаемые приближенными алгоритмами для задач класса NP, не будут оптимальными, поскольку алгоритмы просматривают только часть множества PS_I, зачастую очень маленькую. Качество приближенного алгоритма можно определить, сравнив полученное решение с оптимальным. Иногда оптимальное значение — например, минимальную длину пути коммивояжера — можно узнать, и не зная оптимального решения — самого пути. Качество приближенного алгоритма при этом дается дробью

$$Q_A(I) = \begin{cases} \dfrac{\mathrm{Value}(A(I))}{\mathrm{Value}(S_{\text{optimal}})}, & \text{в задачах минимизации;} \\ \dfrac{\mathrm{Value}(S_{\text{optimal}})}{\mathrm{Value}(A(I))}, & \text{в задачах максимизации.} \end{cases}$$

Иногда будет играть роль, рассматриваем ли мы случаи с фиксированным числом входных значений или случаи с общим оптимальным решением. Другими словами, хотим ли мы понять, насколько хорош приближенный алгоритм на входных списках длины 10 или на входных списках различной длины с оптимальным значением 50? Эти две точки зрения могут привести к различным результатам.

В нижеследующих разделах мы изучаем несколько приближенных алгоритмов решения рассмотренных нами ранее задач. Приводимые нами алгоритмы являются не единственно возможными, скорее они призваны продемонстрировать многообразие различных подходов. Все эти приближенные алгоритмы полиномиальны по времени.

9.1.1. Приближения в задаче о коммивояжере

Для решения многих задач (в том числе и из класса P) пригодны так называемые жадные алгоритмы. Эти алгоритмы пытаются сделать наилучший выбор на основе доступной информации. Напомним, что алгоритмы построения минимального остовного дерева и кратчайшего пути являются примерами жадных алгоритмов.

Сначала кажется, что для решения задачи о коммивояжере можно просто воспользоваться алгоритмом поиска кратчайшего пути, однако ситуация не так проста. Алгоритм Дейкстры предназначен для поиска кратчайшего пути между двумя вершинами, но найденный путь не обязательно проходит через все вершины графа. Однако можно воспользоваться этим общим подходом для построения жадного приближенного алгоритма. Стоимость проезда между городами можно задать матрицей примыканий; пример такой матрицы приведен на рис. 9.1[1].

Из/До	2	3	4	**5**	**6**	**7**
1	16	12	13	6	7	11
2		21	18	8	19	5
3			20	1	3	15
4				**14**	10	4
5					2	17
6						9

Рис. 9.1. Матрица примыканий для полного взвешенного графа

Наш алгоритм будет перебирать набор ребер в порядке возрастания всех весов. Он не будет формировать путь; вместо этого он будет проверять, что добавляемые к пути ребра удовлетворяют двум условиям:

1) При добавлении ребра не образуется цикл, если это не завершающее ребро в пути.

2) Добавленное ребро не является третьим, выходящим из какой-либо одной вершины.

В примере с рис. 9.1 мы выберем первым ребро (3,5), поскольку у него наименьший вес. Следующим выбранным ребром будет (5,6). Затем алгоритм рассмотрит ребро (3,6), однако оно будет отвергнуто, поскольку вместе с двумя уже выбранными ребрами образует цикл [3, 5, 6, 3], не проходящий через все вершины. Следующие два ребра (4,7) и (2,7) будут добавлены к циклу. Затем будет рассмотрено ребро (1,5), но оно будет отвергнуто, поскольку это третье ребро, выходящее из

[1] Эта матрица верхняя треугольная, поскольку стоимость проезда между городами i и j одинакова в обоих направлениях. Если бы мы выписали все стоимости, то нижний треугольник матрицы попросту совпал бы с верхним. Использование верхнетреугольной матрицы позволяет упростить трассировку алгоритма.

вершины 5. Затем будут добавлены ребра (1,6) и (1,4). Последним добавленным ребром будет (2,3). В результате мы получим циклический путь [1, 4, 7, 2, 3, 5, 6, 1], полная длина которого 53. Это неплохое приближение, однако заведомо не оптимальное решение: есть по крайней мере один более короткий путь, [1, 4, 7, 2, 5, 3, 6, 1], полная длина которого 41.

9.1.2. Приближения в задаче о раскладке по ящикам

Один из подходов к получению приближенного решения задачи о раскладке по ящикам предлагает упаковывать первый подходящий предмет. Стратегия состоит в том, что ящики просматриваются поочередно пока не найдется ящик, в котором достаточно свободного места для упаковки очередного предмета. Как, например, будет происходить упаковка предметов размером (0.5, 0.7, 0.3, 0.9, 0.6, 0.8, 0.1, 0.4, 0.2, 0.5)? Видно, что в первый ящик попадут предметы размером [0.5, 0.3, 0.1], во второй — предметы размером [0.7, 0.2], в третий — предмет размером [0.9], в четвертый — предметы размером [0.6, 0.4], в пятый — [0.8], и, наконец, в шестой — предмет размером [0.5]. Эта упаковка не оптимальна, потому что возможна упаковка в пять ящиков: [0.9, 0.1], [0.8, 0.2], [0.7, 0.3], [0.6, 0.4] и [0.5,0.5]. Вот алгоритм укладки в первый подходящий ящик.

```
FirstFit(size,N,bin)
size  список размеров предметов
N     число предметов
bin   положения разложенных предметов

for i=1 to N do
   binUsed[i]=0
end do
for item=1 to N do
   binLoc=1
   while used[binLoc]+size[item]>1 do
      binLoc=binLoc+1
   end while
   bin[item]=binLoc
   used[binLoc]=used[binLoc]+size[item]
end for
```

В другом варианте этого алгоритма список предметов сначала сортируется по убыванию размера, а затем они раскладываются поочередно в первый подходящий ящик[2]. Читатель без труда проверит, что в рассмотренном примере модифицированный алгоритм приведет к оптимальному решению. Однако модифицированный алгоритм не всегда приводит к лучшим результатам, чем обычный. Рассмотрим набор предметов размеров (0.2, 0.6, 0.5, 0.2, 0.8, 0.3, 0.2, 0.2). Обычная раскладка в первый подходящий ящик приводит к оптимальной раскладке в три ящика. Начав с сортировки списка, мы получим список (0.8, 0.6, 0.5, 0.3, 0.2, 0.2, 0.2, 0.2). Раскладка в первый подходящий ящик отсортированного списка приведет к раскладке по ящикам [0.8, 0.2], [0.6, 0.3], [0.5, 0.2, 0.2] и [0.2], использующей на один ящик больше оптимального решения.

Анализ показывает, что стратегия укладки в первый подходящий ящик по отсортированному списку приводит к числу ящиков, превышающему оптимальное в среднем на 50%. Это означает, что если, скажем, для оптимальной укладки достаточно 10 ящиков, то результат алгоритма будет около 15. Если список предварительно не отсортирован, то дополнительные расходы составят в среднем 70%, т.е. при оптимальном числе ящиков 10 алгоритм сгенерирует укладку в 17 ящиков.

9.1.3. Приближения в задаче об упаковке рюкзака

Приближенный алгоритм упаковки рюкзака представляет собой простой жадный алгоритм, выбирающий наилучшее отношение стоимости к размеру. Сначала мы сортируем список объектов по отношению стоимости к размеру. Каждый объект представляется в виде пары [размер, стоимость]. Для списка объектов ([25, 50], [20, 80], [20, 50], [15, 45], [30, 105], [35, 35], [20, 10], [10, 45]) удельные стоимости имеют значения (2, 4, 2.5, 3, 3.5, 1, 0.5, 4.5). Сортировка удельных стоимостей приводит к списку ([10, 45], [20, 80], [30, 105], [15, 45], [20, 50], [25,50], [35, 35], [20, 10]). После этого мы начинаем заполнять рюкзак последовательно идущими элементами отсортированного списка объектов. Если очередной объект не входит — мы отбрасываем его и переходим к следующему; так продолжается до тех пор пока рюкзак не заполнится или список объектов не будет исчерпан. Таким образом, если емкость рюкзака 80, то мы сможем поместить в него первые четыре предмета суммарного объема 75 и общей стоимостью 275. Это не оптимальное решение:

[2]На самом деле, порядок в отсортированном списке предметов является неубывающим: мы не требуем, чтобы размеры предметов были различными.

первые три предмета с добавкой пятого дают суммарный объем 80 и стоимость 280.

9.1.4. Приближения в задаче о сумме элементов подмножества

Если в задаче об упаковке рюкзака положить стоимость каждого предмета равной его объему, то она превратится в задачу о сумме элементов подмножества. Это означает, что предложенный для решения первой задачи жадный алгоритм применим и здесь. Удельная стоимость каждого предмета равна 1, поэтому на этапе сортировки предметы упорядочиваются по убыванию их размеров.

Для приближенного решения задачи о сумме элементов подмножества имеется и другой алгоритм, который также не лишен жадности. Этот алгоритм выдает тем лучший результат, чем дольше он работает, и если его можно прогнать $N + 1$ раз, то выданный им результат будет оптимальным. Причина этого в том, что на каждом проходе он вовлекает в рассмотрение новые случаи. На первом проходе алгоритм начинает с пустого множества и добавляет в него элементы в убывающем порядке до тех пор, пока не будет достигнут предел или не будут опробованы все элементы входного списка. На втором проходе алгоритм начинает со всевозможных одноэлементных подмножеств и добавляет к ним элементы списка. На третьем проходе изучаются все двухэлементные подмножества. Число проходов ограничено только временем, отведенным на работу алгоритма. Если при списке из 10 элементов удается выполнить 11 проходов, то оптимальное решение будет найдено. Для списка из 10 элементов на первом проходе имеется единственное пустое подмножество, на втором проходе есть 10 одноэлементных подмножеств, на третьем имеется 45 двухэлементных подмножеств. Больше всего начальных подмножеств встречается на шестом проходе — это 252 пятиэлементных подмножества. Поэтому хотя сам процесс и выглядит простым, он все равно требует значительного времени.

Вот как выглядит алгоритм, реализующий один проход:

```
SubsetSum(sizes,N,limit,pass,result,sum)
sizes  список размеров элементов
N      число элементов в списке
limit  предельная сумма элементов подмножества
pass   число элементов в начальном множестве
result список элементов в наилучшем из найденных решений
sum    сумма элементов в найденном решении
```

Число элементов в **начальном** множестве	Размер подмножеств	Добавляемые элементы	Сумма элементов в найденном решении
0	o	27,22,1	50
1	27	22,1	50
	14	27,11,1	53
	11	27,14,1	53
	7	27,14,1	49
	1	27,22	50
2	27,22	1	50
	27,14	11,1	53
	27,11	14,1	53
	27,7	14,1	49
	27,1	22	50
	22,14	11,7,1	55
	22,11	14,7,1	**55**
	22,7	14,11,1	55
	22,1	27	50
	14,11	27,1	53
	14,7	27,1	49
	14,1	27,11	53
	11,7	27,1	46
	11,1	27,14	53
	7,1	27,14	49

Рис. 9.2. Результаты первых трех проходов приближенного алгоритма поиска максимальной суммы элементов подмножества

```
sum=0
for каждого подмножества T в {1,...,n} do
   tempSum=0
   for i=1 to N do
      if i лежит в T then
         tempSum=tempSum+sizes [i]
      end if
   end for
   if tempSum<=limit then
      for j=1 to N do
         if i не лежит в T and tempSum+sizes[j]<=limit then
            tempSum=tempSum+sizes[j]
```

```
            T=T+{j}
         end if
      end for
   end if
   if sum<tempSum then
      sum=tempSum
      result=T
   end if
end for
```

Например, на списке входных значений {27, 22, 14, 11, 7, 1} и при верхнем пределе суммы элементов подмножества равном 55 последовательность проходов процесса изображена на рис. 9.2. Видно, что оптимальное решение 55 найдено на третьем проходе.

9.1.5. Приближения в задаче о раскраске графа

Раскраска графа — необычная задача: как мы уже упоминали ранее, построение раскраски, достаточно близкой к оптимальной, дается столь же сложно, как и построение оптимальной раскраски. Число красок, даваемое наилучшим известным полиномиальным алгоритмом, более чем вдвое превышает оптимальное. Кроме того доказано, что если существует полиномиальный алгоритм, раскрашивающий вершины любого графа числом красок, превышающим оптимальное не более чем вдвое, то существует и полиномиальный алгоритм оптимальной раскраски любого графа. Существование такого алгоритма означало бы, что P=NP. На сложность графа можно наложить некоторые условия, облегчающие его раскраску. Например, известны полиномиальные алгоритмы раскраски планарных графов, т.е. таких графов, которые можно изобразить на плоскости в виде набора вершин и ребер, представленных попарно не пересекающимися дугами.

Вот простой алгоритм последовательной раскраски произвольного графа с N вершинами.

```
ColorGraph(G)
G   раскрашиваемый граф

for i=1 to N do
   c=1
   while в G есть вершина, соседняя с вершиной i,
```

```
        покрашенная цветом c do
      c=c+1
    end while
    покрасить вершину i цветом c
end for
```

Степенью графа называется наибольшее число ребер, выходящее из одной вершины. Число C красок, используемое приведенным алгоритмом, равно степени графа, увеличенной на единицу. Можно достичь и лучшего результата, но мы не обсуждаем таких алгоритмов в нашей книге.

9.1.6. Упражнения

1) Какой путь найдет алгоритм решения задачи коммивояжера на следующей матрице расстояний между городами?

Из/До	2	3	4	5	6	7
1	5	1	2	16	17	21
2		10	7	18	8	15
3			9	11	13	4
4				3	20	14
5					12	6
6						19

Оптимален ли этот путь?

2) Для задачи раскладки по ящикам имеется алгоритм, помещающий всякий объект таким образом, чтобы после его укладки в ящике оставалось как можно меньше места. Укладка в новый ящик применяется только в том случае, если очередной объект не помещается ни в какой из имеющихся ящиков. Запрограммируйте этот алгоритм. Покажите, как он работает на двух приведенных в тексте неотсортированных списках объектов.

3) Еще один алгоритм раскладки по ящикам основан на технике очередного подходящего: мы продолжаем укладку каждого ящика до тех пор, пока очередной объект в него помещается. Если объект не влазит в ящик, то мы берем следующий ящик, а к ранее уложенным ящикам не возвращаемся. Запрограммируйте этот алгоритм. Покажите, как он работает на двух приведенных в тексте неотсортированных списках объектов.

4) Еще один алгоритм раскладки по ящикам помещает всякий объект таким образом, чтобы после его укладки в ящике оставалось как можно больше места. Укладка в новый ящик применяется только в том случае, если очередной объект не помещается ни в какой из имеющихся ящиков. Запрограммируйте этот алгоритм. Покажите, как он работает на двух приведенных в тексте неотсортированных списках объектов.

5) Какой стоимости удастся достичь приближенному алгоритму упаковки рюкзака на списке ([5, 20], [10, 25], [15, 30], [20, 70], [25, 75], [30, 15], [35, 35], [40, 60]) при ограничении объема 55? Оптимален ли полученный результат?

6) Какого наилучшего результата удастся достичь приближенному алгоритму подсчета сумм элементов подмножеств для О, 1 и 2 проходов на наборе значений {29, 21, 16, 11, 3} при ограничении суммы 52? Оптимален ли полученный результат?

9.2. Вероятностные алгоритмы

Подход, используемый в вероятностных алгоритмах, в корне отличается от детерминированного подхода в алгоритмах из глав 1-7. В некоторых ситуациях вероятностные алгоритмы позволяют получить результаты, которых нельзя достигнуть обычными методами. В нашу задачу не входит исчерпывающее описание этого подхода. Приведенные примеры призваны лишь проиллюстрировать спектр имеющихся возможностей.

9.2.1. Численные вероятностные алгоритмы

Численные вероятностные алгоритмы предназначены для получения приблизительных ответов на некоторые математические вопросы. Чем дольше работает такой алгоритм, тем точнее полученный ответ.

Игла Бюффона

Предположим, что у Вас есть 355 одинаковых палочек, длина каждой из которых равна половине ширины доски дощатого пола. Сколько палочек пересечет щели между досками пола, если их подбросить и дать упасть на пол? Это может быть любое число между 0 и 355, однако Джордж Луис Леклерк показал, что в среднем число таких палочек

будет почти в точности равно 113. Для каждой палочки вероятность пересечь щель равна $1/\pi$. Объясняется такая величина соотношением между возможным углом поворота упавшей палочки и расстоянием между щелями. Для палочки, упавшей перпендикулярно к щелям, вероятность пересечения щели равна одной второй (это отношение длины палочки к ширине доски пола). Если, однако, палочка упала параллельно доске, то она почти наверняка не пересекает щель. Поэтому число тг можно вычислять, подбрасывая палочки и подсчитывая, сколько из них пересекло щели. Отношение общего числа палочек к числу тех из них, что пересекли щели, будет приближением числа тг.

Аналогичный способ приближенного подсчета числа тг заключается в бросании стрелок в мишень, представляющую собой квадрат, в который вписан круг (рис. 9.3). Мы выбираем точки в квадрате случайным образом и подсчитываем, какая их часть попала в круг. Если радиус круга равен r, то его площадь равна πr^2, а площадь квадрата равна $(2r)^2 = 4r^2$. Отношение площади круга к площади квадрата равно $\pi/4$. Если мы выбираем точки в квадрате действительно случайно, то стрелки распределятся по квадрату более или менее равномерно. Для случайного бросания стрелок в мишень число π приблизительно равно $4 * c/s$, где с — число стрелок, попавших в круг, а s — общее число брошенных стрелок. Чем больше стрелок брошено, тем более точное приближение к числу тг мы получаем.

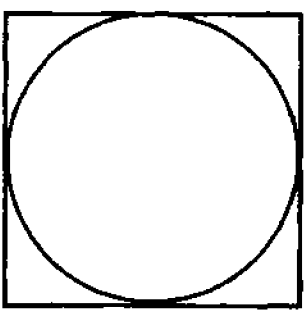


Рис. 9.3. Круг, вписанный в квадрат

С помощью этой техники можно подсчитать площадь произвольной неправильной фигуры, для которой мы умеем проверять, принадлежит ли ей заданная точка (x, y). Для этого мы просто генерируем случайные точки в объемлющем фигуру квадрате и подсчитываем отношение

числа тех из них, которые попали внутрь фигуры, к общему числу сгенерированных точек. Отношение

$$\frac{\text{число стрелок, попавших внутрь фигуры}}{\text{общее число брошенных стрелок}}$$

совпадает с отношением

$$\frac{\text{площадь фигуры}}{\text{площадь квадрата}}.$$

Метод Монте Карло[3]

Напомним, что для положительной непрерывной функции / площадь под ее графиком называется интегралом этой функции. Интегралы некоторых функций трудно или невозможно вычислять аналитически, однако их можно подсчитать приблизительно, воспользовавшись техникой бросания стрелок. Для иллюстрации этого метода ограничимся частью графика функции /, заключенной между положительными полуосями координат и прямыми $x = 1$ и $y = 1$ (см. рис. 9.4). Вам будет нетрудно обобщить рассмотрение на произвольный ограничивающий прямоугольник.

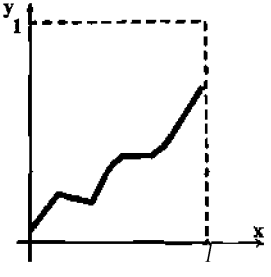


Рис. 9.4. График функции, ограниченный осями x к y и прямыми $x = 1$ и $y = 1$

Мы бросаем стрелки в квадрат случайным образом и подсчитываем, сколько из них оказались под графиком. Отношение числа стрелок под графиком к общему числу брошенных стрелок будет приблизительно равно площади под графиком функции. Как и в предыдущих случаях, чем больше стрелок мы бросим, тем точнее окажется приближение. Вот как выглядит соответствующий алгоритм:

[3] К сожалению, это стандартное название используемого метода, который не имеет ничего общего с обсуждающимися ниже алгоритмами Монте Карло.

```
Integrate(f,dartCount)
f          интегрируемая функция
dartCount число бросаемых стрелок

hits=0
for i=1 to dartCount do
   x=uniform(0,1)
   y=uniform(0,1)
   if y<=f(x) then
      hits=hits+1
   end if
end for
return hits/dartCount
```

Вероятностный подсчет

Как хорошо известно, можно держать пари, что в компании из 25 случайно выбранных людей есть по крайней мере двое с совпадающим днем рождения. Хотя на первый взгляд такое пари может показаться глупым, вероятность его выигрыша составляет 56%. В общем случае имеется $N!/(N-k)!$ способов выбрать k различных объектов в множестве из N объектов, если порядок выбора играет роль. Если же позволяются повторения, то число возможностей увеличивается до N^k. Используя эти сведения, мы заключаем, что вероятность выиграть пари равна $1 - 365!/(340! * 365^{25})$. На практике вероятность выигрыша даже выше, поскольку приведенное рассуждение не учитывает того, что дни рождения распределены по году неравномерно. Выписанное число не так-то легко вычислить, как нелегко решить и обратную задачу: сколько выборок нужно сделать из N-элементного множества прежде, чем мы повторно выберем один и тот же элемент? Другими словами, если в году 365 дней, то сколько людей надо взять, чтобы вероятность того, что среди них найдутся двое с одинаковым днем рождения, была достаточно велика? Для приближенного подсчета таких чисел можно применить следующий алгоритм:

```
ProbabilityCount(N)

k=0
s={}
a=uniform(1,N)
```

```
repeat
   k=k+1
   s=s+{a}
   a=uniform(1,N)
until a in s
return k
```

Эта функция генерирует случайное число между 1 и N до тех пор, пока какое-либо из чисел не будет сгенерировано повторно. Для получения более точного результата ее можно запускать несколько раз, а полученные ответы усреднять. В примере с днями рождения ($N = 365$) выдаваемые ответы будут близки к 25.

9.2.2. Алгоритмы Монте Карло

Алгоритмы Монте Карло всегда выдают какие-либо результаты, однако вероятность того, что результат правильный, возрастает при увеличении времени работы алгоритма. Иногда такие алгоритмы возвращают неправильные ответы. Алгоритм называется p-правильным, если он возвращает правильный ответ с вероятностью p ($1/2 < p < 1$). Если число правильных ответов на данном входе превышает единицу, то алгоритм Монте Карло называется стойким, если возвращаемый им правильный ответ всегда один и тот же.

Результаты алгоритма Монте Карло можно улучшить двумя способами. Во-первых, можно увеличить время его работы. Во-вторых, можно повторять его несколько раз. Вторая возможность реализуется только, если алгоритм стойкий. В этом случае алгоритм можно вызывать много раз и выбирать тот ответ, который встречается чаще всего. Подобные действия могут выглядеть приблизительно так:

```
Monte3(x)

one=Monte(x)
two=Monte(x)
three=Monte(x)
if one=two or one=three then
   return one
else
   return two
end if
```

Этот алгоритм возвращает первое сгенерированное значение в том случае, если оно появляется среди ответов по крайней мере дважды. Если же это не так, то алгоритм возвращает второе значение: либо оно совпадает с третьим, либо все три значения различны, и тогда все равно, какое возвращать. Поскольку вероятность возвращения алгоритмом Монте Карло правильного ответа превышает половину, маловероятно, чтобы все три ответа оказались различными. Процедура `Monte3` превращает стойкий 80%-правильный алгоритм Монте Карло в 90%-правильный. Такой подход к повышению правильности алгоритма не всегда является наилучшим.

Рассмотрим алгоритм Монте Карло принятия решения, возвращаемый которым отрицательный ответ оказывается правильным в 100% случаев, и только в случае положительного ответа могут встречаться ошибки, т.е. в случае положительного ответа правильный ответ может быть как положительным, так и отрицательным. Это означает, что полученный алгоритмом отрицательный ответ должен рассматриваться как окончательный, а повторные вызовы функции призваны лишь искать серии положительных ответов, чтобы увеличить вероятность того, что это действительно правильный ответ.

Вот как выглядит такой алгоритм:

```
MultipleMonte(x)

if not Monte(x) then
   return false
end if
if not Monte(x) then
   return false
end if
return Monte(x)
```

Такой алгоритм возвращает положительный ответ только в случае получения трех положительных ответов подряд. Если исходный алгоритм Монте Карло выдавал правильный положительный ответ лишь в 55% случаев, то описанная функция повышает вероятность правильных положительных ответов до 90%. Такое улучшение возможно и для численных алгоритмов, склонных выдавать одно и то же число.

Элемент, образующий большинство

Описанную выше технику можно применить к задаче проверки, есть ли в массиве элемент, образующий большинство. Это элемент, записанный более, чем в половине ячеек массива. Очевидный способ решения этой задачи требует порядка $O(N^2)$ сравнений, поскольку нам потребуется сравнить каждый элемент со всеми остальными. Но для ее решения известен и линейный алгоритм, похожий на алгоритм выборки из главы 2, поэтому приводимый ниже алгоритм Монте Карло призван лишь проиллюстрировать применяемую технику.

```
Majority(list,N)
list список элементов
N    число элементов в списке

choice=uniform(1,N)
count=0
for i=1 to N do
   if list[i]=list[choice] then
      count=count+1
   end if
end for
return (count>n/2)
```

Эта функция выбирает случайный элемент списка и проверяет, занимает ли он больше половины ячеек. Этот алгоритм смещен к утвердительному ответу: если функция возвращает утвердительный ответ, то это означает, что мы нашли элемент, представляющий большинство списка. Однако, если возвращается отрицательный ответ, то возможно, что мы просто выбрали неправильный элемент. Если в массиве есть элемент, образующий большинство, то вероятность выбрать элемент из меньшинства меньше половины, причем она тем меньше, чем представительнее большинство. Таким образом, алгоритм возвращает правильный ответ не менее, чем в 50% случаев. Если вызвать функцию Majority пять раз, то правильность алгоритма возрастает до 97%, а его сложность будет $5N$, т.е. она имеет порядок N.

Алгоритм Монте Карло проверки числа на простоту

Проверить, является ли данное число N простым, можно алгоритмом Монте Карло. В этом случае мы генерируем случайное число меж-

ду 2 и $\sqrt{N}$ и проверяем, делится ли N на это число. Если да, то число N составное, в противном случае мы не можем ничего сказать. Это не очень хороший алгоритм, потому что он возвращает отрицательный ответ слишком часто. Например, для числа 60 329, которое является произведением трех простых чисел 23, 43 и 61, алгоритм будет генерировать случайное число между 2 и 245, но только три числа из этого интервала привели бы к правильному результату. Вероятность успеха всего 1.2%.

Хотя этот простой алгоритм работает и не очень хорошо, имеются аналогичные подходы к проверке простоты числа, основанные на той же идее и дающие большую вероятность правильного ответа. В этой книге мы их не рассматриваем.

9.2.3. Алгоритмы Лас Вегаса

Алгоритмы Лас Вегаса никогда не возвращают неправильный ответ, хотя иногда они не возвращают вообще никакого ответа. Чем дольше работают эти алгоритмы, тем выше вероятность того, что они вернут правильный ответ. Алгоритм Лас Вегаса принимает случайное решение, а затем проверяет, приводит ли это решение к успеху. Программа, использующая алгоритм Лас Вегаса, вызывает его раз за разом, пока он не достигнет результата. Если обозначить через success(x) и failure(x) время, необходимое для того, чтобы получить соответственно положительный или отрицательный ответ на входных данных длины x, а через $p(x)$ вероятность успешного завершения работы алгоритма, то мы приходим к равенству

$$\text{time}(x) = p(x) * \text{success}(x) + (1 - p(x)) * (\text{failure}(x) + \text{time}(x)).$$

Это равенство означает, что в случае успеха затраченное время совпадает с временем получения успешного результата, а в случае неудачи затраченное время равно сумме времени на достижение неудачного результата и еще на один вызов функции. Решая это уравнение относительно times(x), мы получаем

$$\text{time}(x) = p(x) * \text{success}(x) + (1 - p(x)) * \text{failure}(x) + (1 - p(x))\text{time}(x)$$

$$\text{time}(x) - (1 - p(x))\text{time}(x) = p(x) * \text{success}(x) + (1 - p(x)) * \text{failure}(x)$$

$$\text{time}(x) - \text{time}(x) + p(x) * \text{time}(x) = p(x) * \text{success}(x) + (1 - p(x)) * \text{failure}(x)$$

$$p(x) * \text{time}(x) - p(x) * \text{success}(x) + (1 - p(x)) * \text{failure}(x)$$

$$\text{time}(x) = \text{success}(x) + ((1 - p(x))/p(x)) * \text{failure}(x)$$

Эта формула означает, что время выполнения зависит от времени получения успешного результата, безуспешного результата и вероятности каждого из этих исходов. Интересно, что при убывании вероятности $p(x)$ успешного результата время выполнения все равно может быть невысоким, если скорость получения безуспешного результата возрастает. Поэтому эффективность можно повысить, если быстрее получать безуспешный результат.

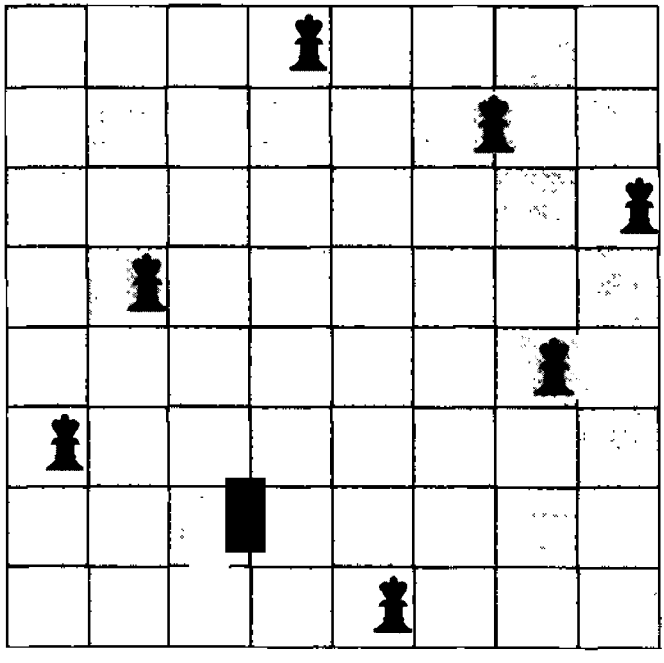


Рис. 9.5. Одно из решений задачи о восьми ферзях

Как такой подход работает на практике? Обратимся к задаче о расстановке восьми ферзей на шахматной доске так, чтобы они не били ДРУГ друга[4].

[4]Ферзь на шахматной доске атакует все поля, находящиеся с ним на одной горизонтали, на одной вертикали или на одной диагонали.

На рис. 9.5 изображено одно из решений этой задачи. Рекурсивный алгоритм ее решения помещает ферзя в первой клетке первой вертикали, а затем вызывает себя для того, чтобы поставить ферзя на вторую горизонталь. Если в какой-то момент алгоритму не удается найти положения для очередного ферзя на очередной горизонтали, то алгоритм возвращается на предыдущий шаг и пробует другое размещение ферзя на предыдущей строке.

Имеется вероятностная альтернатива детерминированному рекурсивному алгоритму. Мы можем поочередно размещать ферзей на доске случайным образом на очередной свободной горизонтали доски. Отличие алгоритма Лас Вегаса от стандартного рекурсивного алгоритма состоит в том, что при невозможности разместить очередного ферзя алгоритм попросту сдается и сообщает о неудаче. Рекурсивный же алгоритм пытается добиться положительного результата. Вот как выглядит алгоритм Лас Вегаса для расстановки восьми ферзей:

```
Queens(result)
result содержит номера вертикалей для ферзей
       с соответствующих горизонталей
возвращает 1 в случае успеха и 0 в случае неудачи

row=1
repeat
   // ферзи уже расставлены в горизонталях 1..row-1
   spotsPossible=0
   for i=1 to 8 do
      if клетка (row,i) атакована then
         spotsPossible=spotsPossible+1
         if uniform(1,spotsPossible)=1 then
            try=i
         end if
      end if
   end for
   if spotsPossible>0 then
      result[row]=try
      row=row+1
   end if
return  (spotsPossible>0)
```

Посмотрим, как работает этот алгоритм. В цикле repeat мы проходим по всем восьми горизонталям доски. Для каждой из горизонталей мы последовательно просматриваем все ее клеточки и если клетка не атакована, то переменная spotsPossible увеличивается на единицу. Следующий оператор if выглядит несколько странно, но посмотрим, что происходит, если опустить первую горизонталь, на которой не атакована ни одна клетка. На первой вертикали функция **uniform** генерирует случайное число между 1 и 1, т.е. 1, поэтому переменная try будет указывать на первую вертикаль. Во второй вертикали **uniform** генерирует число между 1 и 2, которое с 50%-ной вероятностью будет единицей и с 50%-ной вероятностью двойкой, поэтому вероятность того, что новым значением переменной try станет двойка, равна 50%. В третьей вертикали uniform генерирует число между 1 и 3; это число с вероятностью 33% будет 1, и также с вероятностью 33% значение try станет равно 3. Окончательно мы заключаем, что для каждой из свободных вертикалей вероятность быть опробованной на данном проходе равна 1/**spotsPossible**. Затем все повторяется для остальных горизонталей. Такие действия продолжаются до тех пор пока либо значение spotsPossible не станет нулевым ввиду отсутствия неатакованных клеток, либо переменная rows не примет значение 9, поскольку все ферзи будут расставлены. В первом случае алгоритм завершает свою работу и сообщает о неудачном исходе. Во втором проблема расстановки восьми ферзей решена, и алгоритм сообщает об удачном исходе.

Полный статистический анализ алгоритма показывает, что вероятность успеха равна 0.1293, а среднее число повторений, необходимых для его достижения, около 6.971. Приведенное выше уравнение показывает, что алгоритм выполнит при этом 55 проходов. Рекурсивному же алгоритму понадобится по крайней мере вдвое больше проходов.

9.2.4. Шервудские алгоритмы

Шервудские алгоритмы всегда возвращают ответ, и этот ответ всегда правильный. Эти алгоритмы применимы в ситуациях, когда различие между наилучшим, средним и наихудшим случаями в детерминированном алгоритме очень велико. Применение случайности позволяет шервудским алгоритмам сократить спектр возможностей, подтянув наихудший и наилучший случаи к среднему.

Примером может служить поиск осевого элемента в алгоритме быстрой сортировки. При анализе этого алгоритма мы пришли к выводу, что наихудшим для него является ситуация, в которой список уже от-

сортирован, так как каждый раз мы будем натыкаться на минимальный элемент списка. Если же вместо выбора начального элемента мы будем выбирать случайный элемент между началом и концом, то вероятность наихудшего исхода уменьшится. Совсем избежать его нельзя — и при случайном выборе оси мы можем всякий раз натыкаться на наименьший элемент, однако вероятность такого исхода очень мала. Оборотная сторона такого подхода состоит в том, что если в списке реализовался наилучший исход для детерминированного алгоритма — первым элементом всякий раз оказывается медиана оставшейся части списка, — то маловероятно, чтобы наш случайный выбор пал именно на нее. Поэтому шансы на наилучший и наихудший исход понижаются.

Тот же самый подход можно применять и в задаче поиска. При двоичном поиске прежде, чем добраться до некоторых элементов списка, мы должны с необходимостью совершить несколько неудачных проверок. Шервудский вариант «двоичного» поиска выбирает случайный элемент между началом и концом списка и сравнивает его с искомым. Иногда оставшийся кусок оказывается меньше, чем тот, что мы бы получили при настоящем двоичном поиске, а иногда — больше. Так, например, не исключено, что в списке из 400 элементов мы выберем для пробного сравнения не 200-ый, а сотый. Если искомый элемент находится среди первых ста, то наша шервудская версия алгоритма поиска позволяет отбросить 75% списка вместо 50% в стандартном алгоритме. Однако если интересующий нас элемент больше сотого, то отброшенными окажутся лишь 25% списка. Вновь в некоторых случаях мы получаем выигрыш, а иногда проигрываем.

Как правило, шервудские алгоритмы уменьшают время обработки наихудшего случая, однако увеличивают время обработки наилучшего. Подобно Робин Гуду из Шервудского леса этот подход грабит богатых, чтобы отдать бедным.

9.2.5. Сравнение вероятностных алгоритмов

Подведем итоги обсуждению алгоритмов в этом параграфе. Численные вероятностные алгоритмы всегда дают ответ, и этот ответ будет тем точнее, чем дольше они работают. Алгоритмы Монте Карло всегда дают ответ, но иногда ответ оказывается неправильным. Чем дольше выполняется алгоритм Монте Карло, тем выше вероятность того, что он даст правильный ответ. Повторный вызов алгоритма Монте Карло также приводит к улучшению результата. Алгоритмы Лас Вегаса не могут вернуть неправильного результата, но они могут и не

вернуть никакого результата, если им не удалось найти правильный ответ. Шервудскую технику можно применять к любому детерминированному алгоритму. Она не влияет на правильность алгоритма, а лишь уменьшает вероятность реализации наихудшего поведения. Вероятность наилучшего поведения при этом, правда, тоже понижается.

9.2.6. Упражнения

1) Во время последней вылазки в лес Вы нашли пещеру, карту, компьютер и волшебную кнопку на стене. На карте изображены два острова, оба на расстоянии пяти дней пути от Вашего местонахождения, и также на расстоянии пяти дней пути друг от друга. (Можно представлять себе, что Вы и два острова находитесь в вершинах равностороннего треугольника.) Появившийся гном объясняет Вам, что при нажатии кнопки сокровища — 15 слитков чистого золота, инкрустированного алмазами, изумрудами, рубинами и другими бесценными драгоценностями, -- случайно возникают на одном из двух островов. Компьютер начинает вычислять положение сокровищ, однако у него уйдет на это четыре дня. Компьютер стационарный, поэтому до завершения его работы Вам придется ждать четыре дня на месте. Проблема в том, что каждую ночь дракон уносит по одному слитку в абсолютно недоступное место. Гном готов сообщить местонахождение сокровищ в обмен на три слитка. Кроме того гном говорит, что Вы можете нажимать на кнопку всякий раз после возвращения из путешествия, и сокровища опять будут появляться случайно на одном из островов.

   Рассмотрите *все* возможные варианты Вашего поведения и потенциальную прибыль от каждого из них, и решите, какой из вариантов даст наилучший результат в далекой перспективе (т.е. если Вы предпримете неоднократную охоту за сокровищами). В Ваш список должны войти все рассмотренные Вами исходы и полученный в случае их наступления выигрыш. Предположите, что охота состоится по крайней мере 10 раз. Поскольку число возможностей отправления на охоту предполагается неограниченным, время возвращения в пещеру не следует принимать во внимание.

   Для решения задач со второй по четвертую воспользуйтесь таблицей случайных чисел из приложения А. В каждой из задач начинайте с начала таблицы и двигайтесь по ней, пока либо задача

не будет решена, либо не закончится таблица. Если таблица закончится, вернитесь к ее началу и продолжите решение задачи.

2) Непосредственное интегрирование функции x^3 на отрезке $[0, 1]$ дает результат 0.25. Воспользуйтесь для интегрирования функцией Integrate из § 9.2 с двадцатью стрелками. Сравните ответы, полученные после бросания 5, 10, 15, 20 стрелок с истинным ответом 0.25. (Покажите всю проделанную работу.)

3) Выполните проверку числа на простоту алгоритмом Монте Карло для чисел $182 = 2 \cdot 7 \cdot 13$ и $255 = 3 \cdot 5 \cdot 17$, выписывая случайно выбранные числа и полученный результат. Сколько раз приходится выполнять проверку прежде, чем будет получен правильный результат? (В каждом случае начинайте с начала таблицы случайных чисел.)

4) Изобразите первые три расстановки ферзей на шахматной доске, порождаемые алгоритмом Queens при попытках поиска правильной расстановки.

5) Модифицируйте алгоритм `PivotList` на основе шервудской техники.

6) Запрограммируйте описанный в разделе 9.2.4 алгоритм шервудского поиска на основе алгоритма двоичного поиска.

9.3. Динамическое программирование

Ричард Беллман ввел понятие *динамическое программирование* для характеристики алгоритмов, действующих в зависимости от меняющейся ситуации. Ключевым в его подходе является использование полиномиальных по времени алгоритмов вместо экспоненциальных. Мы рассмотрим два применения динамического программирования: способ повышения эффективности некоторых рекурсивных алгоритмов и способ выбора порядка последовательного умножения матриц, уменьшающий полное время умножения. Еще одно применение динамического программирования — алгоритм приближенного сравнения строк — обсуждалось в главе 5.

9.3.1. Программирование на основе массивов

Программирование с использованием массивов позволяет избежать многократного подсчета одного и того же результата в традиционных рекурсивных алгоритмах. Классическим примером рекурсивного алгоритма служит подсчет чисел Фибоначчи, к которому мы обращались в задаче 1 из раздела 1.5.3. При трассировке рекурсивного алгоритма подсчета десятого числа Фибоначчи видно, что сначала мы должны вычислить девятое и восьмое числа Фибоначчи и сложить их между собой. При традиционном подходе восьмое число Фибоначчи будет вычисляться и при вычислении девятого, но полученный при этом результат будет забыт, а потом вычислен заново. Воспроизведем алгоритм из главы 1:

```
Fibonacci(N)
N   номер возвращаемого числа Фибоначчи

if (N=1) or (N=2) then
   return 1
else return Fibonacci(N-1)+Fibonacci(N)
end if
```

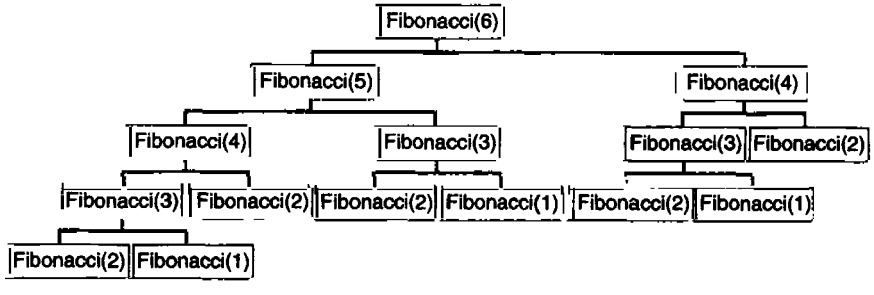


Рис. 9.6. Последовательность вызовов алгоритма Фибоначчи (6)

Применяя этот алгоритм к подсчету шестого числа Фибоначчи, мы получим последовательность вызовов, изображенную на рис. 9.6. Исследуя это дерево, мы заключаем, что вызов Fibonacci (4) будет происходить дважды, а вызов `Fibonacci(3)` — три раза. При возрастании номера вычисляемого числа количество одинаковых вызовов будет лишь увеличиваться. При подсчете десятого числа Фибоначчи третье будет вычисляться 21 раз. Эффективность алгоритма можно повысить,

если заменить движение с конца к началу движением от начала к концу. Альтернативный алгоритм подсчета использует массив предыдущих значений:

```
Fibonacci2(N)
N    номер возвращаемого числа Фибоначчи

if (N=1) or (N=2) then
   return 1
else
   val[1]=1
   val[2]=1
   for i=3 to N do
      val[i]=val[i-1]+val[i-2]
   end for
   return val[N]
end if
```

Это не очень сложный пример, и Вам не составит труда обойтись всего двумя переменными для хранения последних двух элементов массива вместо того, чтобы хранить весь массив.

Более сложный пример представляет собой вычисление биномиальных коэффициентов, которые подсчитывают число различных способов выбрать k объектов в множестве из N различных объектов, если порядок выбранных объектов нас не интересует[5]. Биномиальный коэффициент дается равенством

$$\binom{N}{k} = \frac{N!}{k!(N-k)!}.$$

Вычисления непосредственно по этой формуле далеко не уведут, поскольку факториал растет очень быстро. Альтернативный способ подсчета биномиальных коэффициентов, эквивалентный предложенной формуле, дается следующим соотношением

$$\binom{N}{k} = \begin{cases} 1 & k = 0, \\ 1 & k = N, \\ \binom{N-1}{k-1} + \binom{N-1}{k} & 0 < k < N. \end{cases}$$

[5]Приведенная в разделе 9.2.1 формула отлична от нижеследующей, поскольку там мы учитывали порядок выбираемых объектов.

По выписанному соотношению несложно написать рекурсивный алгоритм, однако он будет страдать теми же недостатками — многократным вычислением одних и тех же величин, — что и рекурсивный алгоритм вычисления чисел Фибоначчи. Вместо этого начнем вычисления с начала и будем продвигаться к нужному биномиальному коэффициенту пока не достигнем его. Этот процесс знаком всем, кто когда-либо выписывал треугольник Паскаля.

Для реализации алгоритма нам понадобится массив BiCoeff из $N+1$ строчек и $k+1$ столбцов, которые мы будем нумеровать начиная с нуля. В элементе `BiCoeff[i,j]` этого массива будет записано значение $\binom{N}{k}$. Сначала мы заносим в каждую из ячеек `BiCoeff[0,0]`, `BiCoeff[0,0]`, `BiCoeff[0,0]` значение 1, а затем устраиваем цикл пока не дойдем до ячейки `BiCoeff[N,k]`. Вот решение этой задачи методами динамического программирования:

```
for i=0 to N do
   for j=0 to min(i,k) do
      if j=1 or j=i then
         Bicoeff[i,j]=1
      else
         BiCoeff[i,j]=BiCoeff[i-1,j-1]+Bicoeff[i-1,j]
      end if
   end for j
end for i
```

Рекурсивному алгоритму придется вычислить $2\binom{N}{k}-N$ промежуточных биномиальных коэффициентов, а с помощью динамического программирования мы снизили это число до $O(Nk)$.

9.3.2. Динамическое умножение матриц

При необходимости перемножить последовательность матриц различных размеров порядок умножения может существенно повлиять на эффективность всей процедуры. Если, например, нам нужно найти произведение четырех матриц M_1, М2, Мз и M_4 размеров соответственно 20 х 5, 5 х 35, 35 х 4 и 4 х 25, то имеется пять существенно различных порядков их умножения, которые потребуют от 3100 до 24 500 операций умножения. Подробности представлены на рис. 9.7. В этой таблице описаны различные способы выбора первой пары умножаемых матриц,

Порядок выполнения умножения	Размер результирующей матрицы	Общая стоимость умножения
M_1M_2	20 x 35	3500
M_2M_3	5 x 4	700
M_3M_4	35 x 25	3500
$(M_1M_2)M_3$	20 x 4	3500 + 2800 = 6300
$M_1(M_2M_3)$	20 x 4	700 + 400 = 1100
$(M_2M_3)M_4$	5 x 25	700 + 500 = 1200
$M_2(M_3M_4)$	5 x 25	3500 + 4375 = 7875
$((M_1M_2)M_3)M_4$	20 x 25	6300 + 2000 = 83000
$(M_1(M_2M_3))M_4$	20 x 25	1100 + 2000 = 3100
$(M_1(M_2M_3)M_4$	20 x 25	1200 + 2000 = 3100
$M_1((M_2M_3)M_4)$	20 x 25	1200 + 2500 = 3700
$M_1(M_2(M_3M_4))$	20 x 25	7875 + 2500 = 10375
$(M_1M_2)(M_3M_4)$	20 x 25	3500 + 3500 + 17 500 = 24 500

Рис. 9.7. Объем работы при умножении матриц в различных порядках

затем присоединения к ним третьей матрицы, а затем и четвертой. В последнем столбце показано число операций умножения при соблюдении порядка, указанного в первом столбце. Напомним, что умножение матрицы порядка A x B на матрицу порядка $B \times C$ требует ABC операций умножения.

Нижеследующий алгоритм строит верхнетреугольную матрицу, в которую записаны минимальные стоимости с рис. 9.7. Размер матрицы M_j равен Sj x s_{j+1}. По завершении работы алгоритма значение минимальной стоимости будет занесено в ячейку `cost[1,N]` (другими словами, в правую верхнюю клетку матрицы). Кроме того результатом алгоритма будет матрица trace, на основе которой следующий алгоритм выберет сам порядок умножения, приводящий к минимальной стоимости.

```
for i=1 to N do
  cost [i,i]=0
end for
for i=1 to N-1 do
  for j=1 to N-1 do
    loc=i+j
    tempCost=infinity
    for k=1 to loc-1 do
     if tempCost>cost[i,k]+cost[k+1,loc]+s[i]*s[k]*s[loc] then
```

```
        tempCost=cost[i,k]+cost[k+1,loc]+s[i]*s[k]*s[loc]
        tempTrace=k
        end if
      end for k
      cost[i,loc]=tempCost
      trace[i,loc]=tempTrace
    end for j
end for i
```

После того, как массив trace вычислен, следующий рекурсивный алгоритм определяет на его основе порядок умножений. Глобальная переменная **position** в этом алгоритме принимает начальное значение 1 и сохраняет изменения, производимые при рекурсивных вызовах.

```
GetOrder(first,last,order)
first номер первой умножаемой матрицы
last  номер последней умножаемой матрицы
order порядок умножения матриц

if first<last then
   middle=trace[first,last]
   GetOrder(first,middle,order)
   GetOrder(middle,last,order)
   order[position]=middle
   position=position+1
end if
```

Применив этот алгоритм к размерам матриц, приведенным в примере, мы получим порядок 2, 1, 3, т.е. сначала должно выполняться второе умножение, затем первое, затем третье. К такому же выводу мы пришли и на основе рис. 9.7.

9.3.3. Упражнения

1) С помощью алгоритмов из раздела 9.3.2 определите наиболее эффективный порядок умножения матриц в каждом из следующих четырех случаев:

   а) M_1 размером 3 x 5, M_2 размером 5 × 2, M_3 размером 2 × 1, M_4 размером 1 x 10;

б) M_1 размером 2 × 7, M_2 размером 7 × 3, M_3 размером 3 × 6, M4 размером 6 x 10;

в) M_1 размером 10 x 3, M_2 размером 3 x 15, M_3 размером 15 x 12, M_4 размером 12 x 7, M_5 размером 7 × 2;

г) M_1 размером 7 × 2, M_2 размером 2 × 4, M_3 размером 4 x 15, M_4 размером 15 x 20, M_5 размером 20 x 5.

9.4. Упражнения по программированию

1) Реализуйте приближенный алгоритм решения задачи о коммивояжере. Выполните Вашу программу на наборе из 10 городов, расстояния между которыми равномерно распределены на интервале между 1 и 45. Выведите на печать матрицу примыканий и полученный результат. Проверьте результат на оптимальность.

2) Напишите приближенный алгоритм раскладки по ящикам. В Вашей программе должны быть реализованы все четыре рассмотренных в тексте и упражнениях стратегии: первый подходящий, наиболее подходящий, следующий подходящий и наименее подходящий ящик. Сгенерируйте случайный набор объектов, осуществите на нем каждую из четырех стратегий и посмотрите, какая из них приводит к меньшему числу ящиков. Сведите полученные результаты в отчет. Проверьте все методы на множествах из 50, 100, 200 и 500 объектов. Для получения более точных результатов проверьте алгоритмы на разнообразных случайных наборах объектов.

3) Напишите программу интегрирования функции x^3 методом Монте Карло на отрезке $[0, 1]$. Мы знаем, что ответ 0.25, поэтому основная цель программы -- посмотреть, сколько стрелок нужно бросить, чтобы получить приближение с той или иной точностью. Выполните программу так, чтобы достичь точности ± 0.0001, ± 0.000001 и ± 0.00000001.

4) Напишите программу описанного в тексте приближенного вычисления числа π. Сколько стрелок нужно бросить для получения пятой верной цифры после запятой? Сколько стрелок нужно бросить для получения седьмой и десятой верной цифры после запятой?

5) Разработайте алгоритм сортировки списка, использующий стандартную процедуру Quicksort и процедуру `Quicksort` с шервудской подпрограммой PivotList. Сгенерируйте несколько случайных списков из 500 значений и подсчитайте число сравнений, выполняемых обеими сортировками в каждом из случаев. Подсчитайте максимальное, минимальное и среднее количество выполненных сравнений. Составьте отчет по результатам Ваших экспериментов. (Дополнительные указания по организации работы даны в упражнениях по программированию в главе 3.)

6) Напишите программу шервудского двоичного поиска. Создайте упорядоченный список из чисел от 1 до 10 000. Сгенерируйте последовательность значений в этом интервале и прогоните их через обычный и шервудский двоичный поиск. Напишите сравнительный отчет с анализом максимального, минимального и среднего числа сравнений, выполняемых обоими алгоритмами. Чем больше чисел Вы испытаете, тем точнее будут результаты.

Приложение А.

Таблица случайных чисел

Ниже приведена таблица случайных чисел между 0 и 1. Если Вам нужна последовательность случайных чисел между 0 и N, то просто умножьте числа из таблицы на N. Если Вам нужны случайные числа в границах low и high, то умножьте значение в таблице на разность high-low и прибавьте к результату значение low.

0	0.21132	20	0.92500	40	0.28029
1	0.26215	21	0.46777	41	0.73297
2	0.79253	22	0.33873	42	0.00309
3	0.28952	23	0.30228	43	0.31992
4	0.93648	24	0.27223	44	0.76521
5	0.93726	25	0.57355	45	0.47253
6	0.35606	26	0.96965	46	0.84203
7	0.16043	27	0.14291	47	0.45840
8	0.40480	28	0.56575	48	0.64955
9	0.74225	29	0.94983	49	0.87323
10	0.70183	30	0.71092	50	0.74374
11	0.41904	31	0.13687	51	0.21248
12	0.75691	32	0.19618	52	0.47449
13	0.00524	33	0.17474	53	0.30492
14	0.59544	34	0.57817	54	0.16348
15	0.51846	35	0.98727	55	0.75307
16	0.38344	36	0.80415	56	0.40643
17	0.30438	37	0.07641	57	0.73857
18	0.05253	38	0.83702	58	0.25217
19	0.16183	39	0.64725	59	0.83369

60	0.32764	100	0.47643	140	0.90770
61	0.62633	101	0.93607	141	0.27310
62	0.96292	102	0.48024	142	0.98280
63	0.34499	103	0.87140	143	0.10394
64	0.31622	104	0.56047	144	0.29839
65	0.48381	105	0.16733	145	0.17819
66	0.49887	106	0.35188	146	0.55171
67	0.42757	107	0.26331	147	0.74780
68	0.70032	108	0.00486	148	0.45567
69	0.07664	109	0.80191	149	0.76785
70	0.31314	110	0.81044	150	0.36943
71	0.47206	111	0.75385	151	0.88635
72	0.05804	112	0.82524	152	0.36378
73	0.42046	113	0.54294	153	0.76584
74	0.10886	114	0.49654	154	0.66698
75	0.11909	115	0.17114	155	0.02154
76	0.21753	116	0.28722		
77	0.78087	117	0.34354		
78	0.83914	118	0.30080		
79	0.25929	119	0.59332		
80	0.25690	120	0.90642		
81	0.67351	121	0.40683		
82	0.70712	122	0.36385		
83	0.03327	123	0.34851		
84	0.50427	124	0.44847		
85	0.86400	125	0.18594		
86	0.16592	126	0.07630		
87	0.83168	127	0.01483		
88	0.53778	128	0.92900		
89	0.36797	129	0.38400		
90	0.91867	130	0.07881		
91	0.25512	131	0.42041		
92	0.18555	132	0.61363		
93	0.45103	133	0.95413		
94	0.91849	134	0.26198		
95	0.31422	135	0.64337		
96	0.52570	136	0.01799		
97	0.62883	137	0.09945		
98	0.36850	138	0.76643		
99	0.02961	139	0.01184		

Приложение Б.

Генерация псевдослучайных чисел

Случайные числа имеют разнообразные компьютерные приложения. Однако никакой алгоритмический процесс не способен сгенерировать по-настоящему случайные числа: при повторном запуске такого процесса мы получим ту же самую последовательность чисел. Случайный процесс не должен обладать таким свойством, и случайность можно сымитировать, запуская программу всякий раз с новой точки. Начальная точка обычно вычисляется по текущему значению системных часов компьютера, и к нашему генератору добавляется тем самым случайность положения часов.

Использование постоянной случайной последовательности облегчает тестирование программы: после каждого исправления мы можем повторять ее на тех же самых данных. После того, как мы сочли, что программа работает правильно, можно добавить случайный выбор начальной точки.

Существуют различные способы генерации случайных последовательностей; здесь мы рассмотрим лишь один из них. В методе смешанных вычетов очередное случайное число вычисляется на основе предыдущего. Вот как выглядит такой алгоритм:

```
function RanNum() returns float
    seed=(seed*p+i) mod m
    return seed/m
end RanNum
```

Значение seed вычисляется по модулю m, поэтому оно всегда находится в интервале $[0, m - 1]$. После деления на m мы заведомо получаем число из отрезка $[0, 1]$, которое и возвращается функцией.

Если три константы p, $г$ и m взаимно просты, то есть не имеют общего делителя, то период получившейся последовательности равен m — раньше повторений не будет. Если, например, p = 25 173, $г$ = 13 849 и m — 65 536, то написанная функция сгенерирует последовательность из 65 536 попарно различных значений прежде, чем значения начнут повторяться. При тестировании программ можно положить начальное значение seed равным 0 — тогда будет всякий раз генерироваться одна и та же последовательность. После завершения тестирования можно в качестве начального значения переменной seed выбирать число секунд или миллисекунд текущего времени на системных часах — последовательность будет более случайной.

Б.1. Случайная последовательность в произвольном интервале

Зачастую в приложениях требуются случайные последовательности не из интервала [0, 1], а из других интервалов. Если нам нужна последовательность в интервале `Low<=N<High`, то ее сгенерирует функция

```
(High-Low)*RanNum()+Low
```

Б.2. Пример применения

Допустим, что нам нужен список чисел от 1 до N в случайном порядке. Есть несколько возможностей сгенерировать такой список.

Б.2.1. Первый способ

Инициализируем элементы списка нулевыми значениями; тогда по мере помещения элементов в список еще не занятые ячейки будут оставаться нулевыми. Если выбранная нами случайная ячейка содержит нуль, то она еще не занята, и мы можем поместить в нее очередное число. Если же она занята, то мы просто генерируем случайный номер следующей ячейки. В результате мы приходим к такому алгоритму:

```
for i=1 to N do
   list[i]=0
end for
for i=1 to N do
   repeat
      location=[N*RanNum()+1]
```

```
   until list[location]=0
   list[location]=i
end for
```

Первые несколько значений быстро займут свои места в списке; проблема этого метода, однако, заключается в том, что по мере заполнения списка становится все труднее и труднее помещать в него очередное значение. Поэтому даже в такой простой ситуации программа может работать довольно долго.

Б.2.2. Второй способ

Как мы видели, проблема с реализацией первого способа возникает, если выбранная случайная ячейка оказывается «заполненной». В предлагаемом ниже варианте, обнаружив заполненную ячейку, мы просто проверяем следующие за ней до тех пор, пока не найдем свободную. В результате мы приходим к следующему алгоритму:

```
for i=1 to N do
   list[i]=0
end for
for i=1 to N do
   location=[N*[RanNum()+1]
   while list[location]/=0 do
      location=(location mod N)+1
   end while
   list[location]=i
end for
```

Этот способ работает достаточно быстро, однако если в списке быстро появится блок из подряд идущих заполненных клеток, то у нас будет довольно длинный подсписок, содержащий числа с сохранением их относительного порядка. Пусть, например, первые 25 из 100 ячеек уже заполнены случайными числами. Тогда имеется 25%-ная вероятность того, что очередная выбранная ячейка окажется среди первых 25, и соответствующее значение будет занесено в ячейку 26. При последующих выпаданиях одной из первых ячеек мы занесем соответствующие значения в 27, 28 и т.д. ячейки. Такое может случиться в любой части списка, и мы получим блок ячеек, значения которых последовательны или почти последовательны.

Б.2.3. Третий способ

В этом последнем способе сгенерированное случайное число указывает, сколько пустых ячеек, считая от текущей, следует пропустить прежде, чем заносить в список очередное число. В результате мы преодолеваем недостаток первого метода, поскольку используем каждое сгенерированное случайное число. Кроме того, снижается и вероятность записи последовательных значений в последовательные ячейки — такое может произойти только, если последовательные ячейки свободны, а сгенерированное число сравнимо с единицей по модулю числа оставшихся свободными ячеек. Вот реализация этого алгоритма:

```
for i=1 to N do
   theList[i]=0
end for
location=1
freeCount=N
for i=1 to N do
   skip=[freeCount*RanNum()+1]
   while skip>0 do
      location=(location mod N)+1
      if theList[location]=0 then
         skip=skip-1
      end if
   end while
   theList[location]=i
   freeCount=freeCount-1
end for
```

В этом алгоритме мы ведем счетчик свободных ячеек и генерируем случайное число, не превышающее значения счетчика. Это делается для того, чтобы не было необходимости проходить по списку несколько раз. Видно, что цикл while всегда завершается, поскольку даже на последнем проходе остается по крайней мере одна пустая ячейка.

Б.3. Итоги

В этом приложении описан один из способов генерации псевдослучайных чисел, а также варианты его использования для создания случайного списка. С помощью описанных методов можно создавать списки, в которых можно в дальнейшем вести поиск, и которые можно сортировать для экспериментальной оценки сложности алгоритмов из глав 2 и 3.

Приложение В.

Ответы к упражнениям

В этом приложении приведены результаты ручного исполнения предложенных в книге алгоритмов на предложенных входных данных.

Алгоритмы из главы 3

Сортировка вставками

Исходный список:	6	2	4	7	1	3	8	5
Проход 1:	2	6	4	7	1	3	8	5
Проход 2:	2	4	6	7	1	3	8	5
Проход 3:	2	4	6	7	1	3	8	5
Проход 4:	1	2	4	6	7	3	8	5
Проход 5:	1	2	3	4	6	7	8	5
Проход 6:	1	2	3	4	6	7	8	5
Проход 7:	1	2	3	4	5	6	7	8

Исходный список:	15	4	10	8	6	9	16	1	7	3	11	14	2	5	12	13
Проход 1:	4	15	10	8	6	9	16	1	7	3	11	14	2	5	12	13
Проход 2:	4	10	15	8	6	9	16	1	7	3	11	14	2	5	12	13
Проход 3:	4	8	10	15	6	9	16	1	7	3	11	14	2	5	12	13
Проход 4:	4	6	8	10	15	9	16	1	7	3	11	14	2	5	12	13
Проход 5:	4	6	8	9	10	15	16	1	7	3	И	14	2	5	12	13
Проход 6:	4	6	8	9	10	15	16	1	7	3	11	14	2	5	12	13
Проход 7:	1	4	6	8	9	10	15	16	7	3	11	14	2	5	12	13
Проход 8:	1	4	6	7	8	9	10	15	16	3	11	14	2	5	12	13
Проход 9:	1	3	4	6	7	8	9	10	15	16	11	14	2	5	12	13
Проход 10:	1	3	4	6	7	8	9	10	И	15	16	14	2	5	12	13
Проход 11:	1	3	4	6	7	8	9	10	11	14	15	16	2	5	12	13

Проход 12:	1	2	3	4	6	7	8	9	10	11	14	15	16	5	12	13
Проход 13:	1	2	3	4	5	6	7	8	9	10	11	14	15	16	12	13
Проход 14:	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16	13
Проход 15:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Пузырьковая сортировка

Исходный список:	6	2	4	7	1	3	8	5
Проход 1:	2	4	6	1	3	7	5	8
Проход 2:	2	4	1	3	6	5	7	8
Проход 3:	2	1	3	4	5	6	7	8
Проход 4:	1	2	3	4	5	6	7	8
Проход 5:	1	2	3	4	5	6	7	8

Исходный список:	15	4	10	8	6	9	16	1	7	3	11	14	2	5	12	13
Проход 1:	4	10	8	6	9	15	1	7	3	11	14	2	5	12	13	16
Проход 2:	4	8	6	9	10	1	7	3	11	14	2	5	12	13	15	16
Проход 3:	4	6	8	9	1	7	3	10	11	2	5	12	13	14	15	16
Проход 4:	4	6	8	1	7	3	9	10	2	5	11	12	13	14	15	16
Проход 5:	4	6	1	7	3	8	9	2	5	10	11	12	13	14	15	16
Проход 6:	4	1	6	3	7	8	2	5	9	10	11	12	13	14	15	16
Проход 7:	1	4	3	6	7	2	5	8	9	10	11	12	13	14	15	16
Проход 8:	1	3	4	6	2	5	7	8	9	10	11	12	13	14	15	16
Проход 9:	1	3	4	2	5	6	7	8	9	10	11	12	13	14	15	16
Проход 10:	1	3	2	4	5	6	7	8	9	10	11	12	13	14	15	16
Проход 11:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Проход 12:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Сортировка Шелла

Исходный список:	6	2	4	7	1	3	8	5
После прохода с шагом 7:	5	2	4	7	1	3	8	6
После прохода с шагом 3:	5	1	3	7	2	4	8	6
После прохода с шагом 1:	1	2	3	4	5	6	7	8

Исходный список:	15	4	10	8	6	9	16	1	7	3	11	14	2	5	12	13
После прохода с шагом 15:	13	4	10	8	6	9	16	1	7	3	11	14	2	5	12	15
После прохода с шагом 7:	1	4	3	8	6	2	5	12	7	10	11	14	9	16	13	15
После прохода с шагом 3:	1	4	2	5	6	3	8	11	7	9	12	13	10	16	14	15
После прохода с шагом 1:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Пирамидальная сортировка

Исходный список:	6	2	4	7	1	3	8	5
После построения пирамиды:	8	7	6	5	1	3	4	2
Проход 1:	7	5	6	2	1	3	4	8
Проход 2:	6	5	4	2	1	3	7	8
Проход 3:	5	3	4	2	1	6	7	8
Проход 4:	4	3	1	2	5	6	7	8
Проход 5:	3	2	1	4	5	6	7	8
Проход 6:	2	1	3	4	5	6	7	8
Проход 7:	1	2	3	4	5	6	7	8

Исходный список:	15	4	10	8	6	9	16	1	7	3	11	14	2	5	12	13
После построения пирамиды:	16	13	15	8	11	14	12	4	7	3	6	9	2	5	10	1
Проход 1:	15	13	14	8	11	9	12	4	7	3	6	1	2	5	10	16
Проход 2:	14	13	12	8	11	9	10	4	7	3	6	1	2	5	15	16
Проход 3:	13	11	12	8	6	9	10	4	7	3	5	1	2	14	15	16
Проход 4:	12	11	10	8	6	9	2	4	7	3	5	1	13	14	15	16
Проход 5:	11	8	10	7	6	9	2	4	1	3	5	12	13	14	15	16
Проход 6:	10	8	9	7	6	5	2	4	1	3	11	12	13	14	15	16
Проход 7:	9	8	5	7	6	3	2	4	1	10	11	12	13	14	15	16
Проход 8:	8	7	5	4	6	3	2	1	9	10	11	12	13	14	15	16
Проход 9:	7	6	5	4	1	3	2	8	9	10	11	12	13	14	15	16
Проход 10:	6	4	5	2	1	3	7	8	9	10	11	12	13	14	15	16
Проход 11:	5	4	3	2	1	6	7	8	9	10	11	12	13	14	15	16
Проход 12:	4	2	3	1	5	6	7	8	9	10	11	12	13	14	15	16
Проход 13:	3	2	1	4	5	6	7	8	9	10	11	12	13	14	15	16
Проход 14:	2	1	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Проход 15:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Сортировка слиянием

Исходный список:	6	2	4	7	1	3	8	5
После слияния элементов 1-2:	2	6	4	7	1	3	8	5
После слияния элементов 3-4:	2	6	4	7	1	3	8	5
После слияния элементов 1-4:	2	4	6	7	1	3	8	5
После слияния элементов 5-6:	2	4	6	7	1	3	8	5
После слияния элементов 7-8:	2	4	6	7	1	3	5	8
После слияния элементов 5-8:	2	4	6	7	1	3	5	8
После слияния элементов 1-8:	1	2	3	4	5	6	7	8

Исходный список:	15	4	10	8	6	9	16	1	7	3	11	14	2	5	12	13
После слияния элементов 1-2:	4	15	10	8	6	9	16	1	7	3	11	14	2	5	12	13
После слияния элементов 3-4:	4	15	8	10	6	9	16	1	7	3	11	14	2	5	12	13
После слияния элементов 1-4:	4	8	10	15	6	9	16	1	7	3	11	14	2	5	12	13
После слияния элементов 5-6:	4	8	10	15	6	9	16	1	7	3	11	14	2	5	12	13
После слияния элементов 7-8:	4	8	10	15	6	9	1	16	7	3	11	14	2	5	12	13
После слияния элементов 5-8:	4	8	10	15	1	6	9	16	7	3	11	14	2	5	12	13
После слияния элементов 1-8:	1	4	6	8	9	10	15	16	7	3	11	14	2	5	12	13
После слияния элементов 9-10:	1	4	6	8	9	10	15	16	3	7	11	14	2	5	12	13
После слияния элементов 11-12:	1	4	6	8	9	10	15	16	3	7	11	14	2	5	12	13
После слияния элементов 9-12:	1	4	6	8	9	10	15	16	3	7	11	14	2	5	12	13
После слияния элементов 13-14:	1	4	6	8	9	10	15	16	3	7	11	14	2	5	12	13
После слияния элементов 15-16:	1	4	6	8	9	10	15	16	3	7	11	14	2	5	12	13
После слияния элементов 13-16:	1	4	6	8	9	10	15	16	3	7	,11	14	2	5	12	13
После слияния элементов 9-16:	1	4	6	8	9	10	15	16	2	3	5	7	11	12	13	14
После слияния элементов 1-16:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Быстрая сортировка

Исходный список:	6	2	4	7	1	3	8	5
Ось в ячейке 6:	5	2	4	1	3	6	8	7
Ось в ячейке 5:	3	2	4	1	5	6	8	7
Ось в ячейке 3:	1	2	3	4	5	6	8	7
Ось в ячейке 1:	1	2	3	4	5	6	8	7
Ось в ячейке 8:	1	2	3	4	5	6	7	8

Исходный список:	15	4	10	8	6	9	16	1	7	3	11	14	2	5	12	13
Ось в ячейки 15:	13	4	10	8	6	9	1	7	3	11	14	2	5	12	15	16
Ось в ячейке 13:	12	4	10	8	6	9	1	7	3	11	2	5	13	14	15	16
Ось в ячейке 12:	5	4	10	8	6	9	1	7	3	ill	2	12	13	14	15	16

Ось в ячейке 5:	2	4	1	3	5	9	10	7	8	11	6	12	13	14	15	16
Ось в ячейке 2:	1	2	4	3	5	9	10	7	8	11	6	12	13	14	15	16
Ось в ячейке 4:	1	2	3	4	5	9	10	7	8	11	6	12	13	14	15	16
Ось в ячейке 9:	1	2	3	4	5	6	7	8	9	11	10	12	13	14	15	16
Ось в ячейке 6:	1	2	3	4	5	6	7	8	9	11	10	12	13	14	15	16
Ось в ячейке 7:	1	2	3	4	5	6	7	8	9	11	10	12	13	14	15	16
Ось в ячейке 11:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Корневая сортировка

Исходный список:	1113	2231	3232	1211	3133	2123	2321	1312	3223	2332	1121	3312
После прохода 1:	2231	1211	2321	1121	3232	1312	2332	3312	1113	3133	2123	3223
После прохода 2:	1211	1312	3312	1113	2321	1121	2123	3223	2231	3232	2332	3133
После прохода 3:	1113	1121	2123	3133	1211	3223	2231	3232	1312	3312	2321	2332
После прохода 4:	1113	1121	1211	1312	2123	2231	2321	2332	3133	3223	3232	3312

Алгоритмы из главы 4

Исходный многочлен $x^3 + 4x^2 - 3x + 2$

Схема Горнера $((x + 4)x - 3)x + 2$

Предварительная обработка коэффициентов

$$x^3 + 4x^2 - 3x + 2 = ((x^2 - 4)(x + 4)) + (x + 18)$$

Исходный многочлен $x^7 - 8x^6 + 3x^5 + 2x^4 - 4x^3 + 5x - 7$

Схема Горнера $(((((x - 8)x + 3)x + 2)x - 4)x + 0)x + 5)x - 7$

Предварительная обработка коэффициентов

$$\begin{aligned} x^7 - 8x^6 + 3x^5 + 2x^4 - 4x^3 + 5x - 7 = \\ (x^4 - 5)(x^3 - 8x^2 + 3x + 2) + (x^3 - 40x^2 + 20x + 3) \\ = ((x^4 - 5)(((x^2 + 2)(x - 8)) + (x + 18))) \\ + (((x^2 + 19)(x - 40)) + (x + 757)) \end{aligned}$$

Стандартное умножение матриц

$$\begin{bmatrix} 1 & 4 \\ 5 & 8 \end{bmatrix} \begin{bmatrix} 6 & 7 \\ 3 & 2 \end{bmatrix} = \begin{bmatrix} 1 \cdot 6 + 4 \cdot 3 & 1 \cdot 7 + 4 \cdot 2 \\ 5 \cdot 6 + 8 \cdot 3 & 5 \cdot 7 + 8 \cdot 2 \end{bmatrix} = \begin{bmatrix} 18 & 15 \\ 54 & 51 \end{bmatrix}$$

Алгоритм Винограда умножения матриц

Множители по строчкам 4 40
Множители по столбцам 18 14

Алгоритм Штрассена

$x_1 = (G_{1,1} + G_{2,2})(H_{1,1} + H_{2,2}) = (1 + 8) \cdot (6 + 2) = 9 \cdot 8 = 72$
$X2 = (G_{2,1} + G_{2,2})H_{1,1} = (5 + 8) \cdot 6 = 13 \cdot 6 = 78$
$x_3 = G_{1,1}(H_{1,2} - H_{2,2}) = 1 \cdot (7 - 2) = 1 \cdot 5 = 5$
$x_4 = G_{2,2}(H_{2,1} - H_{1,1}) = 8 \cdot (3 - 6) = 8 \cdot (-3) = -24$
$x_5 = (G_{1,1} + G_{1,2})H_{2,2} = (1 + 4) \cdot 2 = 5 \cdot 2 = 10$
$x_6 = (G_{2,1} - G_{1,1})(H_{1,1} + H_{1,2}) = (5 - 1) \cdot (6 + 7) = 4 \cdot 13 = 52$
$x_7 = (G_{1,2} - G_{2,2})(H_{2,1} + H_{2,2}) = (4 - 8) \cdot (3 + 2) = (-4) \cdot 5 = -20$
$R_{1,1} = x_1 + x_4 - x_5 + x_7 = 72 + (-24) - 10 + (-20) = 18$
$\#2,1 = x_2 + x_4 = 78 + (-24) = 54$
$\#1,2 = x_3 + x_5 = 5 + 10 = 15$
$\#2,2 = x_1 + x_3 - x_2 + x_6 = 72 + 5 - 78 + 52 = 51$

Метод Гаусса-Жордана

$$\begin{bmatrix} 3 & 9 & 6 & 21 \\ 5 & 3 & 22 & 23 \\ 2 & 5 & 7 & 26 \end{bmatrix} \begin{bmatrix} 1 & 3 & 2 & 7 \\ 5 & 3 & 22 & 23 \\ 2 & 5 & 7 & 26 \end{bmatrix} \begin{bmatrix} 1 & 3 & 2 & 7 \\ 0 & -12 & 12 & -12 \\ 0 & 2 & 3 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 3 & 2 & 7 \\ 0 & 1 & -1 & 1 \\ 0 & 2 & 3 & 12 \end{bmatrix} \begin{bmatrix} 1 & 0 & 5 & 4 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 5 & 10 \end{bmatrix} \begin{bmatrix} 1 & 0 & 5 & 4 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & -6 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

Алгоритмы из главы 5

Алгоритм Кнута-Морриса-Пратта Образец имеет вид

abcabc

Массив неудачных сравнений содержит значения

0 1 1 1 2 3

Алгоритм Бойера-Мура

Образец имеет вид

abcbccabc

После завершения первого цикла массив прыжков содержит

17 16 15 14 13 12 11 10 9

После завершения второго цикла массив прыжков содержит

17 16 15 14 13 12 6 4 1

После завершения второго цикла массив связей содержит

7 8 7 8 8 9 9 9 10

После завершения третьего цикла массив прыжков содержит

14 13 12 11 10 9 6 4 1

После завершения четвертого цикла массив прыжков содержит

14 13 12 11 10 9 6 4 1

Алгоритмы из главы 6

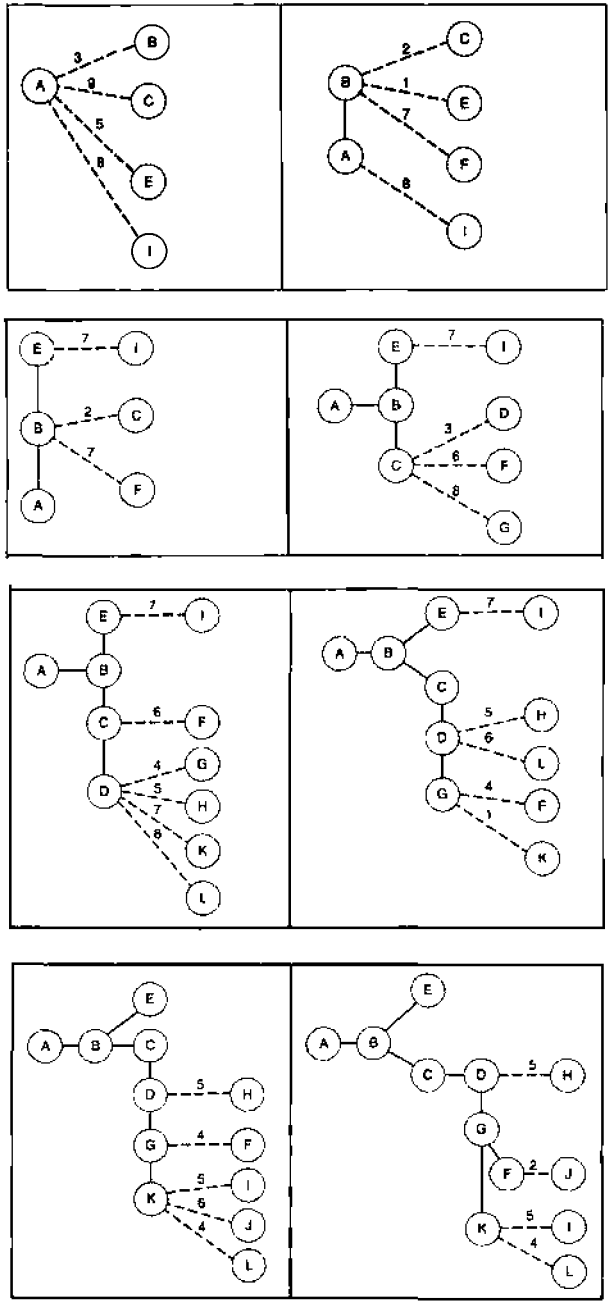
Обход в глубину, начиная с вершины A

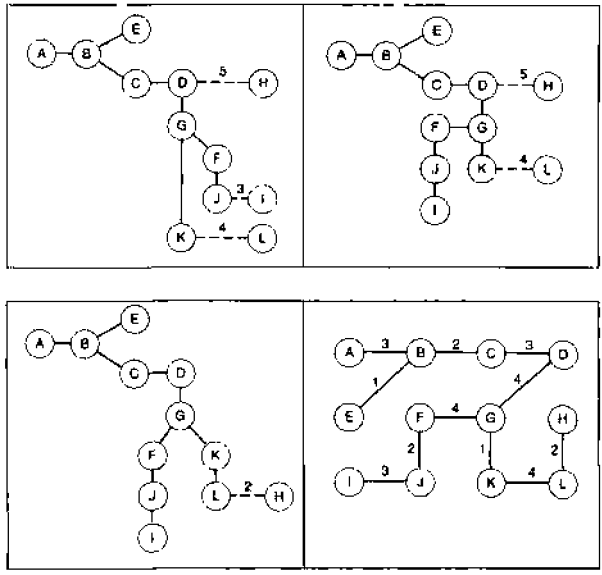
A, B, G, C, D, H, L, K, J, F, I, E

Обход по уровням, начиная с вершины A

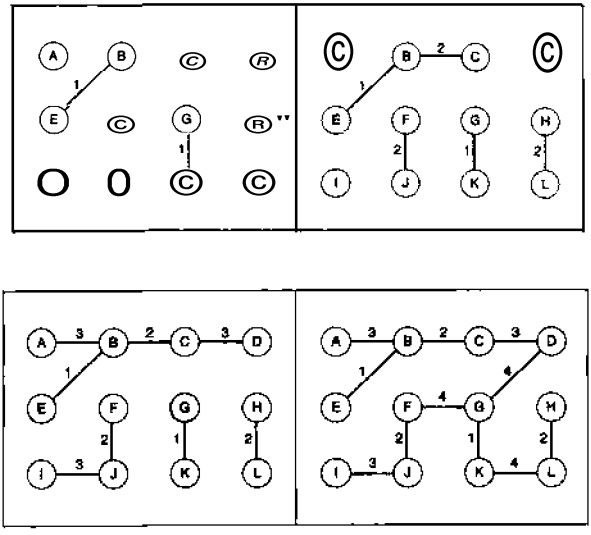
A, B, E, F, G, *I*, C, L, D, H, K, J

Трассировка построения минимального остовного дерева алгоритмом **Дейкстры-Прима**

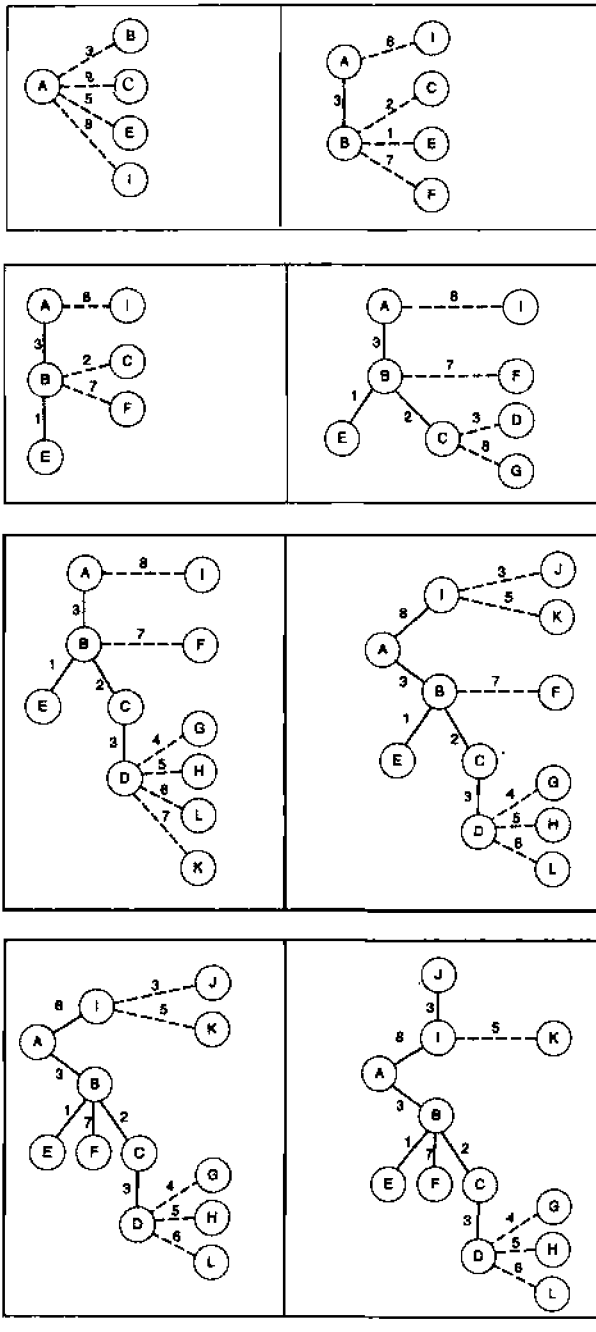


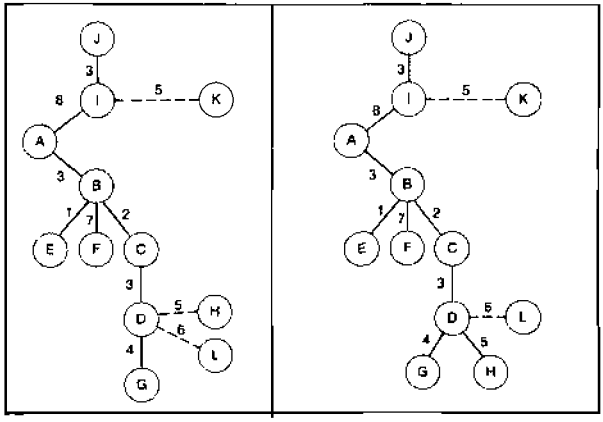


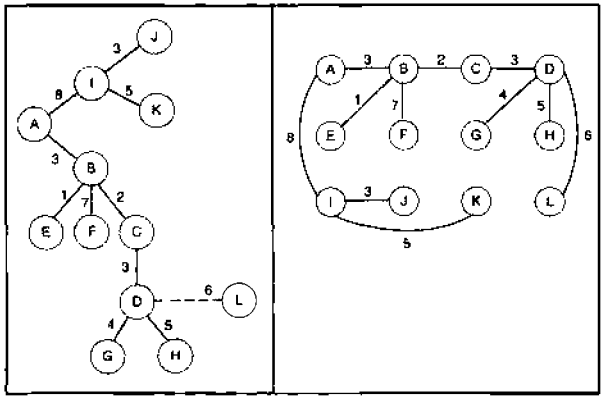
Трассировка построения минимального остовного дерева алгоритмом Крускала



Алгоритм Деикстры построения кратчайшего пути




J
3
I
5
K
8
A
3
B
1
7
2
E
F
C
3
D
5
H
6
L
4
G
J
3
I
5
K
8
A
3
B
1
7
2
E
F
C
3
D
5
L
4
5
G
H


J
3
I
5
K
8
A
3
B
1
7
2
E
F
C
3
D
6
L
4
5
G
H
A
3
B
2
C
3
D
1
7
4
5
8
E
F
G
H
6
I
3
J
K
L
5

Литература

К главе 1. Основы анализа алгоритмов

A. Aho, J. Hopcroft, J. Ullman *The Design and Analyis of Computer Algorithms*, Addison–Wesley, Reading, MA (1971) [Русский перевод: А. Ахо, Дж. Хопкрофт, Дж. Ульман *Построение и анализ вычислительных алгоритмов,* М., Мир (1979)]

S. Baase, A. Van Gelder *Computer Algorithms,* Addison-Wesley Longman, Reading, MA (2000)

T. Cormen, C. Leiserson, R. Rivest *Introduction to Algorithms*, McGraw-Hill, New York 1990)

D. Knuth *Big Omikron and Big Omega and Big theta*, SIGACT News, 8 (2), 18-24 (1976)

I. Parberry *Problems on Algorithms,* Prentice Hall, Englewood Cliffs, NJ (1995)

P. Purdom, C. Brown *The Analysis of Algorithms,* Holt, Rinehart and Winston, New York (1985)

R. Sedgewick *Algotihms,* 2d ed., Addison-Wesley, Reading, MA (1988)

К главе 2. Алгоритмы поиска и выборки

D. E. Knuth *The Art of Computer Programming: Volume 3 Sorting and Searching,* 2d ed., Addison-Wesley, Reading, MA (1998) [Русский перевод первого издания: Д. Кнут *Искусство программирования для ЭВМ,* т. 3, М., Мир (1978)]

К главе 3. Алгоритмы сортировки

C. A. R. Hoare *Quicksort,* Computer Journal, 5 (1), 10-15 (1962)

D. E. Knuth *The Art of Computer Programming: Volume 3 Sorting and Searching,* 2d ed., Addison-Wesley, Reading, MA (1998) [Русский перевод первого издания: Д. Кнут *Искусство программирования для ЭВМ,* т. 3, М., Мир (1978)]

D. L. Shell *A High-Speed Sorting Procedure,* Communications of the ACM, 2 (7), 30-32 (1959)

J. Williams *Algorithm 232: Heapsort,* Communications of the ACM, 7 (6), 347-348 (1964)

К главе 4. Численные алгоритмы

V. Strassen *Gaussian Elimination is Not Optimal,* Numerische matematik, **13,** 354-356 (1969)

S. Winograd *On the Number of Multiplications Necessary to Compute Certain Functions,* Journal of Pure and Applied Mathematics, 165-179 (1970)

К главе 5. Алгоритмы сравнения с образцом

A. Aho, M. Corasick *Efficient String Matching: An Aid to Bibliographic Search,* Communications of the ACM, **18** (6), 333-340 (1975)

R. Boyer, J, Moore *A Fast String Searching Algorithm,* Communications of the ACM, **20** (10), 762-772 (1977)

M. Crochemore, W. Rytter *Text Algorithms,* Oxford University Press, New York (1994)

P. Hall, G. Dowling *Approximate String Matching,* Computing Surveys, **12** (4), 381-402 (1980)

D. Knuth, J. Morris, V. Pratt *Fast Pattern matching in Strings,* Journal on Computing, 6 (2), 323-350 (1977)

G. Landau, U. Vishkin *Introducing Efficient Parallelism into Approximate String matching and a New serial Algorithm,* in: Proceedings of the 18 th Annual ACM Symposium on Theory of Computing, 220-230 (1986)

G. Smith *A Comparison of Three String Matching Algorithms,* Software -– Practice and Experience, **12,** 57-66 (1982)

К главе 6. Алгоритмы на графах

E. Dijkstra *A Note on Two Problems in Connexion with Graphs,* Numerische Mathematik, 1, 269-271 (1959)

S. Even *Graph Algorithms,* Computer Science Press, Rockville, MD (1979)

A. Gibbons *Algorithmic Graph Theory*, Cambridge University Press, Cambridge, England (1985)

J. Hopcroft, R. Tarjan *Dividing a Graph into Triconnected Components,* SIAM Journal on Computing, 2 (3), 135-157 (1973)

S. Khuller, B. Raghavachari *Basic Graph Algorithms,* in: M. Atallah (ed.) Algorithms and Theory of Computation Handbook, CRC Press, New York, 6-1-6-23 (1999)

R. Prim *Shortest Connection Networks and Some Generalizations,* Bell System Technical Journal, **36,** 1389-1401 (1957)

R. Tarjan *Depth-First Search and Linear Graph Algorithms,* SIAM Journal on Computing, 1 (2), 146-160 (1972)

К главе 7. Параллельные алгоритмы

S. Akl *Parallel Sorting,* Academic Press, Orlando, FL (1985)

S. Akl *The Design and Analysis of Parallel Algorithms,* Prentice Hall, Englewood Cliffs, NJ (1989)

S. Fortune, J. Wyllie *Parallelism in Random Access Machines,* in: Proceedings of the 10 th Annual ACM Symposium on Theory of Computing, 114-118 (1978)

L. Goldschlager *A Unified Approach to Models of Synchronous Parallel Machines,* in: Proceedings of the 10 th Annual ACM Symposium on Theory of Computing, 89-94 (1978)

R. Greenlaw, H. Hoover, W. Ruzzo *Limits to Parallel Computation: P-Completeness Theory,* Oxford University Press, New York (1995)

R. Karp, V. Ramachandran *A Survey of Parallel Algorithms and Shared Memory Machines,* in A. vanLeeuwen (ed.), Handbook of Theoretical Computer Science: Algorithms and Complexity, 869–941, Elsevier, New York (1990)

C. Kruskal *Searching, Merging, and Sorting in Parallel Computation,* IEEE Transactions on Computers, **32** (10), 942-946 (1983)

F. Leighton *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann Publishers, San Mateo, CA (1992)

R. Miller, L. Boxer *A Unified Approach to Sequential and Parallel Algorithms*, Prentice Hall Inc., Upper Saddle River, NJ (2000)

R. Miller, Q. Stout *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, The MIT Press, Cambridge, MA (1996)

M. Quinn, N. Deo *Parallel Graph Algorithms*, ACM Computing Serveys, **16** (3), 319-348 (1984)

Y. Shiloach, U. Vishkin *Finding the Maximum, Merging and Sorting in a Parallel Computation Model*, Journal of Algorithms, 2, 88-102 (1981)

К главе 8. Недетерминированные алгоритмы

S. Cook *The Complexity of Theory Proving Procedures,* in: Proceedings of the 3d Annual ACM Symposium on Theory of Computing, 151-158 (1971)

M. Garey, D. Johnson *Computers and Intractability: A Guide to the Theory of NP-Competleness*, W. H. Freeman, San Francisco, CA (1979)

R. Karp *Reducibility Among Combinatorial Problems,* in: R. Miller, J. Thatcher (eds.), Complexity of Computer Computations, Plenum Press, New York, 85-104 (1972)

E. Lawler, J. Lenstra, A. Kan, D. Schmoys (eds.) *The Travelling Salesman Problem,* Wiley, New York (1985)

К главе 9. Другие алгоритмические инструменты

R. Bellman *Dynamic Programming,* Princeton University Press, Princeton, NJ (1957)

R. Bellman, S. Dreyfus *Applied Dynamic Programming,* Princeton University Press, Princeton, NJ (1962)

J. Bentley, D. Johnson, F. Leighton, C. McGeoch, *An Experimental Study of Bin Packing,* in: Proceedings of the 21st Annual Allerton Conference on Communication, Control and Computing, 51-60 (1983)

G. Brassard, P. Bratley *Algorithms: Theory and Practice,* Prentice Hall, Englewood Cliffs, NJ (1988)

M. Garey, R. Graham, J. Ullman *Worst-Case Analysis of Memory Allocation Algorithms,* in: Proceedings of the 4th Annual ACM Symposium on Theory of Computing, 143-150 (1976)

M. Garey, D. Johnson *The Complexity of Near-Optimal Graph Coloring*, Journal of the ACM, **23** (10), 43-49 (1976)

A. Hall *On an experimental determination of* π, Messenger of mathematics, 2, 113-114 (1973)

D. Hochbaum (ed.) *Approximation algorithms for NP-Hard Problems*, PWS Publishing, Boston, MA (1997)

D. Johnson *Fast Allocation Algorithms,* in: Proceedings of the 13th Annual Symposium on Switching and Automata Theory, 144-154 (1972)

D. Johnson *Approximation Algorithms for Combinatorial Problems,* in: Proceedings of the 5th Annual ACM Symposium on Theory of Computing, 38-49 (1973)

D. Johnson *Worst-Case Behavior of Graph Coloring Algorithms,* in: Proceedings of the 5th Southeastern Conference on Combinatorics, Graph Theory, and Computing, Utilitas Mathematica Publishing, Winnipeg, Canada 513-528 (1974)

E. Lawler, J. Lenstra, A. Kan, D. Schmoys (eds.) *The Travelling Salesman Problem,* Wiley, New York (1985)

G. Leclerc *Essai d'arithmétique morale* (1777)

I. Metropolis, S. Ulam *The Monte Carlo Method,* Journal of the American Statistical Association, **44** (247), 335-341 (1949)

S. Sahni *Approximate Algorithms for the 0/1 Knapsack Problem*, Journal of the ACM, **22** (10), 115-124 (1975)

S. Sahni *Algorithms for Scheduling Independent Tasks,* Journal of the ACM, **23** (1), 116-127 (1976) A. Wigderson *Improving the Performance Guarantee for Approximate Graph Coloring,* Journal of the ACM, **30** (4), 729-735 (1983)

F. Yao *Speed-Up in Dynamic Programming,* SIAM Journal on Algebraic and Discrete Methods, 3 (4), 532-540 (1982)

Дополнение

М.В. Ульянов

ПРЕДИСЛОВИЕ

Опубликованное в 2002 году издательством «Техносфера» в серии «Мир программирования» первое издание книги Дж. Макконелла «Анализ алгоритмов. Вводный курс» представляло собой развернутое введение в анализ алгоритмов и знакомило читателей на простых и доступных примерах с основными понятиями в области анализа вычислительных алгоритмов. По своей направленности книга является методически грамотным учебным пособием для студентов и программистов, всех тех, кто интересуется вопросами анализа алгоритмов.

Однако содержание теории алгоритмов значительно шире, чем только область оценки вычислительной сложности алгоритмов, поэтому представляется логичным расширить издание за счет включения в него дополнительных разделов, посвященных элементам теории алгоритмов и оценкам трудоемкости. Кроме того, несомненно важно осветить некоторые новейшие направления в области разработки алгоритмов.

Дополнение к переизданию этой книги ставит своей задачей, сохраняя ее целевую аудиторию, предоставить читателю дополнительную информацию к вводному курсу Дж. Макконелла. Следует иметь в виду, что при систематическом изучении предмета изложение элементов теории алгоритмов должно предшествовать курсу анализа алгоритмов.

Материал дополнения тесно связан с лекционными курсами Московской государственной академии приборостроения и информатики.

Глава Д.1.
Элементы теории алгоритмов

Д.1.1. Введение в теорию алгоритмов

На основе формализации понятия алгоритма возможно сравнение алгоритмов по их эффективности, проверка их правильности и эквивалентности, определение областей применимости.

Первым дошедшим до нас алгоритмом в его интуитивном понимании как конечной последовательности элементарных действий, решающих поставленную задачу, считается предложенный Евклидом в III веке до нашей эры алгоритм нахождения наибольшего общего делителя двух чисел (алгоритм Евклида, см. описание в пункте Д.3.1.5. настоящего дополнения). Отметим, что в течение длительного времени, вплоть до начала XX века, само слово «алгоритм» употреблялось в устойчивом сочетании «алгоритм Евклида». Для описания последовательности пошагового решения других математических задач чаще использовался термин «метод». Обсуждение тонкой терминологической разницы выходит за рамки этого дополнения.

Начальной точкой отсчета современной теории алгоритмов можно считать теорему о неполноте символических логик, доказанную немецким математиком Куртом Геделем в 1931 году ([1]). В этой работе было показано, что некоторые математические проблемы не могут быть решены алгоритмами из определенного класса. Общность результата Геделя связана с вопросом о том, совпадает ли использованный им класс алгоритмов с классом всех алгоритмов в интуитивном понимании этого термина. Эта работа дала толчок к поиску и анализу различных формализации понятия «алгоритм».

Первые фундаментальные работы по теории алгоритмов были опубликованы в середине 1930-х годов Аланом Тьюрингом, Алоизом Черчем и Эмилем Постом. Предложенные ими машина Тьюринга, машина Поста и класс рекурсивно-вычислимых функций Черча были первыми формальными описаниями алгоритма, использующими строго определенные модели вычислений. Сформулированные гипотезы Поста и Черча - Тьюринга постулировали эквивалентность предложенных ими моделей вычислений, как формальных систем, и интуитивного понятия алгоритма. Важным развитием этих работ стала формулировка и доказательство существования алгоритмически неразрешимых проблем.

В 1950-е годы существенный вклад в развитие теории алгоритмов внесли работы Колмогорова и, основанный на теории формальных грамматик, алгоритмический формализм Маркова, так называемые нормальные алгоритмы Маркова. Формальные модели алгоритмов Поста, Тьюринга и Черча, равно как и модели Колмогорова и Маркова, оказались эквивалентными в том смысле, что любой класс проблем, разрешимых в одной модели, разрешим и в другой.

Появление доступных ЭВМ и существенное расширение круга решаемых на них задач привели в 1960-ых -70-ых годах к практически значимым исследованиям алгоритмов и вычислительных задач. На этой основе в данный период оформились

следующие разделы в теории алгоритмов:

- классическая теория алгоритмов (формулировка задач в терминах формальных языков, понятие задачи разрешения, описание сложностных классов задач, формулировка в 1965 году Эдмондсом проблемы $P = NP$, открытие класса NP-полных задач и его исследование; введение новых моделей вычислений — алгебраического дерева вычислений (Бен-Ор), машины с произвольным доступом к памяти, схем алгоритмов Янова, стандартных схем программ Котова и др. [2, 3]);
- теория асимптотического анализа алгоритмов (понятие сложности и трудоемкости алгоритма, критерии оценки алгоритмов, методы получения асимптотических оценок, в частности для рекурсивных алгоритмов, асимптотический анализ трудоемкости или времени выполнения, получение теоретических нижних оценок сложности задач), в развитие которой внесли существенный вклад Кнут, Ахо, Хопкрофт, Ульман, Карп, Мошков, Кудрявцев и др. ([3, 4, 5, 6, 7]);
- теория практического анализа вычислительных алгоритмов (получение явных функции трудоемкости, интервальный анализ функций, практически значимые критерии качества алгоритмов, методики выбора рациональных алгоритмов), основополагающей работой в этом направлении, очевидно, следует считать фундаментальный труд Д. Кнута «Искусство программирования для ЭВМ» ([4]).

К теории алгоритмов тесно примыкают исследования, связанные с разработкой методов создания эффективных алгоритмов (динамическое программирование, метод ветвей и границ, метод декомпозиции, «жадные» алгоритмы, специальные структуры данных и т. д.) ([2, 6]).

В основном тексте книги Макконелла освещены с различной степенью детальности элементы асимптотического анализа алгоритмов, включая решения рекуррентных соотношений, дано введение в сложностные классы задач с точки зрения недетерминированных алгоритмов и описаны наиболее распространенные методы разработки эффективных алгоритмов.

Обобщая исследования в различных разделах теории алгоритмов, можно выделить следующие основные задачи и направления развития, характерные для современной теории алгоритмов:

- формализация понятия «алгоритм» и исследование формальных алгоритмических систем (моделей вычислений);
- доказательство алгоритмической неразрешимости задач;
- формальное доказательство правильности и эквивалентности алгоритмов;
- классификации задач, определение и исследование сложностных классов;
- доказательство теоретических нижних оценок сложности задач;
- получение методов разработки эффективных алгоритмов;
- асимптотический анализ сложности итерационных алгоритмов;
- исследование и анализ рекурсивных алгоритмов;
- получение явных функций трудоемкости алгоритмов;
- разработка классификаций алгоритмов;

- исследование емкостной (по ресурсу памяти) сложности задач и алгоритмов;
- разработка критериев сравнительной оценки ресурсной эффективности алгоритмов и методов их сравнительного анализа.

Полученные в теории алгоритмов результаты находят сегодня достаточно широкое практическое применение, в рамках которого можно выделить два следующих аспекта:

Теоретический аспект: при исследовании некоторой задачи результаты теории алгоритмов позволяют ответить на вопрос - является ли эта задача в принципе алгоритмически разрешимой. Для алгоритмически неразрешимых задач возможно их сведение к задаче останова машины Тьюринга. В случае алгоритмической разрешимости задачи следующим важным теоретическим вопросом является вопрос о принадлежности этой задачи к классу NP-полных задач. При утвердительном ответе можно говорить о существенных временных затратах для получения точного решения этой задачи для больших размерностей исходных данных, иными словами — об отсутствии быстрого точного алгоритма ее решения.

Практический аспект: методы и методики теории алгоритмов, в основном разделов асимптотического и практического анализа, позволяют осуществить:

- рациональный выбор из известного множества алгоритмов решения данной задачи, учитывающий особенности их применения в разрабатываемой программной системе. Например, в условиях ограничений на время выполнения программной реализации алгоритма, и в условиях критерия минимального объема дополнительно используемой алгоритмом памяти, выбор, скорее всего, будет сделан в пользу различных алгоритмов;
- получение временных оценок решения сложных задач на основе функции трудоемкости;
- получение достоверных оценок невозможности решения некоторой задачи за определенное время. Такого рода обратные по временной эффективности задачи оказываются сегодня также востребованы, например, для криптографических методов;
- разработку и совершенствование эффективных алгоритмов решения практически значимых задач в области обработки информации.

Д. 1.2. Формализация понятия алгоритма

Во всех сферах своей деятельности, в частности в сфере обработки информации, человек сталкивается с различными способами или методиками решения разнообразных задач. Они определяют порядок выполнения действий для получения желаемого результата — мы можем трактовать это как первоначальное или интуитивное определение алгоритма. Таким образом, можно нестрого определить алгоритм как однозначно трактуемую процедуру решения задачи. Дополнительные требования о выполнении алгоритма за конечное время для любых входных данных приводят к следующему неформальному определению алгоритма:

Алгоритм — это заданное на некотором языке конечное предписание, задающее конечную последовательность выполнимых и точно определенных элементарных операций для решения задачи, общее для класса возможных исходных данных.

Пусть DZ — область (множество) исходных данных задачи Z, а R — множество возможных результатов, тогда мы можем говорить, что алгоритм осуществляет отображение $DZ \rightarrow R$. Поскольку такое отображение может быть не полным, то в теории алгоритмов вводятся следующие понятия:

Алгоритм называется *частичным* алгоритмом, если мы получаем результат только для некоторых d 6 DZ и *полным* алгоритмом, если алгоритм получает правильный результат для всех d € D_Z.

Несмотря на усилия ученых, сегодня отсутствует одно исчерпывающе строгое определение понятия «алгоритм». Из разнообразных вариантов словесного определения алгоритма наиболее удачные, по мнению автора, принадлежат российским ученым А. Н. Колмогорову и А. А. Маркову ([8]):

Определение 1.1. (Колмогоров): *Алгоритм* — это всякая система вычислений, выполняемых по строго определенным правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи.

Определение 1.2. (Марков): *Алгоритм* — это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату.

Отметим, что различные определения алгоритма, в явной или неявной форме, постулируют следующий ряд общих требований:

- алгоритм должен содержать конечное количество элементарно выполнимых предписаний, т. е. удовлетворять требованию конечности записи;
- алгоритм должен выполнять конечное количество шагов при решении задачи, т. е. удовлетворять требованию конечности действий;
- алгоритм должен быть единым для всех допустимых исходных данных, т. е. удовлетворять требованию универсальности;
- алгоритм должен приводить к правильному по отношению к поставленной задаче решению, т. е. удовлетворять требованию правильности.

Неудобства словесных определений связаны с проблемой однозначной трактовки терминов. В таких определениях должен быть, хотя бы неявно, указан исполнитель действий или предписаний. Алгоритм вычисления производной для полинома фиксированной степени вполне ясен тем, кто знаком с основами математического анализа, но для прочих он может оказаться совершенно непонятным. Это рассуждение заставляет нас указать так же вычислительные возможности исполнителя, а именно уточнить какие операции для него являются «элементарными». Другие трудности связаны с тем, что алгоритм заведомо существует, но его очень трудно описать в некоторой заранее заданной форме. Классический пример такой ситуации — алгоритм завязывания шнурков на ботинках «в бантик». Вы сможете дать только словесное описание этого алгоритма без использования иллюстраций ?

В связи с этим формально строгие определения понятия алгоритма связаны с введением специальных математических конструкций — формальных алгоритмических систем или моделей вычислений, каковыми являются машина Поста, машина Тьюринга, рекурсивно-вычислимые функции Черча, и постулированием тезиса об

эквивалентности такого формализма и понятия «алгоритм». Несмотря на принципиально разные модели вычислений, использующиеся в теории алгоритмов для определения термина «алгоритм», интересным результатом является формулировка гипотез о эквивалентности этих формальных определений в смысле их равномощности.

Д. 1.3. Машина Поста

Одной из фундаментальных статей, результаты которой лежат в основе современной теории алгоритмов, является статья Эмиля Поста (Emil Post), «Финитные комбинаторные процессы, формулировка 1», опубликованная в 1936 году в сентябрьском номере «Журнала символической логики» ([8]). Пост рассматривает общую проблему, состоящую из множества конкретных проблем, при этом решением общей проблемы является такое решение, которое доставляет ответ для каждой конкретной проблемы.

Например, решение уравнения $3 \cdot x + 9 = 0$ — это одна из конкретных проблем, а решение уравнения $a \cdot x + b = 0$ — это общая проблема, тем самым алгоритм должен быть универсальным, т. е. должен быть соотнесен с общей проблемой. Отметим, что сам термин «алгоритм» не используется Э. Постом, его заменяет в статье понятие набора инструкций.

Основные понятия алгоритмического формализма Поста — это пространство символов (язык *L)* в котором задается конкретная проблема и получается ответ, и набор инструкций, т. е. операций в пространстве символов, задающих как сами операции, так и порядок их выполнения. Постовское пространство символов представляет собой бесконечную ленту ящиков (ячеек), каждый из которых может быть или помечен, или не помечен.

Конкретная проблема задается «внешней силой» (термин Поста) пометкой конечного количества ячеек, при этом очевидно, что любая конфигурация начинается и заканчивается помеченным ящиком. После применения к конкретной проблеме некоторого набора инструкций, решение представляется так же в виде набора помеченных и непомеченных ящиков, распознаваемое той же внешней силой.

Пост предложил набор инструкций или элементарно выполнимых операций, которые выполняет «работник», отметим в этой связи, что в 1936 году не было еще ни одной работающей электронной вычислительной машины. Сегодня мы бы сказали, что этот набор инструкций является минимальным набором битовых операций элементарного процессора:

1. пометить ящик, если он пуст;
2. стереть метку, если она есть;
3. переместиться влево на 1 ящик;
4. переместиться вправо на 1 ящик;
5. определить помечен ящик или нет, и по результату перейти на одну из двух указанных инструкций;
6. остановиться.

Отметим, что формулировка инструкций 1 и 2 включает в себя защиту от неправильных ситуаций с данными. Программа представляет собой нумерованную последовательность инструкций, причем переходы в инструкции 5 производятся на указанные в ней номера других инструкций. Программа, или набор инструкций в терминах Э. Поста, является одной и той же для всех конкретных проблем, и поэтому соотнесена с общей проблемой — таким образом, Пост формулирует требование универсальности.

Далее Э. Пост вводит следующие понятия:

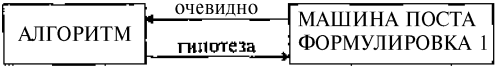
- набор инструкций *применим к* общей проблеме, если для каждой конкретной проблемы не возникает коллизий в инструкциях 1 и 2, т. е. никогда программа не стирает метку в пустом ящике и не устанавливает метку в помеченном ящике;
- набор инструкций *заканчивается,* если после конечного количества инструкций выполняется инструкция 6;
- набор инструкций задает *финитный 1-процесс,* если набор применим, и заканчивается для каждой конкретной проблемы;
- финитный 1-процесс для общей проблемы есть 1-*решение*, если ответ для каждой конкретной проблемы правильный, что определяется внешней силой, задающей конкретную проблему.

По Посту, проблема задается внешней силой путем пометки конечного количества ящиков ленты. В более поздних работах по машине Поста ([8]) принято считать, что машина работает в единичной системе счисления (0 = *V;* 1 = *VV*; *2 = VVV),* т. е. ноль представляется одним помеченным ящиком, а целое положительное число — помеченными ящиками в количестве на единицу больше его значения. Поскольку множество конкретных проблем, составляющих общую проблему, является счетным, то можно установить взаимно однозначное соответствие (биективное отображение) между множеством положительных целых чисел N и множеством конкретных проблем. Общая проблема называется по Посту 1-*заданной*, если существует такой финитный 1-процесс, что, будучи, применим к $n \in N$ в качестве исходной конфигурации ящиков, он задает n-ую конкретную проблему в виде набора помеченных ящиков в пространстве символов.

Если общая проблема 1-задана и 1-разрешима, то, соединяя наборы инструкций по заданию проблемы и ее решению, мы получаем ответ по номеру проблемы — это и есть *формулировка* 1 в терминах статьи Э. Поста.

Э. Пост завершает свою статью следующей фразой ([8]): «Автор ожидает, что его формулировка окажется логически эквивалентной рекурсивности в смысле Геделя–Черча. Цель формулировки, однако, в том, чтобы предложить систему не только определенной логической силы, но и психологической достоверности. В этом последнем смысле подлежат рассмотрению все более и более широкие формулировки. С другой стороны, нашей целью будет показать, что все они логически сводимы к формулировке 1. В настоящий момент мы выдвигаем это умозаключение в качестве *рабочей гипотезы.* ...Успех вышеизложенной программы заключался бы для нас в превращении этой гипотезы не столько в определение или аксиому, сколько в закон природы».

Таким образом, гипотеза Поста состоит в том, что любые более широкие формулировки в смысле алфавита символов ленты, набора инструкций, представления и интерпретации конкретных проблем сводимы к формулировке 1.



Следовательно, если гипотеза верна, то любые другие формальные определения, задающие некоторый класс алгоритмов, эквивалентны классу алгоритмов, заданных формулировкой 1. «Обоснование этой гипотезы происходит сегодня не путем строго математического доказательства, а на пути эксперимента — действительно, всякий раз, когда нам указывают алгоритм, его можно перевести в форму программы машины Поста, приводящей к тому же результату» ([8]).

Д.1.4. Машина Тьюринга

Алан Тьюринг (Alan Turing) в 1936 году опубликовал в трудах Лондонского математического общества статью «О вычислимых числах в приложении к проблеме разрешения», которая наравне с работами Поста и Черча лежит в основе современной теории алгоритмов ([9]). Предыстория создания этой работы связана с формулировкой Давидом Гильбертом на Международном математическом конгрессе в Париже в 1900 году нерешенных математических проблем. Одной из них была задача доказательства непротиворечивости системы аксиом обычной арифметики, которую Гильберт в дальнейшем уточнил как «проблему разрешимости»— нахождение общего метода для определения выполнимости данного высказывания на языке формальной логики.

Статья Тьюринга как раз и давала ответ на эту проблему — вторая проблема Гильберта оказалась неразрешимой. Но значение статьи Тьюринга выходило далеко за рамки той задачи, по поводу которой она была написана.

Приведем характеристику этой работы, принадлежащую Джону Хопкрофту ([12]): «Работая над проблемой Гильберта, Тьюрингу пришлось дать четкое определение самого понятия метода. Отталкиваясь от интуитивного представления о методе как о некоем алгоритме, т. е. процедуре, которая может быть выполнена механически, без творческого вмешательства, он показал, как эту идею можно воплотить в виде подробной модели вычислительного процесса. Полученная модель вычислений, в которой каждый алгоритм разбивался на последовательность простых, элементарных шагов, и была логической конструкцией, названной впоследствии машиной Тьюринга».

Машина Тьюринга является расширением модели конечного автомата (см. 5.1.1. основного текста), расширением, включающим потенциально бесконечную память с возможностью перехода (движения) от обозреваемой в данный момент ячейки к ее левому или правому соседу ([9]).

Формально машина Тьюринга является расширением конечного автомата, и может быть описана следующим образом — пусть заданы:

- конечное множество состояний Q, в которых может находиться машина Тьюринга;
- конечное множество символов ленты Г;

- функция δ (функция переходов или программа), которая задается отображением пары из декартова произведения Q x Г (машина находится в состоянии q_i и обозревает символ 7») в тройку декартова произведения Q x Г x $\{L, R\}$ (машина переходит в состояние q_j, заменяет символ γ_i на символ γ_j и передвигается влево или вправо на один символ ленты) — $Q \times \Gamma \rightarrow Q$ xΓ x {L, R};
- существует выделенный пустой символ e из Г;
- подмножество S — входной алфавит, Е е Г, определяется как подмножество входных символов ленты, причем символ е € (Г \ Σ);
- одно из состояний q_0 £ Q является начальным состоянием машины.

Решаемая проблема задается путем записи на ленту конечного количества символов *Si,* образующих слово *и* в алфавите S.

e	s_1	s_2	s_3	s_4	...	s_n	e

После задания проблемы машина переводится в начальное состояние, и головка устанавливается у самого левого непустого символа — (до, Т ω), после чего в соответствии с указанной функцией переходов $\delta : (q_i, s_i) \rightarrow (q_j, s_k, Lили\ R)$ машина начинает заменять обозреваемые символы, передвигать головку вправо или влево и переходить в другие состояния. Останов машины происходит в том случае, если для пары *(qi, s_i)* функция перехода не определена.

А. Тьюринг высказал предположение, что любой алгоритм в интуитивном смысле этого слова может быть представлен эквивалентной машиной в предложенной им модели вычислений. Это предположение известно как тезис Черча – Тьюринга. Каждый компьютер может моделировать машину Тьюринга, для этого достаточно моделировать операции перезаписи ячеек, сравнения и перехода к другой соседней ячейке с учетом изменения состояния машины. Таким образом, он может моделировать алгоритмы в машине Тьюринга, и из этого тезиса следует, что все компьютеры, независимо от мощности, архитектуры и других особенностей, эквивалентны с точки зрения принципиальной возможности решения алгоритмически разрешимых задач.

Д. 1.5. Алгоритмически неразрешимые проблемы

За время своего существования человечество придумало множество алгоритмов для решения разнообразных практических и научных проблем. Зададимся вопросом - а существуют ли какие-нибудь проблемы, для которых невозможно придумать алгоритмы их решения? Утверждение о существовании алгоритмически неразрешимых проблем является весьма сильным — мы констатируем, что мы не только сейчас не знаем соответствующего алгоритма, но мы не можем принципиально никогда его найти.

Успехи математики к концу XIX века привели к формированию мнения, которое выразил Д. Гильберт — «в математике не может быть неразрешимых проблем», в связи с этим формулировка Гильбертом нерешенных проблем на конгрессе в Париже в 1900 году была руководством к действию, констатацией отсутствия решений лишь на данный момент.

Первой фундаментальной теоретической работой, связанной с доказательством алгоритмической неразрешимости, была работа К. Геделя — его уже упомянутая теорема о неполноте символических логик. Это строго сформулированная математическая проблема, для которой не существует решающего ее алгоритма. Усилиями различных исследователей список алгоритмически неразрешимых проблем был значительно расширен. Сегодня при доказательстве алгоритмической неразрешимости некоторой задачи принято сводить ее к ставшей классической задаче — «задаче останова» машины Тьюринга.

Имеет место следующая теорема, доказательство которой можно найти, например, в [9].

Теорема Д.1.1. Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных, при этом и алгоритм и данные заданы символами на ленте машины Тьюринга, определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Таким образом, фундаментально алгоритмическая неразрешимость связана с бесконечностью выполняемых алгоритмом действий, иными словами, с невозможностью предсказать, что для любых исходных данных решение будет получено за конечное количество шагов.

Приведем несколько примеров алгоритмически неразрешимых проблем.

Пример Д.1.1. *Распределение девяток в записи числа* π ([1]). Определим функцию $f(n)$ = г, где n — количество девяток подряд в десятичной записи числа π, а г — номер самой левой после запятой девятки из n девяток подряд. Поскольку $\pi \approx 3,141592\ldots$, то $f(1) = 5$. Задача состоит в вычислении функции $f(n)$ для произвольно заданного значения n.

Поскольку число n является иррациональным и трансцендентным, то мы не знаем никакой информации о распределении девяток, равно как и любых других цифр, в десятичной записи числа π. Вычисление $f(n)$ связано с вычислением последующих цифр в разложении числа тг, до тех пор, пока мы не обнаружим n девяток подряд. Однако у нас нет общего метода вычисления $f(n)$, поэтому для некоторых п вычисления могут продолжаться бесконечно — мы даже не знаем в принципе (по природе числа тг) существует ли решение для всех значений п.

Пример Д.1.2. *Вычисление совершенных чисел* ([5]). Совершенные числа — это числа, которые равны сумме своих делителей, например: $28 = 1 + 2 + 4 + 7 + 14$. Определим функцию $S(n)$ = n-ое по счету совершенное число и поставим задачу вычисления $S(n)$ по произвольно заданному п. Нет общего метода вычисления совершенных чисел, мы даже не знаем, конечное или счетное множество совершенных чисел, поэтому наш алгоритм должен перебирать все числа подряд, проверяя их на совершенность. Отсутствие общего метода решения не позволяет ответить на вопрос об останове алгоритма. Если мы проверили M чисел при поиске n-ого совершенного числа — означает ли это, что его вообще не существует?

Пример Д.1.3. *Десятая проблема Гильберта.* Пусть задан многочлен n-ой степени с целыми коэффициентами Р, существует ли алгоритм, который определяет, имеет ли уравнение P — 0 решение в целых числах? Ю. В. Матиясевич ([10]) пока-

зал, что такого алгоритма не существует, т. е. отсутствует общий метод определения целых корней уравнения P — О по его целочисленным коэффициентам.

Пример Д.1.4. *Поиск последнего помеченного ящика в машине Поста* ([1]). Пусть на ленте машины Поста заданы наборы помеченных ящиков (кортежи) произвольной длины с произвольными расстояниями между кортежами, и головка находится у самого левого помеченного ящика. Задача состоит в установке головки на самый правый помеченный ящик последнего кортежа. Попытка построения алгоритма, решающего эту задачу, приводит к необходимости ответа на вопрос — когда после обнаружения конца кортежа мы сдвинулись вправо по пустым ящикам на M позиций и не обнаружили начало следующего кортежа — больше на ленте кортежей нет или они есть где-то правее? Информационная неопределенность задачи состоит в отсутствии информации либо о количестве кортежей на ленте, либо о максимальном расстоянии между кортежами — при наличии такой информации, т.е. при разрешении информационной неопределенности, задача становится алгоритмически разрешимой.

Пример Д.1.5. *Проблема «останова» машины Тьюринга* (см. теорему Д.1.1).

Пример Д.1.6. *Проблема эквивалентности алгоритмов* ([9]). По двум произвольным заданным алгоритмам, например, по двум машинам Тьюринга, определить, будут ли они выдавать одинаковые выходные результаты для любых входных данных.

Пример Д.1.7. *Проблема тотальности* ([9]). По записи произвольно заданного алгоритма определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи выглядит следующим образом: является ли частичный алгоритм A всюду определенным алгоритмом?

Пример Д.1.8. *Проблема соответствий Поста над алфавитом* E. В качестве более подробного примера алгоритмически неразрешимой задачи рассмотрим проблему соответствий, формулировка которой принадлежит Посту ([5]). Мы выделили эту задачу, поскольку на первый взгляд она выглядит достаточно «алгоритмизируемой», но, тем не менее, она сводима к проблеме останова и является алгоритмически неразрешимой.

Постановка задачи: Пусть дан алфавит E: $|E| > 2$ (для одно-символьного алфавита задача имеет решение) и дано конечное множество пар из Σ^+ x Σ^+, т.е. пары непустых цепочек произвольного языка над алфавитом E: $(x_1, y_1), \ldots, (x_m, y_m)$. Проблема соответствий Поста состоит в том, что необходимо выяснить, существует ли конечная последовательность этих пар, не обязательно различных, такая что цепочка, составленная из левых подцепочек, совпадает с последовательностью правых подцепочек — такая последовательность называется решающей.

В качестве примера рассмотрим алфавит E — $\{a, b\}$, и две последовательности входных цепочек.

1. *Входные цепочки: (abbb,b),* (a, aab), (ba, b).

Решающая последовательность для этой задачи имеет вид:

(a, aab), (a, aab), (ba, b), *(abbb,b),* так как: *a a ba abbb* = *aab aab b* Ъ.

2. *Входные цепочки:* (ab, aba), (aba, baa), (baa, aa).

Данная задача вообще не имеет решения, так как нельзя начинать с пары *(aba, baa)* или *(baa, aa)*, поскольку не совпадают начальные символы подцепочек, но если на-

чинать с цепочки *(ab, aba)*, то в последующем не будет совпадать общее количество символов «о», т. к. в других двух парах количество символов «а» одинаково.

В общем случае мы можем построить частичный алгоритм, основанный на идее упорядоченной генерации возможных последовательностей цепочек, отметим, что мы имеем счетное множество таких последовательностей, с проверкой выполнения условий задачи. Если последовательность является решающей, то мы получаем результативный ответ за конечное количество шагов. Поскольку общий метод определения отсутствия решающей последовательности не может быть указан, т. к. задача сводима к проблеме «останова» и, следовательно, является алгоритмически неразрешимой, то при отсутствии решающей последовательности алгоритм порождает бесконечный цикл.

В теории алгоритмов такого рода проблемы, для которых может быть предложен частичный алгоритм их решения, частичный в том смысле, что он возможно, но не обязательно, за конечное количество шагов находит решение проблемы, называются *частично разрешимыми проблемами.* Например, проблема останова является частично разрешимой проблемой, а проблемы эквивалентности и тотальности не являются таковыми.

Д. 1.6. Сложностные классы задач и проблема $P=NP$[1]

В начале 1960-х годов, в связи с началом широкого использования вычислительной техники для решения практических задач, возник вопрос о границах практической применимости данного алгоритма решения задачи в смысле ограничений на ее размерность. Какие задачи могут быть решены на ЭВМ за реальное время? Теоретический ответ на этот вопрос был дан в работах Кобмена (Alan Cobham, 1964), и Эдмнодса (Jack Edmonds, 1965), где был независимо введен **сложностный** класс задач P.

Д.1.6.1. Класс P (задачи с полиномиальной сложностью)

Задача называется полиномиальной, т. е. относится к классу P, если существует константа k и алгоритм, решающий эту задачу за время $O(n^k)$, где n есть длина входа алгоритма в битах ([2]). Отметим, что класс задач P определяется через существование полиномиального по времени алгоритма ее решения, при этом неявно предполагается худший случай по времени для всех различных входов длины n.

Задачи класса P - — это, интуитивно, задачи, решаемые за реальное время. Отметим следующие преимущества задач из этого класса:

- для большинства реальных задач из класса P константа k меньше 6;
- класс P инвариантен по модели вычислений (для широкого класса моделей);

[1] В основном тексте книги **Макконелла** (глава 8) сложностный класс NP вводится на основании рассмотрения недетерминированных полиномиальных алгоритмов. Здесь описан другой подход к определению класса NP — через сертификат и полиномиально проверяющий алгоритм, даются также иллюстрации вложенности классов и краткий исторический обзор.

- класс P обладает свойством естественной замкнутости (сумма или произведение полиномов есть полином).

Таким образом, задачи класса P есть уточнение определения «практически разрешимой» задачи для входов больших размерностей.

Д.1.6.2. Класс NP (полиномиально проверяемые задачи)

Представим себе, что некоторый алгоритм получает решение некоторой задачи. Мы вправе задать вопрос — соответствует ли полученный ответ поставленной задаче, и насколько быстро мы можем проверить его правильность?

Рассмотрим в качестве примера задачу о сумме:

> Дано n целых чисел, содержащихся в массиве $A = (a_1, \ldots, a_n)$ и целое число V. Найти массив $X = (x_1, \ldots, x_n)$, $Xi \in \{0,1\}$, такой, что $\sum a_i \cdot Xi = V$. Содержательно: может ли быть представлено число V в виде суммы каких-либо чисел из массива $A1$

Если какой-то алгоритм выдает результат — массив X, то проверка правильности этого результата может быть выполнена с полиномиальной сложностью, а именно проверка того, что $\sum a_i x_i - V$ требует не более $O(n)$ операций.

Попытаемся описать сложность проверки решения более формально. Каждому конкретному множеству исходных данных D, $|D| = n$ поставим в соответствие сертификат 5, такой, что $S| = O(n^l)$, где / — некоторая константа и алгоритм $A_s - A_s(D,S)$,такой, что он выдает «1», если решение правильно, и «О», если решение неверно. Тогда задача принадлежит сложностному классу NP, если существует константа т, и алгоритм A_s имеет временную сложность не более чем $O(n^m)$ ([2]). Содержательно задача относится к классу NP, если ее решение, полученное некоторым алгоритмом, может быть полиномиально проверено. Класс NP был впервые введен в работах Эдмондса.

Д.1.6.3. Проблема $P = NP$

После введения в теорию алгоритмов понятий сложностных классов Эдмондсом (Edmonds, 1965), была сформулирована основная проблема теории сложности -- $P - NP$?, и высказана гипотеза о несовпадении этих классов. Словесно проблему можно сформулировать следующим образом: можно ли все задачи, решение которых проверяется с полиномиальной сложностью, решить за полиномиальное время?

Очевидно, что любая задача, принадлежащая классу P, принадлежит и классу NP, т. к. она может быть полиномиально проверена, при этом задача проверки решения может состоять просто в повторном решении задачи.

На сегодня отсутствуют теоретические доказательства как совпадения этих классов (P = NP), так и их несовпадения. Предположение состоит в том, что класс P является собственным подмножеством класса NP, т. е. множество задач $NP \setminus P$ не пусто, как это показано на рис. Д. 1.1. Это предположение опирается на существование еще одного класса, а именно класса NP-полных задач.

Д.1.6.4. Класс *NPC* (*NP*-полные задачи)

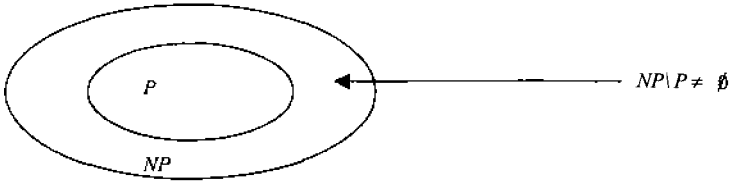


Рис. **Д.** 1.1. Соотношение классов *P* и *NP*

Понятие NP-полноты было введено независимо Куком (Stephen Cook, 1971) и Левиным (1973) и основывается на понятии сводимости одной задачи к другой. Сводимость задач может быть представлена следующим образом: если мы имеем задачу 1 и решающий эту задачу алгоритм, выдающий правильный ответ для всех конкретных проблем, а для задачи 2 алгоритм решения неизвестен, то если мы можем переформулировать (свести) задачу 2 в терминах задачи 1, то мы решаем задачу 2 с помощью алгоритма решения задачи 1.

Таким образом, если задача 1 задана множеством конкретных проблем D_{A1}, а задача 2 — множеством D_{A2}, и существует функция f_s (алгоритм), сводящая конкретную постановку d_{A2} задачи 2 к конкретной постановке d_{A1} задачи 1, т.е. $f_s(d_{A2} \in D_{A2}) = d_{A1} \in D_{A1}$, то задача 2 сводима к задаче 1.

Если при этом временная сложность алгоритма f_s есть $O(n^k)$; т.е. алгоритм сведения принадлежит классу *P,* то говорят, что *задача 2 полиномиально сводится к задаче 1* ([2]).

В теории сложности вычислений принято говорить, что задача задается некоторым языком, тогда если задача 1 задана языком *L1,* а задача 2 - языком *L2,* то полиномиальная сводимость языков, задающих задачи, обозначается следующим образом: $L2 \leq_p L1$.

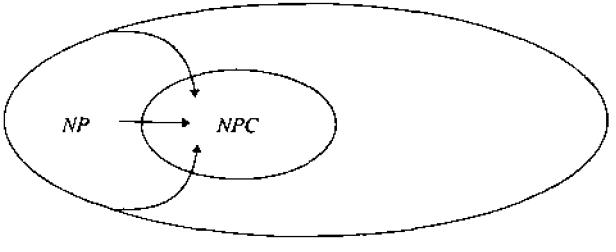


Рис.Д. 1.2. Сводимость задач и класс *NPC*

Определение класса NPC (NP-complete) или класса *NP*-полных задач требует выполнения следующих двух условий: во-первых, задача должна принадлежать классу *NP* (L 6 *NP*), и, во-вторых, к ней полиномиально должны сводиться все задачи из класса *NP* ($L_x \leq_P L$, для каждого $L_x \in$ $\in$ *NP*), что схематично представлено на рис. Д.1.2. Для класса *NPC* доказана следующая теорема:

Теорема **Д.1.2.** Если существует задача, принадлежащая классу NPC, для которой существует полиномиальный алгоритм решения, то класс P совпадает с классом *NP,* т.е. $P = NP$ ([2]).

Схема доказательства состоит в сведении с полиномиальной трудоемкостью любой задачи из класса NP к данной задаче из класса *NPC* и решении этой задачи за полиномиальное время (по условию теоремы).

В настоящее время доказано существование сотен NP-полных задач ([2, 11]), но ни для одной из них пока не удалось найти полиномиального алгоритма решения. Отметим, что для многих из них предложены приближенные полиномиальные алгоритмы ([2]). Сегодня ученые предполагают следующее соотношение классов, показанное на рис. Д.1.3, а именно $P \neq NP$, то есть $NP \setminus P \neq 0$, и ни одна задача из класса NPC не может быть решена, по крайней мере, сегодня, с полиномиальной сложностью.

Исторически первое доказательство NP-полноты было предложено Куком для задачи о выполнимости схемы. Метод сведения за полиномиальное время был предложен Карпом (Richard Karp) и использован им для доказательства NP-полноты целого ряда задач. Описание многих NP-полных задач содержится в книге Гэри и Джонсона [11], описание некоторых других классов задач, рассматриваемых в теории сложности, можно найти, например, в [12].

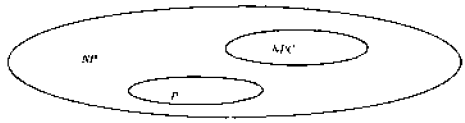


Рис. Д.1.3. Соотношение классов *P, NP, NPC*

Д. 1.7. Классы открытых и закрытых задач и теоретическая нижняя граница временной сложности[2]

В современной теории алгоритмов известны доказательства нижних границ временной сложности для целого ряда задач ([2, 6]). Результаты, полученные в области практической разработки и анализа алгоритмов, позволяют указать наиболее асимптотически эффективные на данный момент алгоритмы. В связи с этим возможно сравнение этих результатов и построение классификации вычислительных задач, отражающей, достигает ли наиболее эффективный из известных в настоящее время алгоритмов доказанной нижней границы временной сложности задачи. Дальнейшее изложение ведется в соответствии с [13].

Д. 1.7.1. Теоретически доказанная нижняя граница сложности - - $f_{thlim}(n)$

Рассмотрим некоторую вычислительную задачу Z. Обозначим, пользуясь терминологией введенной Э. Постом, через DZ множество конкретных допустимых проблем данной задачи Z. Введем так же следующие обозначения — пусть RZ есть множество правильных решений, а Ver — верификационная функция задачи. Пусть $D \in DZ$ — конкретная допустимая проблема данной задачи — вход алгоритма. Обозначим через AZ полное множество всех алгоритмов (включающее как

[2] Эта новая классификация задач предложена автором дополнения в [13] и основана на сравнении теоретической сложности задачи и наиболее эффективного алгоритма ее решения. Такая классификация позволит разработчикам алгоритмического обеспечения более четко понять возможные пределы улучшения эффективности алгоритмов.

известные, так и будущие алгоритмы), решающих задачу Z в понимании алгоритма как финитного 1-процесса по Посту, т. е. любой алгоритм A из AZ применим $\forall D \in DZ$, заканчивается и дает правильный ответ:

$$AZ = \{ A \setminus A : D \to R,\ D \in D_Z,\ R \in R_Z,\ Ver(D, R) = True,\ f_A(D) \neq \text{oc}\ \forall D \in D_Z \},$$

где $f_A(D)$ — функция трудоемкости алгоритма для данного входа, значением которой является количество элементарных операций в принятой модели вычислений, задаваемое алгоритмом A на конкретном входе D.

В общем случае существует подмножество множества D_Z, включающее все конкретные проблемы, имеющие размерность n — обозначим его через D_{nz}:

$$D_{nz} = \{ D | D \in D_Z,\ |D| = n \}.$$

Введем содержательно понятие теоретически доказанной нижней границы временной сложности задачи с обозначением $f_{thlim}(n)$, как в оценки некоторой функции, аргументом которой является длина входа, такой что, теоретически невозможно предложить алгоритм, решающей данную задачу асимптотически быстрее, чем с оценкой $f_{thlim}(n)$. Более корректно это понятие означает, что имеется строгое доказательство того, что любой алгоритм решения данной задачи в худшем случае на входе длины п имеет временную сложность не лучше чем $f_{thlim}(n)$, и данная функция представляет собой точную верхнюю грань множества функций, образующих асимптотическую иерархию (по отношению «$\prec$»), для которых такое доказательство возможно. Иначе говоря, для любой функции $g(n) \succ f_{thlim}(n)$ указанное свойство не может быть доказано. Заметим, что отношения $\prec$ и $\succ$ для сравнения двух функций по асимптотике роста введены в 1871 году Поль-Дюбуа Раймоном:

$$f(x) \prec \partial(x) \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{\partial(x)} = 0, \qquad f(x) \succ \partial(x) \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{\partial(x)} = \infty.$$

Дадим формальное определение для $f_{thlim}(n)$. Определим множество F_{lim}, состоящее ИЗ фуНКЦИЙ f_{lim}:

$$F_{lim} = \{ f_{lim} | \forall A \in A_Z,\ \forall n > 0,\ f_A^{\wedge}(n) = ft(f_{lim}(n)) \},$$

где $f_A^{\wedge}(n)$ — функция трудоемкости для худшего случая из всех входов, имеющих размерность п, т. е. для любого D 6 D_{nz}, тогда:

$$f_{thlim}(n) = \sup_{\prec} \{F_{lim}\}.$$

Классическим примером задачи с теоретически доказанной нижней границей сложности является задача сортировки массива по возрастанию с использованием сравнений. Для этой задачи имеется строгое доказательство, основанное на рассмотрении бинарного дерева сравнений, того, что невозможно отсортировать массив, сравнивая элементы между собой, быстрее, чем за $\Theta(\log_2(n!)) = \Theta(n \cdot \log_2(n))$ ([14]).

Д.1.7.2. Тривиальная (эвристическая) нижняя граница сложности - - $f_{tr}(n)$

Независимо от того, доказана или нет теоретическая нижняя граница временной сложности, мы можем для подавляющего большинства вычислительных задач указать нижнюю границу сложности на основе рациональных рассуждений, например,

- оценка $\Theta(n)$, если для решения задачи необходимо как минимум обработать все исходные данные (задача поиска максимума в массиве, задача умножения матриц, задача коммивояжера);
- оценка 0(1) для задач с необязательной обработкой всех исходных данных, например задача поиска в списке по ключу, при условии, что сам список является входом алгоритма.

Обозначим эту оценку через $f_{tr}(n)$ и будем предполагать априорно, что оценка $f_{tr}(n)$ существует для задачи Z, тогда можно указать следующие возможные соотношения между $f_{thlim}(n)$ и $f_{tr}(n)$:

- оценка $f_{thlim}(n)$ не доказана для задачи Z и для данной задачи принимается оценка $f_{tr}(n)$;
- оценка $f_{thlim}(n)$ доказана, тогда
  а) либо $f_{tr}(\mathrm{n}) \prec f_{thlim}(n)$;
  б) либо $fir(n) = fthiim(n)$.

Д.1.7.3. Оценка сложности наилучшего известного алгоритма - - $f_{Amin}(n)$

Рассмотрим множество A_R всех известных алгоритмов, решающих задачу Z, очевидно, что A_R с A_Z, отметим, что множество A_R конечно.

Определим асимптотически лучший по сложности алгоритм, обозначив его через A_{min}. Отметим, что как это чрезвычайно часто бывает на практике, алгоритмы, имеющие лучшие асимптотические оценки, дают плохие по трудоемкости результаты на малых размерностях из-за значительных коэффициентов при главном порядке функции трудоемкости. В качестве примера можно привести «быстрые» алгоритмы умножения длинных двоичных целых чисел, предложенные Карацубой, Штрассеном и Шенхаге ([2, 6]), реально эффективные, начиная с чисел, имеющих двоичное представление в несколько сотен и тысяч битов соответственно. Заметим так же, что может существовать либо один, либо несколько алгоритмов с минимальной по всему множеству A_R асимптотической оценкой — в таком случае мы выбираем один из них в качестве алгоритма представителя.

Формально определим множество алгоритмов $AM = \{A_m\}$как подмножество известных алгоритмов A_R, обладающих минимальной асимптотической оценкой трудоемкости и выделим во множестве AM любой алгоритм:

$$AM = \{A_m | f_{A_m}(\mathrm{n}) \prec f_A(\mathrm{n}) \ \forall A \in A_R \backslash A_M\},$$

$$|AM| > 1, \ A_{min} \in A_M.$$

Д.1.7.4. Классификация задач по соотношению теоретической нижней границы временной сложности задачи и сложности наиболее эффективного из существующих алгоритмов

Сравнивая между собой полученные оценки для некоторой задачи — $f_{thlim}(n)$, если она существует, $f_{tr}(n)$ и $f_{Amin}(n)$, можно предложить следующую классификацию задач, отражающую соотношение оценки сложности задачи и наиболее эффективного из известных алгоритмов ее решения ([13]).

Класс задач ***THCL*** *(Theoretical close).* Это задачи, для которых $f_{thlim}(n)$ существует и $f_{Amin}(n) = \Theta(f_{thlim}(n))$. Можно говорить, что это класс теоретически (по временной сложности) закрытых задач, т. к. существует алгоритм, решающий данную задачу с асимптотической временной сложностью равной теоретически доказанной нижней границе, что и отражено в названии этого класса. Разработка новых или модификация известных алгоритмов из этого класса может лишь преследовать цель улучшения коэффициента при главном порядке в функции трудоемкости.

Приведем несколько примеров задач этого класса:

1) Задача поиска максимума в массиве:

$$f_{thlim}(n) = 0\ (n);$$
$$f_{Amin}(n) = 0\ (n).$$

2) Задача сортировки массива сравнениями:

$$f_{thlim}(n) = 0\ (n \cdot \log_2(n));$$
$$f_{Amin}(n) = 0\ (n \cdot \log_2(n)).$$

Примером может служить алгоритм сортировки пирамидой ([14]).

3) Задача умножения длинных двоичных целых чисел:

$$f_{thlim}(n) = 0\ (\ln(n) \cdot \ln\ln(n)),$$
$$f_{Amin}(n) = \Theta\ (\ln(n) \cdot \ln\ln(n)).$$

Асимптотически оптимальным является алгоритм Штрассена-Шенхаге ([2]).

Класс задач ***PROP*** *(Practical open).* Это задачи, для которых $f_{thlim}(n)$ существует и $f_{Amin}(n) \succ f_{thlim}(n)$. Таким образом, для задач из этого класса существует зазор между теоретической нижней границей сложности и оценкой наиболее эффективного из существующих сегодня алгоритмов ее решения.

Класс *PROP* — это «практически открытые» задачи, т. е. для задач этого класса имеет место практически значимая проблема разработки алгоритма, обладающего доказанной теоретической нижней границей временной сложности. В случае разработки такого алгоритма данная задача будет переведена в класс *THCL*. Отметим, что достаточно часто на основе самого доказательства теоретической нижней границы временной сложности может быть построен алгоритм решения данной задачи, что определяет небольшое количество задач в данном классе.

Класс задач **THOP** *(Theoretical open).* Это задачи, для которых оценка $f_{thlim}(n)$ не доказана, а $f_{Amin}(n) \succ f_{tr}(n)$. Для задач из этого класса существует зазор между тривиальной нижней границей временной сложности и оценкой наиболее эффективного из существующих сегодня алгоритмов ее решения.

Аббревиатура *THOP* — «теоретически открытые» задачи, отражает тот факт, что для задач этого класса имеет место теоретическая проблема либо доказательства оценки $f_{thlim}(n)$, может быть даже равной $f_{tr}(n)$, переводящего задачу в класс *PROP,* либо доказательства глобальной оптимальности наиболее эффективного из существующих алгоритмов, переводящего задачу в класс *THCL.*

Отметим, что в теоретическом плане класс *HOP* достаточно широк. В этом классе находятся, например, все задачи из класса NPC (NP-полные задачи). В качестве другого непосредственного примера можно указать задачу умножения квадратных матриц, для которой $f_{Amin}(n) = O(n^{2,34})$ ([2]), а тривиальная оценка, отражающая необходимость просмотра всех элементов исходных матриц, имеет вид $f_{tr}(n) = \Theta(n^2)$, при этом теоретическая нижняя граница временной сложности для этой задачи пока не доказана.

Литература к разделу Д.1.

[1] Успенский В. А. Теорема Геделя о неполноте. — М.: Наука, 1982 г. — 112 с. — (Популярные лекции по математике).

[2] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 2001 г. — 960 с.

[3] Гасанов Э. Э., Кудрявцев В. Б. Теория хранения и поиска информации. — М.: ФИЗМАТЛИТ, 2002 г. — 288 с.

[4] Кнут Д. Искусство программирования. Тома 1, 2, 3. 3-е изд. Пер. с англ.: Уч. пос. — М.: Изд. дом «Вильяме», 2001 г.

[5] Ахо А. Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Том 1. — Синтаксический анализ. — М.: Мир, 1978 г. — 612 с.

[6] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильяме», 2001 г. — 384 с.

[7] Мошков М.Ю. Деревья решений. Теория и приложения. — Нижний Новгород: Изд-во Нижегородского ун-та, 1994 г.

[8] Успенский В.А. Машина Поста. — М.: Наука, 1979 г. — 96 с. — (Популярные лекции по математике).

[9] Карпов Ю.Г. Теория автоматов - СПб.: Питер, 2002 г. — 224с., ил.

[10] Матиясевич Ю. В. Диофантовы перечислимые множества // Доклады АН СССР, 1970. Т. 191. С.279 - 282.

[11] Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи М.: Мир, — 1982. — 416 с.

[12] Хопкрофт Дж., Мотовани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений, 2-ое изд.: Пер. с англ. — М.: Издательский дом «Вильяме»; 2002. — 528 с.

[13] Ульянов М. В. Классификации и методы сравнительного анализа вычислительных алгоритмов. — М.: Издательство физико-математической литературы, 2004 г. — 212 с.

[14] Гудман С., Хидетниеми С, Введение в разработку и анализ алгоритмов. — М.: Мир, 1981. — 368 с.

Глава Д.2. Оценкитрудоемкости

Д.2.1. Функция трудоемкости и система обозначений

В результате анализа сложности алгоритма мы получаем асимптотическую оценку (или «порядок алгоритма» см. 1.4.1. основного текста) для количества задаваемых им операций. Зная такие оценки для разных алгоритмов решения определенной задачи, мы получаем возможность сравнить эти алгоритмы с точки зрения их эффективности. Однако асимптотические оценки указывают не более чем *порядок* функции, и результаты сравнения будут справедливы только при очень больших размерах входа.

Для сравнения эффективности алгоритмов при реальных размерах входа и прогнозирования времени выполнения их программных реализаций необходимо знание о точном количестве операций — функций трудоемкости исследуемых алгоритмов. Получение функции трудоемкости требует, очевидно, больших усилий, чем анализ сложности алгоритма. Тем не менее, эти усилия окупаются получением такой информации, которая позволяет проводить практически значимый сравнительный анализ алгоритмов. В этом смысле позволим себе не согласиться с Дж. Макконеллом, — «Точное знание количества операций, выполненных алгоритмом, не играет существенной роли в анализе алгоритмов» (стр. 32 основного текста).

Будем рассматривать в дальнейшем, придерживаясь терминологии Э. Поста, применимые к общей проблеме, правильные и финитные алгоритмы, т. е. алгоритмы, представляющие собой финитный 1-процесс (см. Д.1.3). В качестве модели вычислений будем рассматривать абстрактную машину, включающую процессор с фон-Неймановской архитектурой, адресную память и набор «элементарных» операций, соотнесенных с языком высокого уровня — такая модель вычислений называется машиной с произвольным доступом к памяти (RAM).

Определение Д.2.1. *Трудоемкость алгоритма.* Под трудоемкостью алгоритма для данной конкретной проблемы, заданной множеством D, будем понимать количество «элементарных» операций, задаваемых алгоритмом в принятой модели вычислений. Соответствующую функцию будем обозначать как $f_A(D)$. Заметим, что значением функции трудоемкости является целое положительное число.

При более детальном анализе ряда алгоритмов оказывается, что не всегда количество элементарных операций задаваемых алгоритмом, т. е. значение $f_A(D)$ на входе длины п, где $n = \backslash D \backslash$, совпадает с количеством операций на другом входе такой же длины. Предположим, что мы можем исследовать все возможные множества D длины п, (Вы можете подсчитать, сколько существует разных множеств D длины п, если, например, элементами множества являются двухбайтовые слова?), тогда мы можем экспериментально определить граничные значения $f_A(D)$. Значительно более интересной задачей является теоретическое получение этих границ.

Введем специальные обозначения, связанные с граничными значениями функции трудоемкости данного алгоритма при фиксированной длине входа.

Пусть DA — множество допустимых конкретных проблем данной задачи, решаемой алгоритмом A, $D \in DA$ — конкретная проблема (вход алгоритма A), представляющая собой упорядоченное множество элементов (слов) d:

$$D = \{ di,\ i = \overline{1.\ n} \},\ |D| = n.$$

В общем случае существует подмножество (для большинства алгоритмов собственное) множества D_A, включающее все конкретные проблемы, имеющие размерность n, — обозначим его через D_n:

$$D_n = \left\{ D \middle|\ |D| = n \right\}.$$

В предположении о том, что элементы d_i представляют собой слова фиксированной длины в алфавите $\{0, 1\}$, множество D_n является конечным — обозначим его мощность через M_{Dn}, т.е. $M_{Dn} = |D_n|$. Тогда алгоритм A, имея на входе различные множества D б D_n, будет, возможно, задавать в каком-то случае наибольшее, а в каком-то случае наименьшее количество операций.

Введем следующие обозначения ([1]), отметив, что соответствующая терминология является устоявшейся в литературе по анализу алгоритмов ([2, 3]).

1. $f_A^{\wedge}(n)$ — *худший случай* — наибольшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерности n:

$$f_A^{\wedge}(n) = \max_{D \in D_n} \{ f_A(D) \}.$$

2. $f_A^{\vee}(n)$ — *лучший случай* — наименьшее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерности n:

$$f_A^{\vee}(n) = \min_{D \in D_n} \{ f_A(D) \}.$$

3. $\overline{f}_A(n)$ — *средний случай* — среднее количество операций, совершаемых алгоритмом A для решения конкретных проблем размерности n:

$$\overline{f}_A(n) = \frac{1}{M_{D_n}} \sum_{D \in D_n} f_A(D).$$

Д.2.2. Классификация алгоритмов на основе функции трудоемкости

Дальнейшее исследование алгоритмов на основе функции трудоемкости связано с определением тех факторов, которые влияют на значение функции трудоемкости. К этим факторам можно отнести количество элементов множества D, значения этих элементов и порядок их следования. На основе такого исследования можно построить классификацию алгоритмов по степени влияния этих факторов на функцию трудоемкости ([1]).

Выделим в функции $f_A(D)$ количественную (зависящую от n) и параметрическую (зависящую от значений и порядка элементов в D) составляющие, обозначив их через $f_n(n)$ и $g_p(D)$ соответственно,

$$f_A(D) = f_A(f_n(n), g_p(D)).$$

Для большинства алгоритмов эта зависимость может быть представлена как композиция функций $f_n(n)$ и $g_p(D)$ либо в мультипликативной

$$f_A(D) = f_n(n) \cdot g_p^*(D),$$

либо в аддитивной форме

$$f_A(D) = f_n(n) + g_p^+(D).$$

Укажем, что мультипликативная форма характерна для ряда алгоритмов, когда сильно зависящий от значений элементов в D внешний цикл, определяющий перебор вариантов, содержит внутренний цикл, проверяющий решение с полиномиальной зависимостью от размерности. Такая ситуация возникает, например, для ряда алгоритмов, решающих задачи из класса NPC.

В силу конечности множества D_n, существует худший случай для параметрической составляющей функции трудоемкости. Введем следующие обозначения:

$$\partial^*(\text{п}) = \max_{D \in D_n} \{ g_p^*(D) \}, \quad g^+(\text{п}) = \max_{D \in D_n} \{ g_n^+(D) \}.$$

В зависимости от влияния различных характеристик множества исходных данных на функцию трудоемкости алгоритма, может быть предложена следующая классификация, имеющая практическое значение для анализа алгоритмов:

I. *Класс N* — класс количественно-зависимых по трудоемкости алгоритмов. Это алгоритмы, функция трудоемкости которых зависит только от размерности конкретного входа

$$g^+(n) = 0 \quad \Rightarrow \quad g^*(n) = 1 \quad \Rightarrow \quad f_A(D) = f_n(n).$$

Примерами алгоритмов с количественно-зависимой функцией трудоемкости могут служить алгоритмы для стандартных операций с массивами и матрицами - умножение матриц, умножение матрицы на вектор и т. д. Анализ таких алгоритмов, как правило, не вызывает затруднений. Заметим, что для алгоритмов класса N справедливо соотношение

$$f_A^\wedge(n) = f_A^\vee(n) = f_A(n).$$

II. *Класс PR* — класс параметрически зависимых по трудоемкости алгоритмов. Это алгоритмы, трудоемкость которых определяется не размерностью входа (как правило, для этого класса размерность входа обычно фиксирована), а конкретными значениями всех или некоторых элементов из входного множества D:

$$f_n(n) = const = \text{c}, \quad \Rightarrow \quad f_A(D) = \text{c} \cdot g_*^*(D)$$

Примерами алгоритмов с параметрически-зависимой трудоемкостью являются алгоритмы вычисления стандартных функций с заданной точностью путем вычисления соответствующих степенных рядов. Очевидно, что такие алгоритмы, имея на входе два числовых значения — аргумент функции и точность, задают зависящее только от значений количество операций. Например:

- вычисление x^k последовательным умножением, $f_A(D) = f_A(x, k)$;
- вычисление значения экспоненты с заданной точностью ε:

$$e^x = \sum (x^n/n!). \quad f_A(D) = f_A(x, \varepsilon).$$

III. *Класс* NPR — класс количественно-параметрических по трудоемкости алгоритмов. Это достаточно широкий класс алгоритмов, т. к. в большинстве практических случаев функция трудоемкости зависит как от количества данных на входе, так и от их значений. В этом случае

$$f_n(n) \neq const, \quad \partial^*(n) \neq 1, \quad \Rightarrow \quad f_A^{\wedge}(n) \neq f_A^{\vee}(n),$$
$$f_A(D) = f_n(\text{п}) \cdot g_p^*(D), \quad \text{или} \quad f_A(D) = f_n(\text{п}) + g_n^+(D).$$

В качестве одного из общих примеров можно привести ряд алгоритмов численных методов, в которых параметрически-зависимый внешний цикл по точности включает в себя количественно-зависимый фрагмент по размерности, порождая мультипликативную форму для $f_A(D)$.

По отношению к количественно-параметрическим алгоритмам представляет интерес более детальная классификация, ставящая целью выяснение степени влияния количественной и параметрической составляющей на главный порядок функции трудоемкости и приводящая к выделению следующих подклассов в классе NPR:

III. 1. *Подкласс NPRL (Low)* — подкласс алгоритмов, трудоемкость которых слабо зависит от параметрической составляющей

$$g^+(n) = 0 \ (f_n(n)) \quad \Leftrightarrow \quad g^*(n) = 0(1).$$

Для алгоритмов этого подкласса параметрическая составляющая влияет не более чем на коэффициент главного порядка функции трудоемкости, который определяется количественной составляющей. В этом случае можно говорить о коэффициентно-параметрической функции трудоемкости.

К этому подклассу относится, например, алгоритм поиска максимума в массиве, т. к. количество переприсваиваний максимума в худшем случае (массив отсортирован по возрастанию), определяющее $g^+(n) = \Theta(n)$, имеет равный порядок с оценкой внешнего цикла для перебора п элементов.

III. 2. *Подкласс NPRE (Equivalent)* — подкласс алгоритмов, в трудоемкости которых составляющая $g^*(n)$ имеет порядок роста, не превышающий $f_n(n)$:

$$g^*(\text{п}) = 0 \ (f_n(n)) \quad \Leftrightarrow \quad g^+(n) = \Theta\left(f_n^2(n)\right).$$

Таким образом, для алгоритмов этого подкласса параметрический компонент имеет сопоставимое (в мультипликативной форме) с количественным компонентом

влияние на главный порядок функции трудоемкости. Для алгоритмов, относящихся к этому подклассу, можно говорить о квадратично-количественной функции трудоемкости.

В этот подкласс входит, например, алгоритм сортировки массива методом пузырька, для которого количество перестановок элементов в худшем случае (обратно отсортированный массив) определяет порядок

$$g^{+}(\text{п}) = \text{в}\left(n^2\right), \quad \text{а} \quad f_n(\text{п}) = 0\,(n).$$

III. 3. *Подкласс NPRH (High)* — подкласс алгоритмов, в трудоемкости которых составляющая $g^*(n)$ имеет асимптотический порядок роста выше, чем $f_n(n)$:

$$/\text{п}\ (n) \prec g^*(n).$$

Для алгоритмов этого подкласса именно параметрический компонент определяет главный порядок функции трудоемкости. Можно говорить, что эти алгоритмы являются количественно-сильно-параметрическими по функции трудоемкости алгоритмами.

В этот подкласс входят, например, уже упоминавшиеся в основном тексте книги алгоритмы точного решения NP-полных задач.

Для алгоритмов этого подкласса характерна, как правило, мультипликативная форма функции трудоемкости, которая особенно ярко выражена в алгоритмах большинства итерационных вычислительных методов. В них внешний цикл по точности порождает параметрическую составляющую, а трудоемкость тела цикла имеет количественную оценку.

Другая, и не зависимая от предыдущей, классификация алгоритмов в классе NPR предполагает выделение в функции $g_p(D)$ аддитивных компонент в множестве D, связанных со значениями элементов — $g_v(D)$, и их порядком — $g_s(D)$. Будем предполагать, что функция $g_p(D)$ представима в виде

$$g_p(D) = g_v(D) + g_s(D).$$

Выделим во множестве $D \in D_n$ подмножество однородных по существу задачи элементов — D_e, состоящее из элементов

$$\{d_1, \ldots, d_m\}, \quad |D_e = m, \quad m < n,$$

и определим D_p как множество всех упорядоченных последовательностей из d_i, отметим, что в общем случае $|D_p| = m!$.

Если D_{e1}, $D_{e2} \in Dp$, то соответствующие им множества D_1, $D_2 \in D_n$, тогда порядковая зависимость $g_s(D)$ для функции трудоемкости имеет место, если $f_A(D_1) \neq f_A(D_2)$ хотя бы для одной пары D_1, $D_2 \in D_n$.

Зависимость от значений $g_v(D)$ предполагает, что

$$f_A(D) = f_A(n, p_1, \ldots, p_m), \quad m \leq n, \quad pi \in D, \quad i = \overline{1, m}.$$

Оценивая степень влияния $g_v(D), g_s(D)$ на $g_p(D)$, определим подклассы в классе NPR.

III. a. *Подкласс NPRS (Sequence)* — подкласс алгоритмов с количественной и порядково-зависимой функцией трудоемкости:

$$g_v(D) \prec g_s(D).$$

В этом случае трудоемкость зависит от размерности входа и от порядка расположения однородных элементов; зависимость от значений не может быть полностью исключена, но она не является существенной. В этом случае можно говорить о количественно-порядковом характере функции трудоемкости.

Примерами могут служить большинство алгоритмов сортировки сравнениями, алгоритмы поиска минимума и максимума в массиве.

III. b. *Подкласс NPRV (Value)* — подкласс алгоритмов с функцией трудоемкости, зависящей от длины входа и значений элементов в D:

$$g_s(D) \prec g_v(D).$$

В этом случае трудоемкость зависит от размерности входа и от значений элементов входного множества; зависимость от порядка не является определяющей.

Пример — алгоритм сортировки методом индексов ([3]), трудоемкость которого определяется количеством исходных чисел и значением максимального из них. При этом порядок чисел в массиве вообще не оказывает влияния на трудоемкость, за исключением фрагмента поиска максимума, лежащего в классе $NPRS$.

Отметим, что объединение этих подклассов не образует класс NPR:

$$NPR \setminus (NPRS \cup NPRV) \neq 0.$$

Существуют алгоритмы, в которых и значения, и порядок расположения однородных элементов оказывают реальное и существенное влияние на функцию трудоемкости — например, таковыми являются итерационные алгоритмы решения систем линейных уравнений. Очевидно, что как перестановка значений в исходной матрице, так и изменение самих значений существенно меняют собственные числа матрицы, которые определяют сходимость итерационного процесса получения решения.

Д.2.3. Элементарные операции в процедурном языке высокого уровня

Для того, чтобы получить значения функции трудоемкости, и теоретические и экспериментальные, необходимо определить какие операции в нашей модели вычислений являются «элементарными». Поскольку в основном алгоритмы записываются на языке, близком к одному из процедурных языков высокого уровня (см., например, основной текст книги), то наша задача — определить «элементарные» операции такой системы записи алгоритма.

Практически значимые реализации вычислительных алгоритмов включают в себя обычно следующие программные фрагменты:

- диалоговый или файловый ввод исходных данных;

- проверка исходных данных на допустимость;
- собственно решение поставленной задачи;
- представление (вывод) полученных результатов.

В рамках анализа алгоритмов по трудоемкости мы можем считать, что обслуживающие фрагменты программной реализации (ввод, проверка и вывод) являются общими или эквивалентными для разных алгоритмов решения данной задачи. Такой подход приводит к необходимости анализа только непосредственного алгоритма решения. При этом предполагается размещение исходных данных и результатов в «оперативной» памяти, включая в это понятие как собственно оперативную память, так и кэш память, регистры и буфера реального процессора. Таким образом, множество элементарных операций, учитываемых в функции трудоемкости, не включает операции ввода/вывода данных на внешние носители.

Для формального процедурного языка высокого уровня такими «элементарными» операциями, коррелированными с основными операциями языков процедурного программирования, будем считать следующие:

1) простое присваивание: a $\leftarrow$ 6;
2) одномерная индексация: $a[i]$: (адрес(a) + г $\cdot$ длина элемента);
3) арифметические операции: {*, /, -, +};
4) операции сравнения: a { <, >, =, $\leq$, > } b;
5) логические операции: $(l1)$ {*or, and, not*} $(l2)$;
6) операции адресации в сложных типах данных: $(name1.name2)$.

По отношению к введенному набору элементарных операций алгоритмического базиса формального процедурного языка высокого уровня необходимо сделать несколько замечаний.

- Опираясь на идеи структурного программирования, из набора элементарных операций исключается команда перехода, поскольку ее можно считать связанной с операцией сравнения в конструкции ветвления или цикла по условию. Такое исключение оправдано запретом использования оператора перехода на метку в идеологии структурного программирования.
- Операции доступа к простым именованным ячейкам памяти считаются связанными с операциями, операнды которых они хранят.
- Конструкции циклов не рассматриваются, т. к. могут быть сведены к указанному набору элементарных операций (см. Д.2.4).

Д.2.4. Методика анализа основных алгоритмических конструкций

Несмотря на наличие разнообразных подходов к созданию компьютерных программ и различных технологий программирования, классический процедурный подход остается определяющим при программной реализации алгоритмов. Основными алгоритмическими конструкциями в этом подходе являются конструкции следования, ветвления и цикла. Для получения функции трудоемкости некоторого алгорит-

ма необходима методика анализа основных алгоритмических конструкций. С учетом введенных «элементарных» операций такая методика сводится к следующим положениям.

а) *Конструкция «Следование».* Трудоемкость конструкции есть сумма трудоемкостей блоков, следующих друг за другом

$$f_{\text{следование}} = f_1 + \cdots + f_k,$$

где k — количество блоков в конструкции «Следование».

Отметим, что трудоемкость самой конструкции не зависит от данных, при этом очевидно, что блоки, связанные конструкцией «Следование», могут обладать сильно зависящей от данных трудоемкостью.

б) *Конструкция «Ветвление».*

if (0

then

блок с трудоемкостью f_{then}, выполняемый с вероятностью p;

else

блок с трудоемкостью f_{else}, выполняемый с вероятностью $(1 - p)$.

В этой конструкции (l) обозначает логическое выражение, состоящее из логических переменных и/или арифметических, символьных или других по типу сравнений с ограничением 0(1) на трудоемкость его вычисления. Вероятности перехода на соответствующие блоки могут меняться, как в зависимости от данных, так и в зависимости от параметров внешних охватывающих циклов и/или других условий. Достаточно трудно дать какие-либо общие рекомендации для получения p вне зависимости от конкретного алгоритма или особенностей входа ([2, 4, 5]).

Общая трудоемкость конструкции «Ветвление» для построения функции трудоемкости в среднем случае требует анализа для получения вероятности p выполнения переходов на блоки «then» и «else». При известном значении вероятности p трудоемкость конструкции определяется как

$$f_{\text{ветвление}} = f_l + f_{then} \cdot p + f_{else} \cdot (1 - p),$$

где f_l — трудоемкость вычисления условия (l).

В общем случае вероятность перехода p есть функция исходных данных и связанных с ними промежуточных результатов $p = p(D)$.

Очевидно, что для анализа худшего случая может быть выбран тот блок ветвления, который имеет большую трудоемкость, а для лучшего случая — блок с меньшей трудоемкостью.

в) *Конструкция «Цикл по счетчику».*

```
for i ← 1 to n                    i ← 1
    тело цикла          ⇔           тело цикла
end                                 i ← i + 1
                                  until i ≤ n
```

После сведения конструкции к введенным элементарным операциям, ее трудоемкость определяется следующим образом

$$f_{\text{цикл}} = 1 + \delta \cdot n + n \cdot f_{\text{тело цикла}}.$$

Анализ вложенных циклов по счетчику с независимыми индексами не составляет труда и сводится к погружению трудоемкости цикла в трудоемкость тела охватывающего его цикла.

Для k вложенных зависимых циклов трудоемкость определяется в виде вложенных сумм с зависимыми индексами (см. 1.3.3 основного текста).

г) *Конструкция «Цикл по условию».* Конкретная реализация цикла по условию (цикл с верхним или нижним условием) не меняет методику оценки его трудоемкости. На каждом проходе выполняется оценка условия и может быть изменение каких-либо переменных, влияющих на значение этого условия. Общие рекомендации по определению суммарного количества проходов цикла крайне затруднительны из-за сложных зависимостей от исходных данных. Для худшего случая могут быть использованы верхние граничные оценки. Так, например, для задачи решения системы линейных уравнений итерационными методами количество итераций по точности (сходимость) определяется собственными числами исходной матрицы, трудоемкость вычисления которых сопоставима по трудоемкости с получением самого решения.

Отметим, что для получения функций трудоемкости для лучшего, среднего и худшего случаев при фиксированной размерности задачи, если алгоритм не принадлежит классу N, особенности анализа алгоритмических конструкций, зависящих от данных («ветвление» и «цикл по условию») будут различны.

Несмотря на достаточно простые подходы к анализу основных алгоритмических конструкций, получение функций трудоемкости алгоритмов остается достаточно сложной задачей, требующей применения специальных подходов и методов, а иногда и введения специальных функций. Тем не менее, для количественно-зависимых алгоритмов из класса N можно получить точное значение функции трудоемкости.

Основные трудности анализа алгоритма в среднем случае связаны как с выяснением значений трудоемкости для конкретных входов, так и с определением вероятности их появления. В связи с этим задача вычисления $\overline{f}_A(n)$ для алгоритмов класса *NPR* является достаточно трудоемкой. Для получения оценок сложности алгоритмов в среднем случае в литературе предлагаются различные методы как общего, так и частного характера, например метод вероятностного анализа ([2]), амортизационный анализ ([3]), метод классов входных данных (см. 1.1.1 основного текста). Эти методы могут быть использованы и для получения функции трудоемкости алгоритмов в среднем случае.

Предлагаемая методика опирается на введенные «элементарные» операции процедурного языка высокого уровня, методику анализа алгоритмических конструкций и известные методы анализа алгоритмов. Описания различных методов анализа алгоритмов содержатся в целом ряде литературных источников ([2, 3, 4, 5]), посвященных анализу сложности алгоритмов, в частности и в основном тексте книги. Приводимые ниже примеры показывают, как на этой основе с учетом введенных

«элементарных» операций языка реализации можно получить функции трудоемкости алгоритмов.

Д.2.5. Примеры анализа трудоемкости алгоритмов

В качестве примеров применения методики для анализа алгоритмов по функции трудоемкости рассмотрим ряд алгоритмов, относящихся к классу N и различным подклассам класса NPR.

Пример Д.2.1. *Алгоритм суммирования элементов квадратной матрицы.*

```
SumM (A, n; Sum)
Sum ← 0                          (1 операция)
For i ← 1 to n                   (3 операции на 1 проход цикла)
  For j ← 1 to n                 (3 операции на 1 проход цикла)
    Sum ← Sum + A[i,j]           (4 операции)
  end for j
end for i
End.
```

Этот простейший алгоритм, очевидно, выполняет одинаковое количество операций при фиксированном значении n. Таким образом, он является количественно-зависимым и относится к классу N. Внутренний цикл не зависит от внешнего, что позволяет непосредственно применить методику анализа конструкции «Цикл по счетчику», в этом случае

$$f_A(n) = 1 + 1 + n \cdot (3 + 1 + n \cdot (3 + 4)) = 7 \cdot n^2 + 4 \cdot n + 2 = 0\left(n^2\right).$$

Заметим, что под n здесь понимается линейная размерность матрицы, в то время как на вход алгоритма подается n^2 значений. Такой подход не согласуется с классической теорией алгоритмов, в которой сложность алгоритма определяется как функция длины входа [6], но достаточно часто используется в литературе по практическому применению методов анализа алгоритмов ([2]).

Пример Д.2.2. *Алгоритм поиска максимума в массиве*

```
MaxS (S,n; Max)
Max ← S[1]                       (2 операции)
For i ← 2 to n                   (3 операции на 1 проход цикла)
  if Max < S[i]                  (2 операции)
    then Max ← S[i]              (2 операции)
end for
return Max
End
```

Данный алгоритм является количественно-параметрическим и относится к классу $NPRS$, поэтому при фиксированной размерности входа необходимо проводить отдельный анализ для худшего, лучшего и среднего случая.

1) *Худший случай.* Максимальное количество переприсваиваний максимума (на каждом проходе цикла) будет в том случае, если элементы массива отсортированы по возрастанию. Трудоемкость алгоритма в этом случае равна:

$$f_A^{\wedge}(\text{п}) = 1 + 1 + 1 + (\text{п} - 1) \cdot (3 + 2 + 2) = 7 \cdot n - 4 = 0\,(\text{п})$$

2) *Лучший случай.* Минимальное количество переприсваивания максимума (ни одного на каждом проходе цикла) будет в случае, если максимальный элемент расположен на первом месте в массиве. Трудоемкость алгоритма в этом случае равна:

$$f_A^{\vee}(\text{п}) = 1 + 1 + 1 + (\text{п} - 1) \cdot (3 + 2) = 5 \cdot \text{п} - 2 = \Theta\,(\text{га})$$

3) *Средний случай.* Алгоритм поиска максимума последовательно перебирает элементы массива, сравнивая текущий элемент массива с текущим значением максимума. На очередном шаге, когда просматривается k-ый элемент массива, переприсваивание максимума произойдет, если в подмассиве из первых k элементов максимальным элементом является последний.

Очевидно, что в случае равномерного распределения S[i], вероятность того, что максимальный из k элементов расположен в определенной (последней) позиции равна $1/k$. Тогда в массиве из n элементов общее количество операций переприсваивания максимума определяется как:

$$\sum_{i=1}^{n} \frac{1}{i} = H_n \approx \ln(n) + \gamma, \quad \gamma \approx 0.57.$$

Величина H_n называется n-ым гармоническим числом, 7 — постоянная Эйлера. Таким образом, точное значение (математическое ожидание) среднего количества операций присваивания в алгоритме поиска максимума в массиве из п элементов определяется величиной H_n (при очень большом количестве испытаний), тогда:

$$\bar{f}_A(\text{п}) = 1 + (\text{п} - 1) \cdot (3 + 2) + 2 \cdot (\ln(n) + 7) = 5 \cdot \text{п} + 2 \cdot \ln(\text{га}) - 4 + 2 \cdot 7 = \text{в}\,(n).$$

Заметим, что полученная функция трудоемкости справедлива для принятого предположения о равномерном распределении значений элементов.

Пример Д.2.3. *Алгоритм сортировки массива методом прямого включения.* Это достаточно хорошо известный алгоритм, использующий метод последовательного включения нового элемента в уже отсортированную часть массива. В основном тексте книги он называется «сортировка вставками».

```
Sort_Ins(A, n)
For i ← 2 to n                    (3 операции на 1 проход цикла)
  key ← A[i]                      (2 операции)
  j ← i-1                         (2 операции)
  While (j>0) and (A[j]>key)      (4 операции)
    A[j+1] ← A[j]                 (4 операции)
    j ← j-1                       (2 операции)
    A[j+1] ← key                  (3 операции)
end for
End
```

Алгоритм относится к классу $NPRS$, т. к. количество сравнений и перемещений определяется для фиксированного набора значений элементов порядком их расположения в исходном массиве. Для анализа трудоемкости в среднем определим, следуя [2], среднее общее количество сравнений, используя метод вероятностного анализа. Заметим, что при вставке очередного элемента на i-ом шаге (г — 1 элементов уже отсортированы) существует г позиций для вставки числа, в предположении о равновероятном попадании нового элемента в любую позицию алгоритм выполняет в среднем г/2 сравнений и перемещений для его обработки, что в сумме по всему внешнему циклу дает

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{n^2}{4} - \frac{n}{4}.$$

Используя методику анализа алгоритмических конструкций с учетом того, что на последнем проходе цикла While выполняется только сравнение, окончательно имеем

$$\overline{f}_A(n) = 1 + (n-1) \cdot (3+2+2+4+3) + (4+4+2) \cdot (n^2/4 - n/4) = \\ = 2{,}5 \cdot n^2 + 11{,}5 \cdot n - 13.$$

Пример Д.2.4. *Алгоритм решения задачи о сумме методом прямого перебора.* Словесно задача о сумме формулируется как задача нахождения таких чисел из данной совокупности, которые в сумме дают заданное число; классически задача формулируется в терминах целых чисел ([3]). В терминах структур данных языка высокого уровня задача формулируется как задача определения таких элементов исходного массива 5 из п чисел, которые в сумме дают заданное число V. Отметим, что задача относится к сложностному классу NPC (см. задачу об упаковке рюкзака в 8.2.3 основного текста), а анализируемый ниже точный алгоритм ее решения не может быть использован реально для больших размерностей задачи. Алгоритм приведен здесь в качестве примера использования методики амортизационного анализа ([3]) для исследования трудоемкости двоичного счетчика.

Формулировка задачи о сумме. Дано: массив S_i, г = 1, . . . , п и число V,требуется определить такие элементы Si, что $\sum_{i=1}^{n} S_i = V$. Условия существования решения имеют вид

$$\min_{i=1,n} \{S_i\} \le V \le \sum_{i=1}^{n} S_i.$$

Получим асимптотическую оценку сложности решения данной задачи для алгоритма, использующего прямой перебор всех возможных вариантов. Поскольку исходный массив содержит п чисел, то проверке на равенство V подлежат следующие варианты решений:

- V состоит из 1 элемента $\Rightarrow$ $C_n^1 = n$ вариантов;
- V состоит из 2 элементов $\Rightarrow$ $C_n^2 = (n \cdot (n-1)/2$ вариантов;
- и т. д. до проверки одного варианта с n слагаемыми.

Поскольку сумма биномиальных коэффициентов для степени n равна 2^n, и для каждого варианта необходимо выполнить суммирование (с оценкой $O(n)$) для проверки на V, то оценка сложности алгоритма в худшем случае имеет вид:

$$f_A^{\wedge}(\text{п}) = O(n \cdot 2^n).$$

Определим вспомогательный массив, хранящий текущее сочетание исходных чисел в массиве 5, подлежащих проверке на V,— массив $Cnt[j]$,элемент массива равен «О», если число $S[j]$не входит в V и равен «1», если число $S[j]$входит в V. Решение получено, если $V - \Sigma S[j] \cdot Cnt[j], j = 1,n$.

Может быть предложена реализация механизма полного перебора вариантов, использующая двоичный счетчик, организованный в массиве *Cnt*. Задача об увеличении двоичного счетчика в массиве *Cnt* на «1» может быть решена без использования алгоритма сложения битовых чисел следующим образом:

- если в массиве *Cnt* есть одна или более «1» справа подряд (... 0111), то увеличение на «1» приводит к сбросу всех правых «1» и установке в «1» следующего самого правого «О»;
- если самый правый элемент массива *Cnt* содержит «О» (... 0), то увеличение на «1» приводит к переустановке «О» в «1» в самой правой позиции.

Рассматривая массив *Cnt* как указатель на элементы массива 5, подлежащие суммированию в данный момент, мы производим суммирование и проверку на V до тех пор, пока решение не будет найдено или же безрезультатно будут просмотрены все возможные варианты сумм.

Таким образом, алгоритм точного решения задачи о сумме методом прямого перебора, реализующий перебор с помощью двоичного счетчика, имеет следующий вид:

```
TaskSum(S, n, V; Cnt, fl)
fl ← false
i ← 0
repeat
  Cnt[i] ← 0
  i ← i+1
Until i > n
Cnt[n] ← 1
Repeat
  Sum ← 0
  i ← 1
  Repeat
    Sum ← Sum + S[I]*Cnt[i]
    i ← i+1
  Until i > n
  if Sum = V
      fl ← true
  Return(Cnt,fl)
  j ← n
  While Cnt[j] = 1
     Cnt[j]=0
      j ← j - 1
  Cnt[j] ← 1
  Until Cnt[0]=1
  Return(Cnt,fl)
  End.
```

Алгоритм относится к классу $NPRH$, поскольку количество задаваемых алгоритмом операций существенно определяется значениями исходных данных при фиксированной длине входа. Рассмотрим лучший и худший случай для данного алгоритма.

а) В лучшем случае, когда последний элемент массива совпадает со значением V: V = S[n], необходимо выполнить только одно суммирование для проверки, что приводит к оценке

$$f_A^{\vee}(\text{п}) = \text{е}\,(n).$$

Функция трудоемкости для лучшего случая имеет вид

$$f_A^{\vee}(n) = 2 + (n+1)\,.\,(3+2) + 2 + 2 + n \cdot (3+5) + 1 + 1 = 13 \cdot n + 13.$$

б) В худшем случае, если решения вообще нет, то придется проверить все варианты, и следовательно

$$f_A^{\wedge}(\text{п}) = \text{в}\,(\text{п} \cdot 2^n).$$

Получим функцию трудоемкости для худшего случая, используя принятую методику подсчета элементарных операций

$$f_A^{\wedge}(\text{п}) = 2 + (n+1) \cdot (3+2) + 2 + (2^n - 1) \cdot (2 + n \cdot (3+5) + 1 + 1 + f_{cnt} + 2 + 2). \quad (\text{Д.2.1})$$

Для получения значения f_{cnt} — количества операций, необходимых для увеличения счетчика на «1», рассмотрим по шагам проходы цикла While, в котором выполняется изменение значения счетчика — таблица Д.2.1.

Таблица Д.2.1.

Cnt	Количество проходов в While	Операций
001	1	1 • 6+2
010	0	2
011	2	2 • 6+2
100	0	2
101	1	1 • 6+2
ПО	0	2
111	3	3 • 6+2

Учитывая, что вероятности появления четных и нечетных чисел равны, трудоемкость увеличения счетчика может быть получена следующим образом:

$$f_{cnt} = (1/2) \cdot 2 + (1/2) \cdot 2 + (1/2) \cdot ((1/2) \cdot 1 \cdot 6 + (1/4) \cdot 2 \cdot 6 + (1/8) \cdot 3 \cdot 6 + \ldots) =$$

$$= 2 + (1/2) \cdot 6 \cdot \left(\frac{1}{2} + \frac{2}{} + \frac{3}{2^3} + \ldots\right) \le 2 + 3 \cdot \sum_{k=1}^{\infty} \frac{k}{2^k}.$$

Так как $\sum k \cdot x^k = x/(1-x)^3$ (см. [3]), то $\sum k \cdot (1/2)^k = (1/2)/(1-(1/2))^2 = 2$, и, следовательно, $f_{cnt} < 8$, и не зависит от длины массива счетчика.

Подстановка f_{cnt} в (Д.2.1) дает

$$/\text{л}\,(\text{п}) = 4 + 5 \cdot n + 5 + (2^n - 1) \cdot (8 \cdot \text{п} + 16),$$

и окончательно

$$f_A^{\wedge}(\text{п}) = 8 \cdot n \cdot 2^{\text{тм}} + 16 \cdot 2^n - 3 \cdot n - 7.$$

Анализ трудоемкости в среднем случае может основываться на предположении, что все возможные значения трудоемкости равновероятны (обратите внимание на

то, как велик разброс трудоемкости при фиксированном значении n). При этом предположении в среднем мы ожидаем приблизительно половину трудоемкости худшего случая. Предлагаем читателю провести небольшой, но очень полезный вычислительный эксперимент — реализуйте этот алгоритм на Вашем компьютере, и выполните его для 32 чисел в массиве S задав значение V, превышающее сумму (для получения трудоемкости в худшем случае). Какое время выполнения Вы ожидаете, и каково оно на самом деле?

Пример Д.2.5. *Алгоритм быстрого возведения числа в целую степень.* Задача о быстром возведении числа в целую степень, т.е. вычисление значения $y = x^n$ для целого n, лежит в основе алгоритмического обеспечения многих криптосистем ([7]). Отметим, что в этом аспекте применения вычисления производятся с целым значением x в группе вычетов по mod k. Представляет интерес детальный анализ известного быстрого алгоритма возведения в степень методом последовательного возведения в квадрат ([3]). В целях этого анализа представляется целесообразным введение трех следующих специальных функций ([1]).

Функция $\beta(n)$. Функция определена для целого положительного n, и $\beta(n)$ есть количество значащих битов в двоичном представлении целого числа n. Отметим, что функция $\beta(n)$ может быть задана аналитически в виде

$$\beta(n) = [\log_2(n)] + 1,$$

где $[z]$ — целая часть z, $n > 0$.

Функция $\beta_1(n)$. Функция определена для целого положительного n, и $\beta_1(n)$ есть количество единиц в двоичном представлении положительного целого числа n. Отметим, что функция $\beta_1(n)$ не является монотонно возрастающей функцией, например $\beta_1(n) = 1$, для всех $n = 2^k$, *а* для n = 2^k - 1 значение $\beta_1(n) - k$. График функции $\beta_1(n)$ для начальных значений n представлен на рис. Д.2.1. В силу определения $\beta_1(n)$ справедливо следующее неравенство

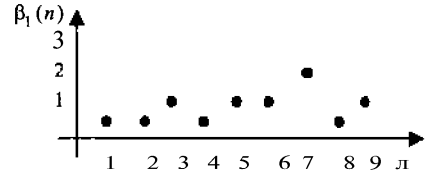


Рис. Д.2.1. Значения функции $\beta_1(n)$ для $n = 1, 2, \ldots, 9$

$$1 \le \beta_1(n) \le \beta(n),$$

поскольку

$$\beta(\text{п}) = [\log_2(n)] + 1, \text{ то } \beta_1(\text{п}) = O(\log_2(n)).$$

Отметим, что функция $\beta_1(n)$ не задается аналитически, в [8] дано словесное описание функции, эквивалентной функции $\beta_1(n)$, и указан следующий рекурсивный способ ее задания:

$$\begin{cases} \beta_1(0) = 0; \\ \beta_1(1) = 1; \\ \beta_1(2n) = \beta_1(n); \\ \beta_1(2n+1) = \beta_1(n) + 1. \end{cases}$$

Функция $\beta_0(n)$. Функция определена для целого положительного п, и $\beta_0(n)$ есть количество нулей в двоичном представлении числа п. Отметим, что функция $\beta_0(n)$ не является монотонно возрастающей функцией, так как $\beta_0(n) = 0$ для всех $n = 2^k - 1$.

Для любого п > 0 справедливо соотношение

$$\beta(n) = \beta_0(n) + \beta_1(n).$$

Для дальнейшего анализа представляет также интерес определение среднего значения функции $\beta_1(n)$ для n = О, 1, 2, N, где $N = 2^k - 1$ (т.е. если двоичное представление числа п занимает не более k двоичных разрядов), обозначим его через $\beta_s(N)$, тогда

$$A^{(N)} = \frac{1}{N+1}\sum_{m=0}^{N}\beta_1(m).$$

Поскольку количество чисел, имеющих l единиц в k разрядах, равно количеству сочетаний из / по k ([9]), то

$$\sum_{ra=0}^{N}\beta_1(\text{т}) = \sum_{l=1}^{k} l \cdot C_k^l = \sum_{l=1}^{k} \frac{k!}{(l-1)!(k-l)!} = k \cdot \sum_{l=0}^{k-1} C_{k-1}^l = k \cdot 2^{k-1}.$$

Поскольку $N = 2^k - 1$, то

$$\beta_s(N) = \frac{1}{N+1}\sum_{m=0}^{N}\beta_s(m) = \frac{k \cdot 2^{k-1}}{2^k - 1 + 1} = \frac{k}{2} = \frac{\log_2(N+1)}{2} = \frac{\beta(N)}{2}. \qquad (Д.2.2)$$

Идея быстрого алгоритма решения задачи о возведении в степень состоит в использовании двоичного представления числа п и вычисления степеней x путем повторного возведения в квадрат ([3]). Пусть, например, п = 11, тогда

$$x^{11} = x^8 \cdot x^2 \cdot x^1, \quad x^2 = x \cdot x, \quad x^4 = x^2 \cdot x^2, \quad x^8 = x^4 \cdot x^4.$$

Алгоритмическая реализация требует последовательного выделения битов числа n, возведения x в квадрат и умножения y на те двоичные степени x, для которых в двоичном разложении n присутствует единица.

```
Xn(x,n;y);
z ← x;
y ← 1;
Repeat
  If (n mod 2) = 1                (проверка на «1» в текущем разряде)
    then
      y ← y*z;                    (умножение на текущую степень x)
    z ← z*z;                      (повторное возведение в квадрат)
    n ← n div 2;                  (переход к следующему разряду)
Until n = O
Return (y)
End
```

Получим точную функцию трудоемкости данного алгоритма, используя введенные ранее обозначения и принятую методику счета элементарных операций:

$$f_A(\text{п}) = 2 + \beta(\text{п}) \cdot (2+2+2+1) + \beta_1(\text{п}) \cdot (2) = 7 \cdot \beta(\text{п}) + 2 \cdot \text{ft}(\text{п}) + 2. \quad (\text{Д.2.3})$$

Количество проходов цикла определяется количеством битов в двоичном представлении числа n — $\beta(n)$, а количество повторений операции формирования результата — $y \leftarrow y*z$ — количеством единиц в этом представлении — ft (n), что и отражает формула (Д.2.3).

Определим трудоемкость алгоритма для особенных значений п, при которых функция ft *(n)* достигает текущего минимума или текущего максимума. Это происходит при

$$n = 2^k, \quad \text{или} \quad \text{п} = 2^k - 1, \quad k - \{1, 2, \ldots\}.$$

В случае если

$$n = 2^k, \quad \text{то} \quad \text{ft(n)} = 1 \quad \text{и} \quad f_A(n) = 7 \cdot \beta(n) + 4,$$

однако при

$$n = 2^k - 1, \quad \text{значение} \quad \text{ft} \ (\text{п}) = \beta(\text{п}) \quad \text{и} \quad f_A(\text{п}) = 9 \cdot \beta(n) + 2.$$

Отметим интересный факт, что при $\text{п} = 2^k 1 - 1$ увеличение значения п на единицу приводит, при $k > 3$, к уменьшению количества совершаемых алгоритмом операций. Так как при

$$n = 2^k - 1, \quad \beta(n) = \text{fc}, \ \beta(n+1) = \text{fc} + 1, \ \text{ft(n)} = \text{fc}, \ \text{ft} \ (n+1) = 1,$$

то, следовательно,

$$/\text{л}(\text{п}) = 9 \cdot k + 2, \quad f_A(n+1) = 7 \cdot (k+1) + 2 = 7 \cdot k + 9.$$

Если показатель степени заранее не известен, то можно получить среднюю оценку, в предположении, что представление числа п занимает не более k двоичных разрядов, т. е. $\log_2(n) < k$. Тогда используя формулы (Д.2.2) и (Д.2.3) имеем

$$\overline{f}_A(n) < 7 \cdot \beta(\text{п}) + 2 \cdot \beta_s(n) + 2 = 8 \cdot \log_2(\text{п}) + 2.$$

Таким образом, количество операций, выполняемых быстрым алгоритмом возведения в степень, линейно зависит от количества битов в двоичном представлении показателя степени. Введение специальных функций ft (та) и $\beta(n)$ позволяет получить точное значение функции трудоемкости анализируемого алгоритма.

Д.2.6. Анализ сложности рекурсивных алгоритмов

Одним из основных методов построения рекурсивных алгоритмов является метод декомпозиции. Идея метода состоит в разделении задачи на части меньшей размерности, получении решений для выделенных частей и объединении решений при возврате рекурсивных вызовов.

Если в алгоритме происходит разделение задачи на b подзадач, которое приводит к необходимости решения a подзадач размерностью n/b, то функцию трудоемкости (см. [3]) можно представить в виде

$$f_A(n) = a \cdot f_A(n/b) + d(n) + U(n), \qquad (Д.2.4)$$

где $d(n)$ — трудоемкость алгоритма деления задачи на подзадачи, $U(n)$ — трудоемкость алгоритма объединения полученных решений.

Рассмотрим, например, известный алгоритм сортировки слиянием, принадлежащий Дж. Фон Нейману (см. 3.6. основного текста). На каждом рекурсивном вызове переданный массив делится пополам, что дает оценку для $d(n)$ — 0(1), далее рекурсивно вызывается сортировка полученных массивов половинной длины (до тех пор, пока длина массива не станет равной единице), и возвращенные отсортированные массивы объединяются с трудоемкостью $\Theta(n)$. Тогда ожидаемая трудоемкость на сортировку составит:

$$f_A(n) = 2 \cdot f_A(n/2) + 0\ (1) + 0\ (n).$$

Тем самым возникает вопрос о получении оценки сложности функции трудоемкости, заданной в виде (Д.2.4), для произвольных целых значений a и b. Ответ на этот вопрос можно получить на основе теоремы о рекуррентных соотношениях, авторами которой являются Дж. Бентли, Д. Хакен и Дж. Сакс (статья 1980 г.), приведем ее формулировку в соответствии с [3].

Теорема Д.2.1. Пусть $a > 1$, $b > 1$ — константы, $g(n)$ — функция, пусть далее $/(n) = a \cdot f(n/b) + \partial(n)$, где $n/b = \lfloor n/b \rfloor$ или $n/b = \lceil n/b \rceil$, тогда

(1) Если $\partial(n) = O\left(n^{\log_b a - \varepsilon}\right)$, $\varepsilon > 0$, то $f(n) = 0\ \left(n^{\log_b a}\right)$.
Пример: $/(n) = 8 \cdot f(га/2) + n^2$, тогда $/(n) = 0\ \left(n^3\right)$.

(2) Если $g(n) = \Theta\left(n^{\log_b a}\right)$, то $/(n) = 0\ \left(n^{\log_b a} \cdot \log n\right)$.
Пример: $/(n) = 2 \cdot /(n/2) + 0\ (n)$, тогда $/(n) = 0\ (n \cdot \log n)$.

(3) Если $\partial(n) = 0\ \left(n^{\log_b a + \varepsilon}\right)$, $e > 0$, то $/(n) = 0\ (g(n))$.
Пример: $/(n) = 2 \cdot /(n/2) + n^2$, здесь подходит случай (3), поскольку

$$n^{\log_b a} = n^1, \text{ и следовательно } /(n) = 0\ \left(n^2\right).$$

Данная теорема является мощным средством анализа асимптотической сложности рекурсивных алгоритмов, использующих метод декомпозиции, но, к сожалению, она не дает возможности получить в явном виде коэффициенты функции трудоемкости. Для решения этой задачи необходим более детальный анализ рекурсивного дерева.

На основании этой теоремы получим оценку сложности функции трудоемкости для алгоритма сортировки слиянием. Поскольку в этом случае $\partial(n) = 0\ \left(n^{\log_2 2}\right)$, то $/д(n) = 0\ (n \cdot \log n)$.

Д.2.7. Трудоемкость рекурсивной реализации алгоритмов

Анализ сложности рекурсивных алгоритмов затронут в основном тексте книги Макконелла, однако для получения функции трудоемкости необходим более детальный анализ, учитывающий трудоемкость вызова рекурсивной функции и количество вершин рекурсивного дерева.

Большинство современных языков высокого уровня поддерживают механизм рекурсивного вызова, когда функция, как элемент структуры языка процедурного программирования, может вызывать сама себя с другим аргументом. Эта возможность позволяет напрямую реализовывать вычисление рекурсивно определенных функций. Отметим, что в силу тезиса Черча–Тьюринга аппарат рекурсивных функций Черча равномощен машине Тьюринга, и, следовательно, любой итерационный алгоритм может быть реализован рекурсивно.

Рассмотрим пример рекурсивной функции, вычисляющий значение факториала целочисленного аргумента:

```
F(n)
If  (n=0  or  n=l)          (проверка возможности прямого вычисления)
  then
    F ← 1
  else
    F ← n*F(n− 1)           (рекурсивный вызов функции)
Return  (F)
End.
```

Анализ трудоемкости рекурсивных реализаций алгоритмов, очевидно, связан как с количеством операций, выполняемых при одном вызове функции, так и с количеством таких вызовов. Графическое представление порождаемой данным алгоритмом цепочки рекурсивных вызовов называется деревом рекурсивных вызовов. Более детальное рассмотрение приводит к необходимости учета затрат на организацию вызова функции и передачи параметров.

Должны быть учтены и затраты на возврат вычисленных значений и передачу управления в точку вызова. Эти операции должны быть включены в функцию трудоемкости рекурсивно заданного алгоритма. Можно заметить, что некоторая ветвь дерева рекурсивных вызовов обрывается при достижении такого значения передаваемого параметра, при котором функция может быть вычислена непосредственно. Таким образом, рекурсия эквивалентна конструкции цикла, в котором каждый проход есть выполнение рекурсивной функции с заданным параметром.

Рассмотрим на примере рис. Д.2.2 организацию рекурсии для функции вычисления факториала.

Дерево рекурсивных вызовов может иметь и более сложную структуру, если на каждом вызове порождается несколько обращений. Возвращаясь к рисунку Д.2.1, отметим, что при каждом рекурсивном вызове выполняется ряд операций, обслуживающих этот вызов. Это приводит к необходимости анализа трудоемкости обслуживания рекурсии.

Механизм вызова функции или процедуры в языке высокого уровня существенно зависит от архитектуры компьютера и операционной системы. В рамках архитектуры и операционных систем IBM PC совместимых компьютеров этот механизм реализован с помощью стека. Как передаваемые в процедуру или функцию фактические параметры, так и возвращаемые из них значения помещаются в программный стек специальными командами процессора. Дополнительно сохраняются значения необходимых регистров и адрес возврата в вызывающую процедуру. Схематично этот механизм проиллюстрирован на рис. Д.2.3.

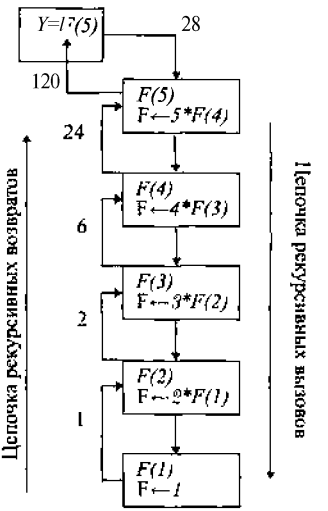


Рис. Д.2.2. Цепочка рекурсивных вызовов и возвратов при вычислении факториала для значения $n = 5$.

Для подсчета трудоемкости вызова будем считать операции помещения слова в стек и выталкивания из стека элементарными операциями в формальной системе, соотнесенными с операцией присваивания. Тогда при вызове процедуры или функции в стек помещается адрес возврата, состояние необходимых регистров процессора, адреса возвращаемых значений и передаваемые параметры. После этого выполняется переход по адресу на вызываемую процедуру, которая извлекает переданные фактические параметры, выполняет вычисления и помещает результаты по указанным в стеке адресам. При завершении работы вызываемая процедура восстанавливает регистры, выталкивает из стека адрес возврата и осуществляет переход по этому адресу.

Для анализа трудоемкости вызова/возврата введем обозначения:

m — количество передаваемых фактических параметров,

k — количество возвращаемых по адресной ссылке значений,

r — количество сохраняемых в стеке регистров.

Тогда трудоемкость в элементарных операциях на один вызов и возврат составит:

$$f_{\text{Вызова}} = m + k + r + 1 + m + k + r + 1 = 2 \cdot (m + k + r + 1). \qquad (Д.2.5)$$

Анализ трудоемкости рекурсивных алгоритмов в части трудоемкости самого рекурсивного вызова можно выполнять разными способами в зависимости от того, как формируется итоговая сумма элементарных операций:

- отдельно по цепочкам рекурсивных вызовов и возвратов;
- совокупно по вершинам дерева рекурсивных вызовов.

Продемонстрируем этот подход на примере рекурсивного вычисления факториала. Для рассмотренного выше рекурсивного алгоритма вычисления факториала количество вершин рекурсивного дерева равно, очевидно, п, при этом передается и возвращается по одному значению (m — 1, k = 1), а на последнем рекурсивном вызове значение функции вычисляется непосредственно - в итоге в предположении о сохранении четырех регистров (r = 4) получаем

$$f_A(\text{п}) = \text{п} \cdot 2 \cdot (1 + 1 + 4 + 1) + (\text{п} - 1) \cdot (1 + 3) + 1 \cdot 2 = 18 \cdot \text{п} - 2.$$

Отметим, что n — параметр алгоритма, а не количество слов на входе.

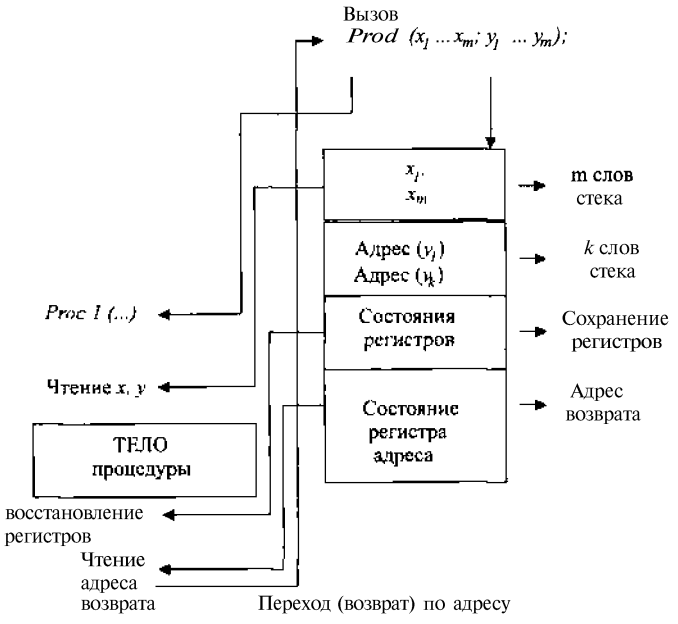


Рис. **Д.2.3.** Механизм вызова процедуры с использованием стека

Д.2.8. Методика подсчета вершин рекурсивного дерева

Для учета особенностей рекурсивной реализации, связанных с трудоемкостью рекурсивных вызовов, возвратов и передачей параметров, при построении функции трудоемкости рекурсивного алгоритма необходимо получить количество вершин рекурсивного дерева как функцию от характеристик множества входных данных. Более того, необходимо учесть, что для листьев рекурсивного дерева алгоритм будет выполнять непосредственное вычисление значений и возврат на уровень вверх, что приводит к необходимости отдельного вычисления трудоемкости при останове рекурсии.

Таким образом, для получения функции трудоемкости необходим детальный анализ дерева рекурсивных вызовов, результатом которого должно быть количество внутренних (порождающих рекурсию) вершин дерева — $V_r(n)$, и количество листьев дерева — $V_1(n)$, связанных с остановом рекурсии.

Для внутренних вершин рекурсии, возможно, выполняется обработка возвращаемых результатов, трудоемкость которой может зависеть от положения вершины в дереве. Введем нумерацию вершин, начиная с корня, по уровням дерева, обозначая вершины через $v_k, k = 1, \ldots, V_r(n)$, а трудоемкость обработки в этой вершине обозначим через $g(v_k)$.

Обозначая через f_r трудоемкость алгоритма на подготовку и реализацию одного рекурсивного вызова, а через f_1 трудоемкость алгоритма при останове рекурсии в

одной вершине, окончательно имеем

$$f_A(n) = V_r(n) \cdot f_r + V_1(n) \cdot \text{Л} + \sum_{k=1}^{V_r(n)} \text{ffK}). \qquad (\text{Д.2.6})$$

Этот подход к получению функции трудоемкости для рекурсивных реализаций будем называть в дальнейшем методикой подсчета вершин рекурсивного дерева.

Рассмотрим такой подход к получению функции трудоемкости рекурсивного алгоритма на примере алгоритма сортировки слиянием. Идея алгоритма состоит в разделении входного массива на две примерно равные части (для нечетной длины массива одна из частей будет на единицу больше), рекурсивном вызове для сортировки частей и слиянии отсортированных массивов после возврата из двух рекурсивных вызовов. В классической реализации останов рекурсии происходит при единичной длине (см. пункт 3.6 основного текста).

Рекурсивный алгоритм *Merge_Sort*[1)] получает на вход массив чисел A и два индекса p и q, указывающие на ту часть массива, которая будет сортироваться при данном вызове.

```
Merge_Sort (A, p, q, Bp, Bq)
if p ≠ q            (проверка на останов рекурсии при p = q)
  then
    r←(p+q) div 2
    Merge_Sort (A,p, r , Bp, Bq)
    Merge_Sort (A, r+1, q, Bp, Bq)
    Merge(A, p, r, q, Bp, Bq)
Return (A)
End.
```

Вспомогательные массивы Bp и Bq используются для слияния отсортированных частей, хотя возможна и несколько более трудоемкая реализация, использующая слияние по месту, но не требующая дополнительных массивов.

Рассмотрим алгоритм слияния отсортированных частей массива A, использующий дополнительные массивы Bp и Bq, в конец которых с целью остановки движения индекса помещается максимальное значение.

Поскольку сам алгоритм рекурсивной сортировки работает так, что объединяемые части массива A находятся рядом друг с другом, то алгоритм слияния вначале копирует отсортированные части в промежуточные массивы, а затем формирует объединенный массив непосредственно в массиве A по указанным индексам. Для удобства анализа в записи алгоритма справа указано количество элементарных операций в данной строке.

```
Merge (A, p, r, q, Bp, Bq)
  max ← A[r]                                    2
```

[1)] Здесь приводится несколько отличающаяся от основного текста книги (см. 3.6.) реализация идеи сортировки слиянием в части подхода к слиянию отсортированных массивов.

```
    If Max < A[q] then                                  2
      max ← A[q]                                        1/2·2
  kp ← r − p + 1                                        3
  p1 ← p − 1                                            2
  for i ← 1 to kp (копирование первой части)            1+ kp·3
    Bp [ i ] ← A[p1 + i ]                               kp·4
  Bp[ kp+1] ← max                                       3
  kq ← q - r                                            2
  for i ← 1 to kq (копирование второй части)            1+ kq·3
    Bq [ i ] ←A[ r + i ]                                kq·4
  Bq [ kq+ 1] ← max                                     3
  PP ← P                                                1
  pq ← r+1                                              2
  for i ← p to q (слияние частей)                       1+m·3
    if Bp [ pp ] < Bq [ pq ]                            m·3
    then
      A[ i ] ← Bp[ pp ]                                 1/2·m·3
      PP ← PP +1                                        1/2·m·2
    else
      A [ i ] ← Bq [ pq ]                               1/2·m·3
      pq ← pq +1                                        1/2·m·2
Return (A)
End.
```

На основании указанного в строках количества операций и с учетом того, что значение $m = kp + kq - q - p + 1$ есть длина объединенного массива, можно получить трудоемкость алгоритма слияния отсортированных массивов как функцию длины массива результата:

$$f_{merge}(m) = 2 + 2 + 1 + 3 + 2 + 1 + 7 \cdot kp + 3 + 2 + 1 + 7 \cdot kq + 3 + 1 + 2 + 1 + m \cdot (3 + 3 + 3 + 2) = 11 \cdot m + 7 \cdot (kp + kq) + 24 = 18 \cdot m + 24. \quad (Д.2.7)$$

Заметим, что приведенный алгоритм слияния отсортированных массивов не зависит по трудоемкости от данных (класс N), этот парадокс объясняется тем, что блоки в конструкции «Ветвление» содержат одинаковое количество операций, и, следовательно, вероятностные переходы не влияют на трудоемкость.

Алгоритм, получая на входе массив из n элементов, делит его пополам при первом вызове, и это деление продолжается рекурсивно вплоть до единичных элементов массива. Продемонстрируем методику прямого подсчета вершин для размерности входа $n = 2^k$, $k = \log_2(n)$.

В этом случае мы имеем полное дерево рекурсивных вызовов глубиной k, содержащее n листьев; фрагмент дерева показан на рис. Д.2.4. Обозначим общее количество вершин дерева через V, тогда

$$V = n + n/2 + n/4 + \ldots + 1 = 2^* \cdot (1 + 1/2 + 1/4 + \ldots 1/2^k) = 2^{k+1} - 1 = 2 \cdot n - 1.$$

Из них все внутренние вершины порождают рекурсию, количество таких вершин — $V_r(n) = n - 1$, остальные п вершин -- это вершины, в которых рассматривается только один элемент массива, что приводит к останову рекурсии, следовательно, $V_1(\text{п}) = \text{п}$.

Таким образом, для п листьев дерева выполняется вызов процедуры *Merge-Sort,* в которой проверяется условие $(p = q)$ и выполняется возврат в вызывающую процедуру для слияния, что в сумме с учетом трудоемкости вызова дает

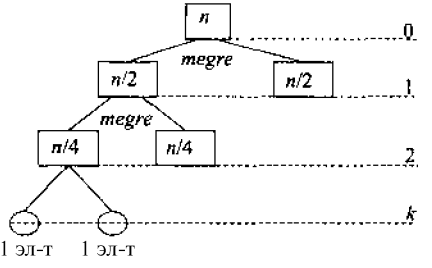


Рис. **Д.2.4.** Фрагмент рекурсивного дерева при сортировке слиянием

$$/1 = 2 \cdot (5 + 4 + 1) + 1 = 21; \quad \Rightarrow$$
$$/1 - V_1(\text{п}) = 21 \cdot n.$$

Для $n - 1$ рекурсивных вершин выполняется проверка длины переданного массива, вычисление середины массива и значения $r + 1$, рекурсивный вызов процедуры *Merge-Sort* и возврат. Поскольку трудоемкость вызова считается при входе в процедуру, то мы получаем

$$f_r = 2 \cdot (5 + 4 + 1) + (1 + 3 + 1) = 25; \qquad f_r \cdot V_r(\text{п}) = 25 \cdot n - 25.$$

Алгоритм слияния отсортированных массивов будет вызван п — 1 раз, при этом трудоемкость складывается из трудоемкости вызова и собственной трудоемкости алгоритма *Merge.* Трудоемкость вызова составит (для 6 параметров и 4 регистров):

$$f_{m\text{вызов}}(\text{п}) = V_r(n) \cdot 2 \cdot (6 + 4 + 1) = 22 \cdot \text{п} - 22.$$

Для анализируемого алгоритма сортировки функция $g(v_k)$ есть трудоемкость слияния в вершине v_k; заметим, что на фиксированном уровне рекурсивного дерева все соответствующие $g(v_k)$ одинаковы, так как слиянию подвергаются массивы одинаковой длины. Это значительно упрощает вычисление суммы

$$\sum_{k=1}^{V_r(n)} g(v_k),$$

поскольку одинаковые слагаемые, связанные с одним уровнем, могут быть объединены. Поскольку трудоемкость алгоритма слияния для массива длиной m составляет $18 \cdot m + 24$ (Д.2.7) и алгоритм вызывается $n-1$ раз с разными длинами массива на каждом уровне дерева, то значения имеют вид

$$\begin{cases} g(v_1) = 18 \cdot n + 24; \\ g(v_2) = g(v_3) = 18 \cdot (n/2) + 24; \\ g(v_4) = g(v_5) = 9(v_6) = 9(v_7) = 18 \cdot (n/4) + 24; \\ \ldots \end{cases}$$

Суммируя $g(v_k)$ по всем рекурсивным вершинам, имеем

$$f_{m\ \text{слияние}}(n) = \sum g(v_k) = 24 \cdot (n - 1) + 18 \cdot n + 2 \cdot 18 \cdot (n/2) + 4 \cdot 18 \cdot (n/4) + \ldots$$

Учитывая, что таким образом обрабатывается k уровней рекурсивного дерева с номерами от 0 до $k - 1$ (см. рис. Д.2.3.), получаем

$$f_{\text{т слияние (п)}} = 18 \cdot n \cdot A + 24 \cdot (n - 1) = 18 \cdot n \cdot \log_2 n + 24 \cdot (n - 1) = 18 \cdot n \cdot \log_2 n + 24 \cdot n - 24.$$

Учитывая все компоненты, получаем окончательный вид функции трудоемкости алгоритма сортировки слиянием:

$$f_A(n) = f_1 \cdot V_1(n) + f_r \cdot V_r(n) + f_{m\ \text{вызов}}(n) + f_{m\ \text{слияние}}(n) = 21 \cdot n + 25 \cdot n - 25 + 22 \cdot n - 22 + 18 \cdot n \cdot \log_2 n + 24 \cdot n - 24 = 18 \cdot n \cdot \log_2 n + 92 \cdot n - 71. \quad (Д.2.8)$$

Если количество чисел на входе алгоритма не равно степени двойки, то необходимо проводить более глубокий анализ, основанный на изучении поведения рекурсивного дерева для вычисления $\Sigma g(v_k)$, однако при любых ситуациях с данными формулы для $V_r(n)$ и $V_1(n)$ остаются в силе, равно как и оценка главного порядка $\Theta(n \cdot \log_2 n)$, полученная по теореме Бентли, Хакен, Сакса.

Д.2.9. Переход к временным оценкам

На основе функции трудоемкости возможно сравнение алгоритмов с целью выбора более эффективного при данном диапазоне длин входа и прогнозирование времен выполнения программных реализаций алгоритмов ([10]). Однако сравнение двух алгоритмов на основе функций трудоемкости расходится с результатами их сравнения по реально наблюдаемым временам выполнения. Основной причиной такого расхождения является различная частотная встречаемость элементарных операций, порождаемая разными алгоритмами, и различие во временах выполнения элементарных операций на реальном процессоре. Возникает задача перехода от функции трудоемкости к оценке времени работы алгоритма на конкретном процессоре – мы хотим определить оценку времени работы программной реализации алгоритма ($T_A(n)$)на основе знания $f_A(n)$ — функции трудоемкости алгоритма для худшего, лучшего или среднего случая.

На пути построения временных оценок мы сталкиваемся с целым набором различных проблем, учет которых вызывает существенные трудности, укажем кратко основные из них:

- особенности компилятора для выбранного языка программирования;
- временные задержки, вносимые операционной системой;
- особенности компьютера, на котором реализуется алгоритм, и, прежде всего, архитектура процессора и тактовые частоты процессора и системной шины;
- различные времена выполнения реальных машинных команд;

- различие во времени выполнения одной команды, в зависимости от значений операндов.

Попытки учета этих факторов привели к появлению различных способов перехода к временным оценкам.

Д.2.9.1. Способ пооперационного анализа

Идея пооперационного анализа состоит в получении пооперационной функции трудоемкости для каждой из используемых алгоритмом элементарных операций с учетом типов данных. Следующим шагом является экспериментальное определение среднего времени выполнения данной элементарной операции на конкретном компьютере в среде выбранного языка программирования и операционной системы. Ожидаемое время выполнения рассчитывается как сумма произведений пооперационной трудоемкости на средние времена операций:

$$T_A(n) = \sum f_{операции}(n) \cdot \bar{t}_{опер}.$$

Ошибка этого способа определяется в основном тем, насколько хорошо последовательность операций, задаваемых алгоритмом, загружает конвейер процессора и каким образом получены временные оценки выполнения элементарных операций.

Д.2.9.2. Способ среднего времени по типам задач

Способ предполагает получение функции трудоемкости и переход к временным оценкам на основе принадлежности задачи к одному из типов:

- задачи научно-технического характера с преобладанием операций с операндами действительного типа;
- задачи дискретной математики с преобладанием операций с операндами целого типа;
- задачи баз данных с преобладанием операций с операндами строкового типа.

Далее на основе анализа множества реальных программ для решения соответствующих типов задач определяется частотная встречаемость операций, и создаются соответствующие тестовые программы, и определяется среднее время на обобщенную операцию в данном типе задач — $\bar{t}_{типзадачи}$.

На основе полученной информации общее время работы программной реализации алгоритма может быть оценено в виде:

$$T_A(n) = f_A(n) \cdot \bar{t}_{типзадачи}.$$

Точность получаемых прогнозов определяется тем, насколько хорошо данный алгоритм согласуется по частотной встречаемости операций с тестовым набором, на основе которого было получено значение $\bar{t}_{типзадачи}$.

Д.2.9.3. Метод прямого определения среднего времени

В этом методе так же проводится совокупный анализ по трудоемкости — определяется $f_A(n)$, после чего экспериментально для различных значений $n_э$ определяется среднее время работы данной программы $T_э$, и на основе известной функции трудоемкости рассчитывается среднее время на обобщенную элементарную операцию, порождаемое данным алгоритмом, компилятором и компьютером — $\bar{t}_a$. Эти данные могут быть использованы (в предположении об устойчивости среднего времени по n) для прогнозирования времени выполнения для других значений размерности задачи следующим образом:

$$\bar{t}_A = \frac{T_e(n)}{f_A(n)}, \quad T(n) = f_A(n) \cdot \bar{t}_A.$$

Пример пооперационного анализа. В ряде случаев именно пооперационный анализ позволяет выявить тонкие аспекты рационального применения того или иного алгоритма решения задачи. В качестве примера рассмотрим задачу умножения двух комплексных чисел:

$$(a + b \cdot \text{г}) \cdot (c + d \cdot \text{г}) = (ac - bd) + \text{г} \cdot (ad + bc) = e + \text{г} \cdot f.$$

1) *Алгоритм A1 (прямое вычисление e, f за четыре умножения)*

```
MultComplex1(a,b, c, d; e, f)
  e←a*c  - b*d                    f_A1 = 8 операций
  f←a*d  + b*c                    /* = 4 операции
  Return (e,f)                    f± = 4 операции
End.
```

2) *Алгоритм A2(вычисление e, f за три умножения)*

```
MultComplex2(a,b, c, d; e, f)
  z1←c*(a + b)
  z2←b*(d + c)                    f_A2 = 13 операций
  z3←a*(d - c)                    f*— 3 операции
  e←z1 - z2                       /± = 5 операций
  f←z1+ z3
  Return (e,f)
End.
```

Пооперационный анализ этих двух алгоритмов не представляет труда, и его результаты приведены справа от записи соответствующих алгоритмов. По совокупному количеству элементарных операций алгоритм *A2* уступает алгоритму *A1*, однако в реальных компьютерах операция умножения требует большего времени, чем операция сложения, и можно путем пооперационного анализа ответить на вопрос: при каких условиях алгоритм *A2* предпочтительнее алгоритма *A1*?

Введем параметры q и r, устанавливающие соотношения между временами выполнения операции умножения, сложения и присваивания для операндов действительного типа. Тогда мы можем привести временные оценки двух алгоритмов

к времени выполнения операции сложения/вычитания — t_+: $t_* = q \cdot t+$, $q > 1$; $t_{\leftarrow} = r \cdot t_+$, $r < 1$, тогда временные оценки, приведенные к t_+, имеют вид:

$$T_{A1} = 4 \cdot q \cdot t_+ + 2 \cdot t_+ + 2 \cdot r \cdot t_+ = t_+ \cdot (4 \cdot q + 2 + 2 \cdot r);$$
$$T_{A2} = 3 \cdot q \cdot t_+ + 5 \cdot t_+ + 5 \cdot r \cdot t_+ = t_+ \cdot (3 \cdot q + 5 + 5 \cdot r).$$

Равенство времен будет достигнуто при условии: $4 \cdot q + 2 + 2 \cdot r = 3 \cdot q + 5 + 5 \cdot r$, откуда $q = 3 + 3 \cdot r$ и, следовательно, при $q > 3 + 3 \cdot r$ алгоритм *A2* будет работать более эффективно.

Таким образом, если среда реализации алгоритмов *A1* и *A2* — язык программирования, обслуживающий его компилятор, и компьютер на котором реализуется задача, такова, что время выполнения операции умножения двух действительных чисел более чем втрое превышает время сложения двух действительных чисел, в предположении, что $r \ll 1$, а это реальное соотношение, то для реализации более предпочтителен алгоритм *A2*. Конечно, выигрыш во времени пренебрежимо мал, если мы перемножаем только два комплексных числа, однако, если этот алгоритм является частью сложной вычислительной задачи с комплексными числами, требующей значительного количества умножений, то выигрыш во времени может быть ощутим. Оценка такого выигрыша на одно умножение комплексных чисел следует из только что проведенного анализа:

$$ДГ = (q - 3 - 3 \cdot r) \cdot t_+$$

Д.2.10. Оценка ресурсной эффективности алгоритмов

При использовании алгоритмов для решения практических задач мы сталкиваемся с проблемой рационального выбора алгоритма решения задачи. Решение этой проблемы связано с построением системы сравнительных оценок, на основании которых мы выбираем более предпочтительный алгоритм. Нас могут интересовать различные свойства алгоритма, например точность решения задачи, но, в основном, речь идет о ресурсной эффективности алгоритма, под которой принято понимать требуемый объем оперативной памяти и время выполнения программной реализации. В рамках построения оценки ресурсной эффективности алгоритма будем считать, что:

- каждая элементарная операция выполняется не более чем за фиксированное время;
- исходные данные алгоритма представляются машинными словами по β битов каждое.

Конкретная проблема задается п словами памяти, таким образом, на входе алгоритма содержится $n_\beta = n \cdot \beta$ *бит* информации. Отметим, что в ряде случаев, особенно при рассмотрении матричных задач, значение п является мерой длины входа алгоритма, отражающей линейную размерность задачи в машинных словах. Программа,

реализующая алгоритм решения задачи, состоит из m машинных инструкций по β_m битов. Обозначим ее длину через m_β, таким образом, длина программы составляет $m_\beta = m \cdot \beta_m$ *бит* информации. Кроме того, алгоритм может требовать следующих дополнительных ресурсов:

- памяти для хранения промежуточных результатов — S_d;
- памяти для организации вычислительного процесса (память, необходимая для реализации рекурсивных вызовов и возвратов) — S_r.

При решении конкретной проблемы, заданной n словами памяти, алгоритм выполняет не более чем конечное количество «элементарных» операций в силу условия рассмотрения только финитных алгоритмов. Анализ ресурсной эффективности алгоритма может быть выполнен на основе комплексной оценки ресурсов компьютера, требуемых алгоритмом для решения задачи, как функции размерности входа. Очевидно, что для различных областей применения алгоритма веса ресурсов будут различны, что приводит к следующей оценке ресурсной эффективности алгоритма:

$$\Psi_A(\text{п}) = c_1 \cdot T_A(n) + c_2 \cdot m_\beta + c_3 \cdot S_d + c_4 \cdot S_r,$$

где Ci — веса ресурсов.

Конкретные значения коэффициентов c_i задают удельные стоимости ресурсов, определяемые условиями применения алгоритма и спецификой программной системы. Выбор рационального алгоритма A_r может быть осуществлен при заданных значениях коэффициентов a по критерию минимума комплексной оценки $\psi_A(n)$, рассчитанной для всех претендующих алгоритмов из множества A. Пусть множество претендующих алгоритмов A состоит из m элементов:

$$A = \{ A_i | i = \overline{1, m} \},$$

тогда

$$A_r(n) : \Psi_{A_r}(\text{п}) = \min_A \{ \Psi_{A_i}(D_A) \},$$

где минимум берется по всем претендующим алгоритмам из множества A, а запись $A_r(n)$ обозначает рациональный алгоритм для длины входа п.

Из условий эксплуатации разрабатываемой программной системы обычно известно, что данная задача будет решаться на определенном сегменте размерностей входа. Например, для системы автоматизации работы отдела кадров можно указать границы изменения численности работающих на этом предприятии. Для выбора рационального алгоритма на этом сегменте определим функцию $R(n)$, значением которой является номер алгоритма r, для которого

$$R(\text{п}) = r \Leftrightarrow A_r(n) : \Psi_{A_r}(\text{п}) = \min_{A_i} \{ \Psi_{A_i} \}, \quad r = 1, \text{ т.}$$

Тогда на исследуемом сегменте размерностей задачи — $[n_1, n_2]$, отражающем особенности применения проектируемой программной системы, возможны следующие случаи:

а) функция $R(n)$ принимает одинаковые значения на $[n_1, n_2]$:

$$R(n_i) = R(n_j) = k, \qquad \forall n_i, n_j \in [n_1, n_2],$$

таким образом, алгоритм с номером k является рациональным по комплексному критерию $\psi_A(n)$ на всем сегменте размерностей входа;

б) функция $R(n)$ принимает различные значения на $[n_1, n_2]$:

$$R(n_i) \neq R(n_j), \qquad n_i, n_j \in [n_1, n_2],$$

тем самым несколько алгоритмов являются рациональными по критерию $\psi_A(n)$ для различных размерностей входа на сегменте $[n_1, n_2]$. В этом случае на сегменте $[n_1, n_2]$ можно выделить подсегменты, в которых значение *R(n) — const,* и реализовать в программной системе адаптивный по размеру входа выбор рационального алгоритма в зависимости от длины входа.

Литература к разделу Д.2.

[1] Ульянов М. В. Классификации и методы сравнительного анализа вычислительных алгоритмов. — М.: Издательство физико-математической литературы, 2004 г. — 212 с.

[2] Гудман С., Хидетниеми С., Введение в разработку и анализ алгоритмов. • — М.: Мир, 1981. — 368 с.

[3] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 2001 г. — 960 с.

[4] Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы: Пер. с англ.: — М.: Издательский дом «Вильяме», 2001 г. — 384 с.

[5] Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — 2-ое изд., испр. — СПб.: Невский диалект, 2001. - 352 с., ил.

[6] Хопкрофт Дж., Мотовани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений, 2-ое изд.: Пер. с англ. — М.: Издательский дом «Вильяме»; 2002. — 528 с.

[7] Чмора А.Л. Современная прикладная криптография. - М.: Гелиос АРВ, 2001. - 256 с., ил.

[8] Грин Д., Кнут Д. Математические методы анализа алгоритмов. — М.: Мир, 1987. — 120с., ил.

[9] Новиков Ф. А. Дискретная математика для программистов. - СПб.: Питер, 2001. - 304 с., ил.

[10] Ульянов М.В. Метод прогнозирования временных оценок программных реализаций алгоритмов на основе функции трудоемкости // Информационные технологии. 2004. №5. с. 54-62.

Глава Д.3.
Идеи современных алгоритмов

Настоящий раздел содержит краткое изложение некоторых идей, на основе которых разрабатываются современные алгоритмы. Подчеркнем тот факт, что сегодня наиболее интересные результаты в разработке эффективных алгоритмов связаны с привлечением аппарата других научных дисциплин, в частности — биологии и теоретических результатов специальных разделов современной математики.

Д.3.1. Алгоритмы и простые числа

В не очень далеком прошлом теория простых чисел считалась разделом чисто теоретической математики, однако современные алгоритмические идеи решения разнообразных задач опираются на результаты этой теории. Значительное повышение интереса к простым числам в области информатики связано с известной криптосистемой RSA (Риверст, Шамир, Адлеман, 1977г. [1]). При построении этой криптосистемы используется ряд алгоритмов, в том числе и алгоритм Евклида (III в. до н. э.), который до сих пор остается «современным» и востребованным в алгоритмической практике.

В целях дальнейшего изложения приведем некоторые сведения и обозначения из теории простых чисел.

Д.3.1.1. Сравнения

Говорят, что два целых числа a и b сравнимы по модулю с, если они дают при делении на c равные остатки. Операция получения остатка от деления а на с записывается в виде: a $\bmod c = d$, что эквивалентно представлению: $a = k \cdot c + d$, $d > 0$, $k > 0$ — целые числа. Сравнимость двух чисел по модулю c означает, что o $\bmod c = b \bmod c$ и записывается в виде $a = b \pmod c$. Если $a = 0 \pmod c$, то число a делится на число c без остатка.

Д.3.1.2. Простые числа

Число p называется простым, если оно не имеет других делителей, кроме единицы и самого себя. Очевидно, что при проверке в качестве возможных делителей есть смысл проверять только простые числа, меньшие или равные квадратному корню из проверяемого числа. Множество простых чисел счетно, доказательство принадлежит Евклиду. Пусть $p_1, \ldots, p_k$, — простые числа, и p_k — последнее из них, но тогда число $a = (p_1 \cdot p_2 \cdot \ldots \cdot p_{k+1})$ в силу основной теоремы арифметики должно разлагаться, и притом, единственно в произведение простых. Но число a не делится нацело ни на одно из pi, и мы пришли к противоречию.

Д.3.1.3. Функция $\pi(n)$

Функция $\pi(n)$ в теории простых чисел обозначает количество простых чисел, не превосходящих n, например, $\pi(12) = 5$, т.к. существует 5 простых чисел не

превосходящих 12. Асимптотическое поведение функции $\pi(n)$ было исследовано в конце XIX века. В 1896 году Адамар и Валле-Пуссен ([2]) доказали, что

$$\pi(n) \approx \frac{n}{\ln(n)}, \quad \lim_{n\to\infty} \pi(n) \cdot \frac{\ln(n)}{n} = 1.$$

Полученный результат означает, что простые числа не так уж «редки», мы получаем оценку $1/(\ln n)$ для вероятности того, что случайно взятое число, не превосходящее n, является простым.

Д.3.1.4. Теорема Ферма

Введем операцию умножения чисел по модулю n, $a \cdot b \bmod n$ и определим степени числа: $a^2 = (a \cdot a) \bmod n$, $a^{k+1} = (a^k \cdot a) \bmod n$. Для любого простого числа p и числа a, $0 < a < p$, справедлива малая теорема Ферма (теорема Ферма-Эйлера): $a^{p-1} = 1 \pmod p$ ([1]).

Д.3.1.5. Алгоритм Евклида

Алгоритм Евклида асимптотически оптимально решает задачу нахождения наибольшего общего делителя двух чисел НОД(a, b). Заметим, что задача нахождения НОД является сегодня составной частью многих эффективных алгоритмов ([2, 3]). Обоснование правильности этого алгоритма основано на свойствах НОД. Если d = = НОД(a,b), *то* d есть наименьший положительный элемент множества целочисленных линейных комбинаций чисел a и 6 — $(a \cdot x + b \cdot y)$, где x, y — целые числа; можно показать, что $a \bmod b$ является целочисленной линейной комбинацией a и 6, поэтому НОД(a, b) = НОД$(b, a \bmod b)$ ([2]).

Приведем рекурсивную реализацию алгоритма Евклида.

```
Ec(a, b);
If b = 0
  then Ec = a;
  else Ec = Ec(b, a mod b);
end.
```

Алгоритм Евклида принадлежит классу *PR* — его трудоемкость определяется значениями чисел. Лучшим случаем при фиксированной битовой длине чисел, очевидно, является ситуация, когда $b = 0$. Заметим, что в ситуации, когда $a < b$, алгоритм первой рекурсией переворачивает пару чисел. Анализ в худшем случае не так очевиден — если $a > b \geq 0$, и $b < F_{k+1}$, где F_{k+1} — $(k + 1)$-ое число Фибоначчи, то процедура Ec(a, b) выполняет менее k рекурсивных вызовов (теорема Ламе, начало XIX века [2]). Из этой теоремы можно получить сложность алгоритма Евклида — поскольку $F_k \approx \varphi^k$, где

$$\varphi = \left(1 + \sqrt{5}\right)/2 \approx 1{,}618$$

(if — величина, обратная к «золотому сечению»), то количество рекурсивных вызовов не превышает $1 + \log_\varphi b$ ([2]). Если мы будем рассматривать числа о и 6

как двоичные β-битовые числа, то можно показать, что процедура `Ec(a, b)` имеет сложность $O(\beta^2)$ битовых операций ([2]).

Д.3.1.6. Вероятностный тест Миллера–Рабина

На практике, особенно в криптографических алгоритмах, возникает необходимость нахождения больших простых чисел. Один из эффективных алгоритмов решения этой задачи — вероятностный тест Миллера–Рабина (1980 г.).

Идея теста Миллера–Рабина состоит в последовательной генерации случайных чисел и проверке их на простоту с использованием теоремы Ферма - Эйлера. Основная задача теста связана с уменьшением вероятности ошибочного признания некоторого составного числа в качестве простого. Проблема состоит в том, что существуют составные числа п (числа Кармайкла), для которых сравнение $a^{n-1} = 1 \pmod n$ справедливо для всех целых чисел a, удовлетворяющих неравенству $0 < a < n$. Решением проблемы является проверка на нетривиальный корень из 1 в момент вычисления a^{n-1} методом последовательного возведения в квадрат (см. Д.2.5.), т. е. проверка того, что $(x \cdot x) \bmod n = 1$, при $x \neq 1$, $x \neq n - 1$. Отсутствие нетривиальных корней свидетельствует в пользу того, что число n является простым ([2]).

Тест Миллера - Рабина.

1. Цикл пока не будет найдено простое число
2. Генерация случайного числа п (двоичное **β-битовое** число)
3. Повторять s раз
4. Случайный выбор числа $a \in \{2, \ldots, n-2\}$
   если $(a^{n-1}) \bmod n \neq 1$, то, очевидно, что число п составное;
   если $(a^{n-1}) \bmod n = 1$, то, необходимо проверить другое а;
   при вычислении $(a^{n-1}) \bmod n$ проверить нетривиальный корень из 1
   если $(x \cdot x) \bmod n = 1$ при $x \neq 1$, $x \neq n-1$, то число n — **составное.**
5. Конец цикла по s
6. Выход, если все выбранные числа a показали простоту числа п.
7. Конец теста.

Вероятность ошибки теста экспоненциально падает с ростом успешных проверок с различными значениями а, а именно, если выполнено s успешных проверок, то она составляет 2^{-s}, что приводит реально к выбору s в пределах нескольких десятков. Сложность теста оценивается как $O(s \cdot n^3)$ битовых операций ([2]).

Д.3.1.7. Алгоритм Рабина – Карпа

Одна из нетривиальных идей использования простых чисел для разработки и доказательства эффективности алгоритмов — идея предложенного в 1981 и опубликованного в 1987 году Рабином и Карпом алгоритма поиска подстроки в строке ([4]). Центральная идея алгоритма связана с переходом к сравнению чисел вместо посимвольного сравнения образца и части строки поиска.

Пусть дана символьная строка Г длиной m (m-битовая строка) и образец P длиной n. Нам необходимо найти все вхождения образца в строку. Обозначим через T_r подстроку длиной п, начиная с бита r. Определим

$$H(P) = \underbrace{\overline{\quad}}_{i=1}^{n} 2^{n-i} \cdot P(i), \quad H(T_r) = \underbrace{\overline{\quad}}_{i=1}^{n} 2^{n-i} \cdot T(r + \text{i-1}).$$

Мы рассматриваем тем самым подстроки как двоичные числа, очевидно, что образец P входит в строку T, начиная с позиции r, если Я(Р) — $H(T_r)$. Таким образом, мы сравниваем числа, а не символы, и если вычисление $H(T_r)$ при сдвиге может быть выполнено за $O(1)$, то сложность алгоритма составит $O(n + m)$, т.е. теоретическую границу сложности задачи поиска. Принципиальная проблема этой идеи — экспоненциальный рост чисел $H(P)$ и $H(T_r)$, с ростом длины образца. При обычном байтовом кодировании символов образец всего из 12 букв занимает 96 бит, и, следовательно, Я(Р) $< 2^{96} - 1$. Необходимо было сохранить внешнюю привлекательность идеи, но уложиться в рамки реального диапазона представления целых чисел (машинного слова) в компьютере для образцов любой длины.

Решение проблемы, предложенное Рабином и Карпом, состоит в использовании остатков от деления $H(P)$ и $H(T_r)$ на некоторое простое число p. Эти остатки называются дактилограммами ([4]). Но действительная мощность предложенного метода состоит в доказательстве того, что вероятность ошибки (ложного срабатывания) может быть сделана малой, если простое число выбирается случайно из некоторого множества. Обозначим через

$$H_p(\text{Р}) = \text{Я}(\text{Р}) \bmod p, \qquad H_p(T_r) = \text{Я}(T_r) \bmod p.$$

Однако если вычислять вначале $H(P)$, а затем брать остаток, то мы снова окажемся за пределами машинного слова. Для эффективного вычисления $H_p(P)$ и $H_p(T_r)$ может быть использована схема Горнера ([2]), в результате чего во время вычислений ни один из промежуточных результатов не будет больше $2 \cdot p$. Схема вычислений имеет вид

$$Hp(P) =$$
$$= ((((P(1) \cdot 2 \bmod p + P(2)) \cdot 2 \bmod p + P(3)) \cdot 2 \bmod p) + \ldots + P(n)) \bmod p.$$

Следующая задача на пути создания эффективного алгоритма — быстрое вычисление сдвига значения $H(T_r)$. Заметим, что

$$\text{Я}(T_r) = 2 \cdot \text{Я}(T_{r-1}) - T \cdot T(r - 1) + T(r + \text{п} + 1),$$

и, выполняя вычисления по $\bmod p$, мы можем вычислить $H_p(T_r)$ по $H_p(T_{r-1})$ с трудоемкостью $O(1)$ ([4]). Очевидно, что $H_p(P)$ и $H_p(T_1)$ вычисляются однократно со сложностью $O(n)$. При каждом сдвиге по строке мы вычисляем $H_p(T_r)$ и сравниваем числа, что дает по порядку $O(m-n)$, и мы получаем общую линейную сложность. Обоснование того, что вероятность ложного совпадения (возможно, что числа Я(Р) и $H(T_r)$ различны, в то время как $H_p(P)$ и $H_p(T_r)$ совпадают) мала, происходит на

основе теорем и лемм теории простых чисел. Центральная теорема подхода Рабина–Карпа формулируется следующим образом ([4]).

Теорема Д.3.1. Пусть P и T некоторые строки, причем $n \cdot m > 29$, где $n = |P|$, $m = |T|$. Пусть I — некоторое положительное число. Если p — *случайно* выбранное простое число, не превосходящее I, то вероятность ложного совпадения P и T не превосходит $\pi(n \cdot m)/\pi(I)$. Если выбрать $I = n \cdot m^2$, то вероятность ложного совпадения не превосходит $2,53/m$.

Алгоритм Рабина–Карпа

1. Выбрать положительное целое I (например, $I = n \cdot m^2$).
2. Случайно выбрать простое число p, не превосходящее I (например, используя тест Миллера{ Рабина).
3. По схеме Горнера вычислить $H_p(P)$ и $H_p(T_1)$.
4. Для каждой позиции r в T $(1 < r < m - n + 1)$ вычислить $H_p(T_r)$ и сравнить с $H_p(P)$. Если они равны, то либо объявить о вероятном совпадении, либо выполнить явную проверку, начиная с позиции r.

Существует несколько модификаций метода Рабина–Карпа, связанных с обнаружением ошибок. Один из них основан на методе Бойера-Мура (см. 5.1.3. основного текста) и со сложностью $O(m)$ либо подтверждает отсутствие ложных совпадений, либо декларирует, что, по меньшей мере, одно из совпадений ложное ([4]). В случае ложных совпадений нужно выполнять алгоритм заново с новым значением p, вплоть до отсутствия ложных совпадений. Тем самым алгоритм Рабина–Карпа преобразуется из метода с малой вероятностью ошибки и со сложностью $O(m + n)$ в худшем случае в метод, который не делает ошибок, но обладает *ожидаемой* сложностью $O(m + n)$ -- это преобразование алгоритма Монте-Карло в алгоритм Лас-Вегаса (см. 9.2. основного текста).

Отметим, что метод Рабина–Карпа успешно применяется (наряду с целым рядом других алгоритмов поиска подстрок) в таком разделе современной науки как вычислительная молекулярная биология ([4]), для поиска совпадающих цепочек ДНК и расшифровки генов.

Д.3.2. Генетические алгоритмы

Практическая необходимость решения ряда NP-полных задач в оптимизационной постановке при проектировании и исследовании сложных систем привела разработчиков алгоритмического обеспечения к использованию биологических механизмов поиска наилучших решений. В настоящее время эффективные алгоритмы разрабатываются в рамках научного направления, которое можно назвать «природные вычисления» ([5]), объединяющего такие разделы, как генетические алгоритмы, эволюционное программирование, нейросетевые вычисления ([6]), клеточные автоматы и ДНК-вычисления ([4]), муравьиные алгоритмы. Исследователи обращаются к природным механизмам, которые миллионы лет обеспечивают адаптацию биоценозов к окружающей среде. Одним из таких механизмов, имеющих фундамен-

тальный характер, является механизм наследственности. Его использование для решения задач оптимизации привело к появлению генетических алгоритмов.

В живой природе особи в биоценозе конкурируют друг с другом за различные ресурсы, такие, как пища или вода. Кроме того, особи одного вида в популяции конкурируют между собой, например, за привлечение брачного партнера. Те особи, которые более приспособлены к окружающим условиям, будут иметь больше шансов на создание потомства. Слабо приспособленные либо не произведут потомства, либо их потомство будет очень немногочисленным. Это означает, что гены от высоко приспособленных особей будут распространяться в последующих поколениях. Комбинация хороших характеристик от различных родителей иногда может приводить к появлению потомка, приспособленность которого даже больше, чем приспособленность его родителей. Таким образом, вид в целом развивается, лучше и лучше приспосабливаясь к среде обитания.

Д.3.2.1. Введение в генетические алгоритмы

Алгоритм решения задач оптимизации, основанный на идеях наследственности в биологических популяциях, был впервые предложен Джоном Холландом (1975 г.). Он получил название репродуктивного плана Холланда, и широко использовался как базовый алгоритм в эволюционных вычислениях. Дальнейшее развитие эти идеи, как собственно и свое название — генетические алгоритмы, получили в работах Гольдберга и Де Йонга [7].

Цель генетического алгоритма при решении задачи оптимизации состоит в том, чтобы найти лучшее возможное, но не гарантированно оптимальное решение. Для реализации генетического алгоритма необходимо выбрать подходящую структуру данных для представления решений. В постановке задачи поиска оптимума, экземпляр этой структуры должен содержать информацию о некоторой точке в пространстве решений.

Структура данных генетического алгоритма состоит из набора хромосом. Хромосома, как правило, представляет собой битовую строку, так что термин строка часто заменяет понятие «хромосома». Вообще говоря, хромосомы генетических алгоритмов не ограничены только бинарным представлением. Известны другие реализации, построенные на векторах вещественных чисел ([8]). Несмотря на то, что для многих реальных задач, видимо, больше подходят строки переменной длины, в настоящее время структуры фиксированной длины наиболее распространены и изучены.

Для иллюстрации идеи мы ограничимся только структурами, которые являются битовыми строками. Каждая хромосома (строка) представляет собой последовательное объединение ряда подкомпонентов, которые называются генами. Гены расположены в различных позициях или локусах хромосомы, и принимают значения, называемые аллелями — это биологическая терминология. В представлении хромосомы бинарной строкой, ген является битом этой строки, локус — есть позиция бита в строке, а аллель — это значение гена, 0 или 1. Биологический термин «генотип» относится к полной генетической модели особи и соответствует структуре в генетическом алгоритме. Термин «фенотип» относится к внешним наблюдаемым

признакам и соответствует вектору в пространстве параметров задачи. В генетике под *мутацией* понимается преобразование хромосомы, случайно изменяющее один или несколько генов. Наиболее распространенный вид мутаций — случайное изменение только одного из генов хромосомы. Термин *кроссинговер* обозначает порождение из двух хромосом двух новых путем обмена генами. В литературе по генетическим алгоритмам также употребляется термин кроссовер, скрещивание или рекомбинация. В простейшем случае кроссинговер в генетическом алгоритме реализуется так же, как и в биологии. При скрещивании хромосомы разрезаются в случайной точке и обмениваются частями между собой. Например, если хромосомы (11, 12, 13, 14) и (О, О, О, 0) разрезать между вторым и третьим генами и обменять их части, то получатся следующие потомки (11, 12, 0, 0) и (О, О, 13, 14).

Основные структуры и фазы генетического алгоритма. Приведем простой иллюстративный пример ([8]) — задачу максимизации некоторой функции двух переменных $f(x_1, x_2)$, при ограничениях: $0 < x_1 < 1$ и $0 < x_2 < 1$. Обычно методика кодирования реальных переменных x_1 и x_2 состоит в преобразовании их в двоичные целочисленные строки определенной длины, достаточной для того, чтобы обеспечить желаемую точность. Предположим, что 10-ти разрядное кодирование достаточно для x_1 *и* x_2. Установить соответствие между генотипом и фенотипом можно, разделив соответствующее двоичное целое число на 2^{10} - 1. Например, 0000000000 соответствует 0/1023 или 0, тогда как 1111111111 соответствует 1023/1023 или 1.

Оптимизируемая структура данных есть 20-ти битовая строка, представляющая собой конкатенацию (объединение) кодировок x_1 и x_2. Пусть переменная x_1 размещается в крайних левых 10-ти битах строки, тогда как x_2 размещается в правой части генотипа особи. Таким образом, генотип представляет собой точку в 20-ти мерном целочисленном пространстве (вершину единичного гиперкуба), которое исследуется генетическим алгоритмом. Для этой задачи фенотип будет представлять собой точку в двумерном пространстве параметров (x_1, x_2).

Чтобы решить задачу оптимизации нужно задать некоторую меру качества для каждой структуры в пространстве поиска. Для этой цели используется функция приспособленности. При максимизации целевая функция часто сама выступает в качестве функции приспособленности, для задач минимизации целевая функция инвертируется и смещается в область положительных значений.

Рассмотрим фазы работы простого генетического алгоритма ([8]). В начале случайным образом генерируется начальная популяция (набор хромосом). Работа алгоритма представляет собой итерационный процесс, который продолжается до тех пор, пока не будет смоделировано заданное число поколений или выполнен некоторый критерий останова. В каждом поколении реализуется пропорциональный отбор приспособленности, одноточечная рекомбинация и вероятностная мутация. Пропорциональный отбор реализуется путем назначения каждой особи (хромосоме) *i* вероятности *P(i),* равной отношению ее приспособленности к суммарной приспособленности популяции (по целевой функции):

$$P(i) = \frac{f(i)}{\sum_{i=1}^{n} f(i)}$$

Затем происходит отбор (с замещением) всех n особей для дальнейшей генетической обработки, согласно убыванию величины P(г'). Простейший пропорциональный отбор реализуется с помощью рулетки (Гольдберг, 1989 г.). «Колесо» рулетки содержит по одному сектору для каждого члена популяции, а размер i-ого сектора пропорционален соответствующей величине $P(i)$. При таком отборе члены популяции, обладающие более высокой приспособленностью, будут выбираться чаще по вероятности.

После отбора выбранные n особей подвергаются рекомбинации с заданной вероятностью P_c, при этом п хромосом случайным образом разбиваются на $n/2$ пар. Для каждой пары с вероятность P_c может быть выполнена рекомбинация. Если рекомбинация происходит, то полученные потомки заменяют собой родителей. Одноточечная рекомбинация работает следующим образом. Случайным образом выбирается одна из точек разрыва, т. е. участок между соседними битами в строке. Обе родительские структуры разрываются на два сегмента по этому участку. Затем соответствующие сегменты различных родителей склеиваются и получаются два генотипа потомков.

После стадии рекомбинации выполняется фаза мутации. В каждой строке, которая подвергается мутации, каждый бит инвертируется с вероятностью P_m. Популяция, полученная после мутации, записывается поверх старой, и на этом завершается цикл одного поколения в генетическом алгоритме.

Полученное новое поколение обладает (по вероятности) более высокой приспособленностью, наследованной от «хороших» представителей предыдущего поколения. Таким образом, из поколения в поколение, хорошие характеристики распространяются по всей популяции. Скрещивание наиболее приспособленных особей приводит к тому, что исследуются наиболее перспективные участки пространства поиска. В результате популяция будет сходиться к локально оптимальному решению задачи, а иногда, может быть, благодаря мутации, и к глобальному оптимуму.

Теперь мы можем сформулировать основные шаги генетического алгоритма.

Генетический алгоритм

1. Создать начальную популяцию
2. Цикл по поколениям пока не выполнено условие останова
   //цикл жизни одного поколения
3. Оценить приспособленность каждой особи
4. Выполнить отбор по приспособленности
5. Случайным образом разбить популяцию на две группы пар
6. Выполнить фазу вероятностной рекомбинации для пар **популяции** и заменить родителей
7. Выполнить фазу вероятностной мутации
8. Оценить приспособленность новой популяции и вычислить условие останова
9. Объявить потомков новым поколением
10. Конец цикла по поколениям

Модификации генетического алгоритма. Очевидно, что тонкая настройка базового генетического алгоритма может быть выполнена путем изменения значений

вероятностей рекомбинации и мутации, существует много исследований и предложений в данной области, более подробно этот вопрос освещен в [6, 7, 9, 10].

В настоящее время предлагаются разнообразные модификации генетических алгоритмов в части методов отбора по приспособленности, рекомбинации и мутации ([6, 8, 10]). Приведем несколько примеров ([8]).

Метод турнирного отбора (Бриндел, 1981 г.; Гольдберг и Деб, 1991 г.) реализуется в виде п турниров для выборки n особей. Каждый турнир состоит в выборе k элементов из популяции и отбора лучшей особи среди них. Элитные методы отбора (Де Йонг, 1975 г.) гарантируют, что при отборе обязательно будут выживать лучший или лучшие члены популяции. Наиболее распространена процедура обязательного сохранения только одной лучшей особи, если она не прошла через процесс отбора, рекомбинации и мутации. Этот метод может быть внедрен практически в любой стандартный метод отбора.

Двухточечная рекомбинация (Гольдберг 1989 г.) и равномерная рекомбинация (Сисверда, 1989 г.) являются вполне достойными альтернативами одноточечному оператору. При двухточечной рекомбинации выбираются две точки разрыва, и родительские хромосомы обмениваются сегментом, который находится между двумя этими точками. Равномерная рекомбинация предполагает, что каждый бит первого родителя наследуется первым потомком с заданной вероятностью; в противном случае этот бит передается второму потомку.

Механизмы мутаций могут быть так же заимствованы из молекулярной биологии, например, обмен концевых участков хромосомы (механизм транслокации), обмен смежных сегментов (транспозиция) ([4]). По мнению автора, интерес представляет механизм инверсии, т. е. перестановки генов в хромосоме, управляющим параметром при этом может выступать инверсионное расстояние — минимальное количество единичных инверсий генов, преобразующих исходную хромосому в мутированную ([4]).

Применение генетических алгоритмов. Основная проблема, связанная с применением генетических алгоритмов — это их эвристический характер. Говоря более строго, какова вероятность достижения популяцией глобального оптимума в заданной области при данных настройках алгоритма? В настоящее время не существует строгого ответа и теоретически обоснованных оценок. Имеются предположения, что генетический алгоритм может стать эффективной процедурой поиска оптимального решения, если:

- пространство поиска достаточно велико, и предполагается, что целевая функция не является гладкой и унимодальной в области поиска, т. е. не содержит один гладкий экстремум;
- задача не требует нахождения глобального оптимума, необходимо достаточно быстро найти приемлемое «хорошее» решение, что довольно часто встречается в реальных задачах.

Если целевая функция обладает свойствами гладкости и унимодальности, то любой градиентный метод, такой как метод наискорейшего спуска, будет более эффективен. Генетический алгоритм является в определенном смысле универсальным методом, т. е. он явно не учитывает специфику задачи или должен быть на нее

каким-то образом специально настроен. Поэтому если мы имеем некоторую дополнительную информацию о целевой функции и пространстве поиска (как, например, для хорошо известной задачи коммивояжера), то методы поиска, использующие эвристики, определяемые задачей, часто превосходят любой универсальный метод.

С другой стороны, при достаточно сложном рельефе функции приспособленности градиентные методы с единственным решением могут останавливаться в локальном решении. Наличие у генетических алгоритмов целой «популяции» решений, совместно с вероятностным механизмом мутации, позволяют предполагать меньшую вероятность нахождения локального оптимума и большую эффективность работы на многоэкстремальном ландшафте.

Сегодня генетические алгоритмы успешно применяются для решения классических *NP*-полных задач, задач оптимизации в пространствах с большим количеством измерений, ряда экономических задач оптимального характера, например, задач распределения инвестиций.

Д.3.3. Муравьиные алгоритмы

Муравьиные алгоритмы представляют собой новый перспективный метод решения задач оптимизации, в основе которого лежит моделирование поведения колонии муравьев. Колония представляет собой систему с очень простыми правилами автономного поведения особей. Однако, несмотря на примитивность поведения каждого отдельного муравья, поведение всей колонии оказывается достаточно разумным. Эти принципы проверены временем — удачная адаптация к окружающему миру на протяжении миллионов лет означает, что природа выработала очень удачный механизм поведения. Исследования в этой области начались в середине 90-х годов XX века, автором идеи является Марко Дориго из Университета Брюсселя, Бельгия ([11, 12, 13]).

Д.3.3.1. Биологические принципы поведения муравьиной колонии

Муравьи относятся к социальным насекомым, образующим коллективы. В биологии коллектив муравьев называется колонией. Число муравьев в колонии может достигать нескольких миллионов, на сегодня известны суперколонии муравьев (Formica lugubrus), протянувшиеся на сотни километров. Одним из подтверждений оптимальности поведения колоний является тот факт, что сеть гнезд суперколоний близка к минимальному остовному дереву (см. 6.4.) графа их муравейников ([5]).

Основу поведения муравьиной колонии составляет самоорганизация, обеспечивающая достижения общих целей колонии на основе низкоуровневого взаимодействия. Колония не имеет централизованного управления, и ее особенностями является обмен локальной информацией только между отдельными особями (прямой обмен — пища, визуальные и химические контакты) и наличие непрямого обмена, который и используется в муравьиных алгоритмах.

Непрямой обмен — *стигмержи* (stigmergy), представляет собой разнесенное во времени взаимодействие, при котором одна особь изменяет некоторую область

окружающей среды, а другие используют эту информацию позже, в момент, когда они в нее попадают. Биологи установили, что такое отложенное взаимодействие происходит через специальное химическое вещество — *феромон* (pheromone), секрет специальных желез, откладываемый при перемещении муравья. Концентрация феромона на тропе определяет предпочтительность движения по ней. Адаптивность поведения реализуется испарением феромона, который в природе воспринимается муравьями в течение нескольких суток. Мы можем провести некоторую аналогию между распределением феромона в окружающем колонию пространстве, и «глобальной» памятью муравейника, носящей динамический характер ([5]).

Д.3.3.2. Идея муравьиного алгоритма

Идея муравьиного алгоритма — моделирование поведения муравьев, связанное с их способностью быстро находить кратчайший путь от муравейника к источнику пищи и адаптироваться к изменяющимся условиям, находя новый кратчайший путь. При своем движении муравей метит свой путь феромоном, и эта информация используется другими муравьями для выбора пути. Это элементарное правило поведения и определяет способность муравьев находить новый путь, если старый оказывается недоступным. Дойдя до преграды, муравьи с равной вероятностью будут обходить ее справа и слева. То же самое будет происходить и на обратной стороне преграды. Однако, те муравьи, которые случайно выберут кратчайший путь, будут быстрее его проходить, и за несколько передвижений он будет более обогащен феромоном. Поскольку движение муравьев определяется концентрацией феромона, то следующие будут предпочитать именно этот путь, продолжая обогащать его феромоном, до тех пор, пока этот путь по какой-либо причине не станет доступен. Очевидная положительная обратная связь быстро приведет к тому, что кратчайший путь станет единственным маршрутом движения большинства муравьев. Моделирование испарения феромона — отрицательной обратной связи, гарантирует нам, что найденное локально оптимальное решение не будет единственным — муравьи будут искать и другие пути. Если мы моделируем процесс такого поведения на некотором графе, ребра которого представляют собой возможные пути перемещения муравьев, в течение определенного времени, то наиболее обогащенный феромоном путь по ребрам этого графа и будет являться решением задачи, полученным с помощью муравьиного алгоритма. Рассмотрим конкретный пример.

Д.3.3.3. Формализация задачи коммивояжера в терминах муравьиного подхода

Задача формулируется как задача поиска минимального по стоимости замкнутого маршрута по всем вершинам без повторений на полном взвешенном графе с n вершинами. Содержательно вершины графа являются городами, которые должен посетить коммивояжер, а веса ребер отражают расстояния (длины) или стоимости проезда. Эта задача является NP-трудной, и точный переборный алгоритм ее решения имеет факториальную сложность. Приводимое здесь описание муравьиного алгоритма для задачи коммивояжера является кратким изложением статьи С. Д. Штовбы ([5]).

Моделирование поведения муравьев связано с распределением феромона на тропе — ребре графа в задаче коммивояжера. При этом вероятность включения ребра в маршрут отдельного муравья пропорциональна количеству феромона на этом ребре, а количество откладываемого феромона пропорционально длине маршрута. Чем короче маршрут, тем больше феромона будет отложено на его ребрах, следовательно, большее количество муравьев будет включать его в синтез собственных маршрутов. Моделирование такого подхода, использующего только положительную обратную связь, приводит к преждевременной сходимости -- большинство муравьев двигается по локально оптимальному маршруту. Избежать этого можно, моделируя отрицательную обратную связь в виде испарения феромона. При этом если феромон испаряется быстро, то это приводит к потере памяти колонии и забыванию хороших решений, с дугой стороны, большое время испарения может привести к получению устойчивого локально оптимального решения.

Теперь, с учетом особенностей задачи коммивояжера, мы можем описать локальные правила поведения муравьев при выборе пути.

1. Муравьи имеют собственную «память». Поскольку каждый город может быть посещен только один раз, у каждого муравья есть список уже посещенных городов — список запретов. Обозначим через $J_{i,k}$ список городов, которые необходимо посетить муравью k, находящемуся в городе г.

2. Муравьи обладают «зрением»— видимость есть эвристическое желание посетить город j, если муравей находится в городе г. Будем считать, что видимость обратно пропорциональна расстоянию между городами

$$\eta_{ij} = 1/D_{ij}.$$

3. Муравьи обладают «обонянием»— они могут улавливать след феромона, подтверждающий желание посетить город j из города г, на основании опыта других муравьев. Количество феромона на ребре (г, j) в момент времени t обозначим через $\tau_{ij}(t)$.

4. На этом основании мы можем сформулировать вероятностно-пропорциональное правило ([5]), определяющее вероятность перехода fc-ого муравья из города $г$ в город j:

$$\begin{cases} P_{ij,k}(t) = \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum\limits_{l \in J_{i,k}} [\tau_{il}(t)]^{\alpha} \cdot [\eta_{il}]^{\beta}}, & j \in J_{i,k}; \\ P_{ij,k}(t) = 0, & a\ J_{i,k}, \end{cases} \qquad (Д.3.1)$$

где а, β — параметры, задающие веса следа феромона, при $a = 0$ алгоритм вырождается до жадного алгоритма (будет выбран ближайший город). Заметим, что выбор города является вероятностным, правило (Д.3.1) лишь определяет ширину зоны города j; в общую зону всех городов $J_{i,k}$ бросается случайное число, которое и определяет выбор муравья. Правило (Д.3.1) не изменяется в ходе алгоритма, но у двух разных муравьев значение вероятности перехода будут отличаться, т.к. они имеют разный список разрешенных городов.

5. Пройдя ребро (г, j) муравей откладывает на нем некоторое количество феромона, которое должно быть связано с оптимальностью сделанного выбора. Пусть

$T_k(t)$ есть маршрут, пройденный муравьем k к моменту времени t, $L_k(t)$ — длина этого маршрута, а Q — параметр, имеющий значение порядка длины оптимального пути. Тогда откладываемое количество феромона может быть задано в виде

$$\Delta\tau_{ij,k}(t) = \begin{cases} \dfrac{Q}{L_k(t)}, & (i,j) \in T_k(t); \\ 0, & (i,j) \notin T_k(t). \end{cases}$$

Правила внешней среды определяют, в первую очередь, испарение феромона. Пусть $p \in [0,1]$ есть коэффициент испарения, тогда правило испарения имеет вид

$$\text{ту}(t+1) = (1-\text{р}) \cdot \text{ту}(t) + \text{Дту}(t); \quad \text{Дту}(\text{t}) = \sum_{k=1}^{\text{га}} 53\,\Delta\tau_{ij,k}(t), \qquad (\text{Д.3.2})$$

где m — количество муравьев в колонии.

В начале алгоритма количество феромона на ребрах принимается равным небольшому положительному числу. Общее количество муравьев остается постоянным и равным количеству городов, каждый муравей начинает маршрут из своего города.

Дополнительная модификация алгоритма может состоять во введении так называемых «элитных» муравьев, которые усиливают ребра наилучшего маршрута, найденного с начала работы алгоритма. Обозначим через T^* наилучший текущий маршрут, через L^* — его длину. Тогда если в колонии есть e элитных муравьев, ребра маршрута получат дополнительное количество феромона

$$\Delta\tau_e = e \cdot Q/L^*. \qquad (\text{Д.3.3})$$

***Муравьиный** алгоритм для задачи коммивояжера*

1. Ввод матрицы расстояний D.
2. Инициализация параметров **алгоритма**——α, β, **e**, **Q**.
3. Инициализация ребер——**присвоение** видимости η_{ij} и начальной концентрации феромона.
4. Размещение муравьев в случайно выбранные города без совпадений.
5. Выбор начального кратчайшего маршрута и определение L^* // основной **цикл**.
6. Цикл по времени жизни колонии **t=1,tmax**.
   7. Цикл по всем муравьям **k=1,m**
      8. Построить маршрут $T_k(t)$ по правилу **(Д.3.1)** и рассчитать длину $L_k(t)$.
   9. конец цикла по муравьям.
   10. Проверка всех $L_k(t)$ на лучшее решение по сравнению с L^*.
      11. Если да, то обновить L^* и T^*.
   12. Цикл по всем **ребрам** графа.
      13. Обновить следы феромона на ребре по правилам **(Д.3.2)** и (Д.3.3).
   14. конец цикла по ребрам.

15. конец цикла по **времени**.
16. Вывести кратчайший маршрут T^* и его длину L^*.

Сложность данного алгоритма определяется непосредственно из приведенного выше текста — $O(t_{max}, n^2, m)$, таким образом, сложность алгоритма зависит от времени жизни колонии (t_{max}), количества городов (n) и количества муравьев в колонии (т).

Д.3.3.4. Области применения и возможные модификации

Поскольку в основе муравьиного алгоритма лежит моделирование передвижения муравьев по некоторым путям, то такой подход может стать эффективным способом поиска рациональных решений для задач оптимизации, допускающих графовую интерпретацию. Ряд экспериментов показывает, что эффективность муравьиных алгоритмов растет с ростом размерности решаемых задач оптимизации. Хорошие результаты получаются для нестационарных систем с изменяемыми во времени параметрами, например, для расчетов телекоммуникационных и компьютерных сетей ([5]). В [14] описано применение муравьиного алгоритма для разработки оптимальной структуры съемочных сетей GPS, в рамках создания высокоточных геодезических и съемочных технологий. В настоящее время на основе применения муравьиных алгоритмов получены хорошие результаты для таких сложных оптимизационных задач, как задача коммивояжера, транспортная задача, задача календарного планирования, задача раскраски графа, квадратичной задачи о назначениях, задачи оптимизации сетевых графиков и ряда других ([5]).

Качество получаемых решений во многом зависит от настроечных параметров в вероятностно-пропорциональном правиле выбора пути на основе текущего количества феромона и параметров правил откладывания и испарения феромона. Возможно, что динамическая адаптационная настройка этих параметров может способствовать получению лучших решений. Немаловажную роль играет и начальное распределение феромона, а также выбор условно оптимального решения на шаге инициализации.

В [5] отмечается, что перспективными путями улучшения муравьиных алгоритмов является адаптация параметров с использованием базы нечетких правил и их гибридизация, например, с генетическими алгоритмами. Как вариант, такая гибридизация может состоять в обмене, через определенные промежутки времени, текущими наилучшими решениями.

Много полезной информации по муравьиным алгоритмам читатель может найти на специальных англоязычных сайтах по этому направлению ([13, 15]).

Литература к разделу Д.3.

[1] Коутинхо С. Введение в теорию чисел. Алгоритм RSA. — М.: Постмаркет, 2001 г. — 328с.
[2] Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. — М.: МЦНМО, 2001 г. — 960 с.
[3] Ноден П., Китте К. Алгебраическая алгоритмика: Пер.с франц. — М.: Мир, 1999 г. — 720 с.

[4] Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология / Пер с англ. И.В. Романовского. — СПб.: Невский диалект; БХВ-Петербург, 2003 г. — 654 с.
[5] Штовба С.Д. Муравьиные алгоритмы // Exponenta Pro Математика в приложениях. 2003. 4. С.70–75.
[6] Рутковский Л., Пилиньский М., Рутковская Д. Нейронные сети, генетические алгоритмы и нечеткие системы. — М.: Горячая линия-Телеком, 2004 г. — 452 с.
[7] http://www.iissvit.narod.ru/ssilki.htm.
[8] http://www.krf.bsu.by/ELib/Genetic/GenAlg_2/index.htm.
[9] http://www.neuroproject.ru.
[10] http://www.aic.nrl.navy.mil/galist/.
[11] Bonavear E., Dorigo M. Swarm Intelligence: from Natural to Artificial Systems. -— Oxford University Press, 1999, — 307 p.
[12] Corne D., Dorigo M., Glover F. New Ideas in Optimization. — McGrav-Hill, 1999.
[13] http://iridia.ulb.ac.be/dorigo/ACO/ACO.html.
[14] http://www.agp.ru/projects/.
[15] http://www.swarm.org.

КНИГИ ИЗДАТЕЛЬСТВА "ТЕХНОСФЕРА" МОЖНО ПРИОБРЕСТИ В ГОРОДАХ:

г. Москва

Торговый дом "Библио-Глобус", ул. Мясницкая, 6
Торговый дом книги "Москва", ул. Тверская, 8
"Московский дом книги", ул. Новый Арбат, 8
"Дом технической книги", Ленинский проспект, 40
"Молодая гвардия", ул. Б. Полянка, 28
"Дом книги на Соколе", Ленинградский проспект, 78, к. 1
"Дом педагогической книги", ул. Б. Дмитровка, 7/5, стр. 1
Магазин "Математическая книга", Б. Власьевский пер., 11
"Читай-Город", ул. Новослободская, 21
"Дом книги в Медведково", Заревый проезд, 12
"Дом книг у Красных Ворот", ул. Садовая-Черногрязская, 51
"Дом медицинской книги", Комсомольский проспект, 25
"Букбери", Торгово-развлекательный центр "Мега" Калужской шоссе
Торговый центр "Глобал Сити" ул. Кировоградская, 14
Никитский бульвар, 17, стр. 1
"Библиосфера", ул. Марксистская, 9

Московская область

г. Долгопрудный

Физматкнига, Институтский пер., 6а

г. Зеленоград

"Алекс и К", Панфиловский проспект, к. 1106-б

Города России

г. Санкт-Петербург

"Санкт-Петербургский дом книги", Невский пр., 28
"Дом технической книги", Пушкинская пл., 2

г. Новосибирск

ООО "Топ книга", ул. Захарова, 103

г. Воронеж

"Книжный мир семьи", пр. Революции, 58

г. Омск

"Техническая книга"

г. Волгоград

"Техническая книга", ул. Мира, 11

Заявки на книги присылайте по адресу:
125319 Москва, а/я 594
Издательство «Техносфера»
e-mail: knigi@technosphera.ru
sales@technosphera.ru
факс: (095) 956 33 46

В заявке обязательно указывайте
свой почтовый адрес!

Подробная информация о книгах на сайте
http://www.technosphera.ru

Дж. **Макконелл**
Основы современных алгоритмов
2-е дополненное издание

Компьютерная верстка — Е.А. Зарубина, С.А. Кулешов
Дизайн книжных серий — С.Ю. Биричев
Ответственный за выпуск — **Л.Ф.** Соловейчик

Формат 70 x 100/16. Печать **офсетная.**
Гарнитура Computer modern **LaTeX.**
Печ.л. 23. Тираж 2000 **экз.Зак.** № 2129
Бумага офсет №1, плотность 65 г/м.^{2}

Издательство «Техносфера»
Москва, ул. Тверская, дом 10 строение 3

Диапозитивы изготовлены ООО **«Европолиграфик»**
Отпечатано в ООО **ПФ** «Полиграфист»
160001 г. Вологда, ул. Челюскинцев, дом 3