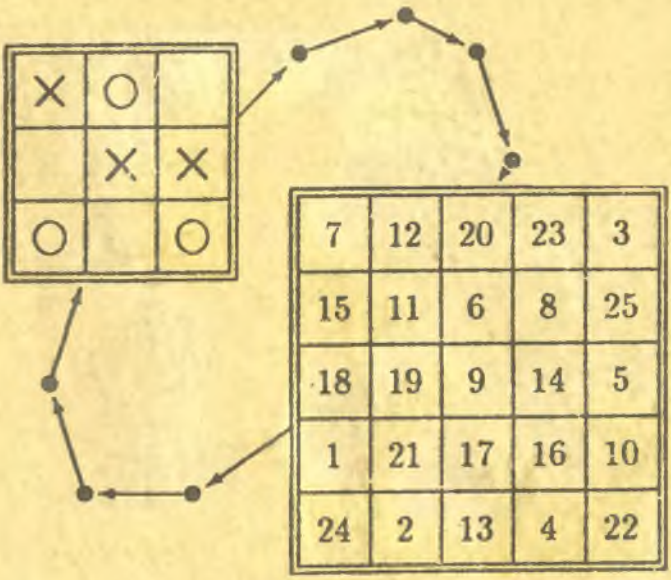
Ю. Л. Костюк

ОСНОВЫ АЛГОРИТМИЗАЦИИ



Томский государственный университет

Ю. Л. Костюк

ОСНОВЫ АЛГОРИТМИЗАЦИИ

Учебное пособие

Издательство Томского университета

Томск – 1996

УДК 681.3

Костюк Ю. Л. Основы алгоритмизации: Учебное пособие. — Томск: Изд-во Том. ун-та, 1996. — 124 с. — 1000 экз. — 1602020000.

Излагаются методы исследования и доказательства правильности алгоритмов. Приводятся и исследуются некоторые важные классы алгоритмов. Среди них: вычисление рекуррентных последовательностей; сортировка и поиск; рекурсивные алгоритмы, включая решение головоломок; обработка текста; шифрование; трансляция; геометрические алгоритмы. Особое внимание уделено эффективности алгоритмов.

Алгоритмы записаны на языке Паскаль.

Книга будет полезна преподавателям информатики, школьникам, студентам младших курсов, а также программистам, желающим глубже понять природу алгоритмов.

Рецензент — профессор А. Е. Мудров

ISBN 5 - 7511 - 0821 - 3

К $\frac{1602020000}{177(012)-96}$

© Ю. Л. Костюк, 1996

Предисловие

Самым главным в профессии программиста является умение создавать хорошие программы. Все же остальное (знание языка программирования, транслятора, операционной системы, умение быстро работать за компьютером) необходимо лишь постольку, поскольку помогает писать хорошие алгоритмы и, тем самым, разрабатывать хорошие программы.

К сожалению, большинство учебников по программированию мало внимания уделяет *алгоритмизации* — науке об алгоритмах. Эта книга предназначена для тех, кто уже познакомился с языком программирования, умеет работать за компьютером и успел понять, что этого мало для настоящего программиста.

В книге используется язык Паскаль, который хорош как для начинающих, так и опытных программистов. Для чтения книги достаточно знать такие основные элементы Паскаля, как:

1) целые, вещественные и символьные типы данных;

2) одно- и двумерные массивы, символьные строки;

3) присваивания и формулы, арифметические операции;

4) условные операторы **if** и **case**, операции сравнения и логические операции;

5) циклы **while** и **for**;

6) процедуры и функции, их описание, вызов, подстановка параметров;

7) стандартные процедуры и функции.

В настоящее время используется много разных компьютеров и еще больше различных трансляторов Паскаля. Большинство алгоритмов можно реализовать на любом из них. Однако в некоторых алгоритмах (их совсем немного) приходится учитывать специфику конкретного транслятора. Такие алгоритмы в книге написаны для транслятора Turbo Pascal 5.5 и более поздних версий.

При написании книги ставились следующие цели:

1) научить читателя методам проверки алгоритмов, в том числе аналитическому доказательству правильности;

2) научить методам исследования качества алгоритмов;

3) познакомить со многими интересными и важными классами алгоритмов, предназначенных для решения разных задач;

4) научить разрабатывать новые алгоритмы из рассмотренных классов.

От читателя требуется знание элементарной математики: алгебры, геометрии, а также элементарной логики (изучаемой в школьном курсе информатики). Основные рассматриваемые алгоритмы базируются на идеях *дискретной* математики — фундамента информатики. В то же время в книге намеренно обойдены вопросы построения *численных* алгоритмов, например, приближенные вычисления, решения систем уравнений и т.п. Дело в том, что как раз эти задачи довольно широко представлены в литературе по программированию.

Первая глава посвящена проблеме аналитического доказательства правильности алгоритмов и исследованию их качества. Без этого невозможно научиться создавать хорошие алгоритмы. Последующие главы посвящены изучению различных классов алгоритмов, от самых простых до более сложных. Для многих алгоритмов приводится подробное доказательство и исследование. В принципе книгу можно использовать просто как сборник полезных алгоритмов, пропустив все доказательства, однако такой метод ее изучения мало что даст читателю для роста профессионального уровня.

Каждая глава снабжена заданиями для самостоятельной работы. Желательно большинство из них реализовать на компьютере и испытать на тестовых данных.

Приведенные в книге алгоритмы имеются в компьютерном варианте на дискете. Они рассчитаны на Turbo Pascal в операционной системе MS DOS. На этой же дискете располагаются алгоритмы заданий с пояснениями, а также дополнительные задачи. О приобретении дискеты справляться по телефону (382-2) 23-24-53 или письменно по адресу:

634050, г. Томск, пр. Ленина, 36, ТГУ, факультет информатики.

Просьба по указанному адресу присылать отзывы и замечания по книге.

Глава 1. РАЗРАБОТКА АЛГОРИТМОВ И ПРОВЕРКА ИХ ПРАВИЛЬНОСТИ

Природа алгоритмов и сложна и проста одновременно. С алгоритмами человек сталкивается всюду в своей повседневной жизни: любое сколько-нибудь сложное действие человека, которое можно разделить на этапы, является алгоритмом. Однако сама попытка строгого выделения этапов, их осмысления, выявления всех сопутствующих условий у неподготовленного человека (у непрограммиста!), как правило, вызывает затруднения. Иными словами, большинство людей понимают алгоритмы лишь интуитивно.

В обыденной жизни этого достаточно: если действия "алгоритма" выполняет человек, то он, руководствуясь здравым смыслом и своим опытом, по ходу дела уточняет детали и в конце концов получает задуманный результат. Однако программист должен рассчитывать на механического вычислителя, лишенного фантазии, для которого "понятны" только абсолютно определенные, формализованные алгоритмы. Что же значит создать алгоритм?

Во-первых, для этого необходимо определить *исполнителя* — раба, робота, называйте как угодно, — того, кто будет выполнять действия, записанные в алгоритме. Важно лишь, чтобы любое действие исполнитель понимал строго однозначно и исполнял его абсолютно точно по определению этого действия. Например, исполнитель, который необходим нам, — это компьютер с языком Паскаль, точнее, с транслятором Паскаля и средством выполнения паскалевских программ — операционной системой.

Во-вторых, программист должен вычленить отдельные действия и связать их между собой таким образом, чтобы выполняющий эти действия исполнитель получил результат, задуманный программистом. Как правило, алгоритм должен быть *универсальным*, т.е. таким, чтобы для него можно было задавать *различные* входные данные. Обычно количество всевозможных входных данных огромно, все их невозможно перебрать просто физически. Несмотря на

это, программист должен быть уверен, что при *любых допустимых* входных данных алгоритм вычислит *правильный* результат.

1.1. Тестирование

Как показывает опыт разработки алгоритмов, даже при весьма аккуратной и тщательной работе в первоначально составленном алгоритме неизбежны ошибки. Обычно чем больше алгоритм (чем длиннее его запись) и чем сложнее его логика, тем больше бывает в нем ошибок. Из-за этих ошибок исполнитель, действуя буквально, не в состоянии получить ожидаемый результат, т.е. алгоритм оказывается неправильным. Конечно, если бы исполнитель обладал фантазией, он, быть может, иногда и сумел бы догадаться, что ожидается от алгоритма и в чем состоит ошибка, но такого "исполнителя" пока никто не придумал.

Поэтому важное значение в разработке алгоритма имеет проверка правильности. Обычно такую проверку осуществляют путем пробного запуска алгоритма на исполнение. Пробный запуск состоит из двух этапов: 1) трансляции, при которой выявляются синтаксические ошибки в формальной записи алгоритма; 2) собственно выполнения, когда делаются вычисления по конкретным входным данным. Для хорошей проверки алгоритма второй этап необходимо многократно повторять с *различными* вариантами входных данных. Эти варианты входных данных называют тестами, а сам второй этап — тестированием.

Как правило, всех возможных тестов столь много, что при тестировании нельзя испытать алгоритм на всех или сколько-нибудь заметной части общего количества тестов. Пусть, например, требуется проверить алгоритм умножения двух 10-значных неотрицательных десятичных чисел. Количество различных значений для каждого сомножителя равно 10^{10}, а количество различных пар сомножителей — 10^{20}. Даже если выполнять по 10^6 тестов в секунду, для полного тестирования потребуется более трех миллионов лет! Если вы испытали свой алгоритм на ста тестах и все 100 раз получали правильный результат, нет никакой гарантии, что правильный результат вы получите и в сто первый раз. На основе подобных рассуждений Э.Дейкстра сформулировал закон: *тестированием можно доказать наличие ошибок в программе, но никогда — их отсутствие.*

К счастью, на практике ситуация не столь безнадежная. Существует наука, называемая *технологией программирования,* которая дает рекомендации относительно того, как разрабатывать и как проверять алгоритмы. Оказывается, что если тесты тщательно подбирать, а не брать наугад, то можно значительно улучшить ситуацию. Прежде всего следует учесть, что задача тестирования состоит в том, что *тест должен выявлять максимум ошибок!* Тогда для обнаружения (и последующего исправления) большей части ошибок в программе можно обойтись малым числом тестов.

Если при выполнении какого-либо теста обнаружен неправильный результат, то поиск причины ошибки может оказаться непростым делом, ведь требуется полностью переосмыслить алгоритм! В этом случае часто помогает *бумажное тестирование,* когда тест, давший ошибочный результат на компьютере, повторно выполняют вручную. При этом на бумаге последовательно выписывают значения переменных на каждом шаге вычислений. Если проделать такую работу аккуратно, то можно вовремя обнаружить, когда некоторая переменная получит неправильное значение, и тогда можно догадаться о причине ошибки. Конечно, такую работу можно выполнить лишь для тестов очень небольшой размерности, поэтому при обнаружении ошибочных результатов для большого теста необходимо попытаться подобрать такой маленький тест, чтобы на нем проявилась аналогичная ошибка, и только потом проводить бумажное тестирование.

Многие трансляторы позволяют выполнять программу в режиме отладки по шагам, причем на каждом шаге можно увидеть значение переменных. И тогда можно сравнить значения переменных, полученные в ходе бумажного тестирования, с значениями, полученными на компьютере, это даст еще одну возможность поиска ошибки в алгоритме.

Тем не менее полной гарантии, что в оттестированной программе отсутствуют ошибки, конечно же, нет: закон Э.Дейкстры верен!

1.2. Аналитическое доказательство правильности

Казалось бы, выхода нет и мы должны смириться с тем, что в любой программе всегда есть необнаруженные ошибки. Но Э.Дейкстрой и рядом других великих программистов был разработан принципиально иной метод доказательства правильности алго-

ритмов — аналитический, который позволяет проверить правильность алгоритма не для одного конкретного набора входных значений, как при обычном тестировании, а для бесконечного количества наборов входных значений. Поэтому такой метод в принципе может решить проблему правильности алгоритмов. Конечно, этот метод сложнее простого тестирования, поэтому его можно рекомендовать тому, кто достаточно хорошо умеет программировать, умеет грамотно подбирать тесты и, кроме того, не боится математических рассуждений. Начинающему программисту в первый раз лучше пропустить разделы 1.2 и 1.3, а при чтении следующих разделов можно пропускать доказательства.

Идея доказательства правильности заключается в следующем. Любой алгоритм в процессе своего исполнения использует и изменяет значения переменных. Для таких переменных составляют два логических условия, называемых *предусловием* и *постусловием*. Предполагается, что предусловие должно быть истинным *перед* исполнением алгоритма, а постусловие — *после* исполнения.

Доказательство правильности содержит два этапа:

1) доказательство *завершимости* алгоритма, т.е. доказательство того, что при любых значениях переменных на входе алгоритма он когда-нибудь закончит свое выполнение;

2) доказательство *истинности постусловия* после завершения алгоритма.

Для проведения доказательства следует знать свойства отдельных операторов алгоритма, которые определяются языком программирования.

Доказательство завершимости бывает очень простым, почти очевидным, например, если алгоритм состоит из последовательности операторов присваивания или из цикла **for**. В то же время иногда приходится прибегать к громоздким рассуждениям, например для цикла **while**, особенно если переменные, входящие в условие этого цикла, изменяются сложным образом. В любом случае необходимо определить, при *каком условии* алгоритм завершится: это условие будет частью *постусловия*.

Доказательство истинности постусловия после завершения алгоритма, как правило, не очевидно. Для его проведения используют следующие методы:

1) *перечисление вариантов*, когда вся область значений пере-

менных, входящих в предусловие, разбивается на несколько частей, и тогда для каждой части доказательство проводится отдельно;

2) *метод математической индукции*, широко используемый в математике;

3) *инвариант*, когда из предусловия и постусловия выделяется общая, неизменяемая часть;

4) *метод эквивалентов* используют, когда есть два разных алгоритма с одинаковыми пред- и постусловиями, и тогда для доказательства правильности второго алгоритма доказывают правильность первого алгоритма и эквивалентность первого и второго;

5) *абстракция*, применяемая при доказательстве сложных алгоритмов.

Применение различных методов доказательства рассмотрим на нескольких простых примерах.

Алгоритм 1.1. Пусть для числовых переменных x, y выполнено предусловие $x > 0$, $y > 0$. (В такой записи запятая подразумевает логическую связку "и".) Требуется доказать, что после выполнения алгоритма

if $x < y$ **then begin** $z := y$; $y := x$; $x := z$ **end**.

будет справедливым постусловие $x \geq y > 0$.

▷ **Доказательство.** Завершимость алгоритма очевидна. Доказательство истинности постусловия проведем перечислением двух вариантов:

1) $x \geq y$; условие оператора **if** будет ложным, поэтому больше никаких действий в алгоритме выполняться не будет; из этого условия и предусловия следует справедливость постусловия;

2) $x < y$, условие оператора **if** будет истинным, поэтому в алгоритме будут выполнены присваивания, в результате которых переменные x и y обменяются своими значениями, что приведет к смене условия $x < y$ на противоположное, т.е. на $x \geq y$, а из этого условия и предусловия снова следует справедливость постусловия.

Во втором случае предполагалось известным то, что операторы $z := y$; $y := x$; $x := z$ обменивают значениями переменные x и y. Докажем это. Предусловие: $x = a$, $y = b$. После первого присваивания $z = b$, $x = a$, $y = b$, после второго: $z = b$, $x = a$, $y = a$, после третьего: $z = b$, $x = b$, $y = a$, что и требовалось доказать. ◁

Доказательство правильности — мощное средство для обнаружения ошибок в алгоритме. Предположим, что присваивания в алгоритме 1.1 записаны в виде $x := y$; $y := x$. Тогда после первого присваивания $x = b$, $y = b$, после второго *ничего не изменилось*, т.е. вместо обмена значениями получилось присваивание обеим переменным первоначального значения переменной y.

А теперь рассмотрим метод математической индукции. Метод позволяет доказывать утверждение, зависящее от *параметра индукции* — целого числа. Утверждение доказывается для всех значений параметра, бо́льших или равных некоторому начальному значению.

Доказательство методом математической индукции состоит из трех этапов: 1) базиса, 2) предположения, 3) вывода.

Этап базиса заключается в проверке того, что доказываемое утверждение $P(n)$ относительно параметра индукции n справедливо при $n = n_0$.

На втором этапе делается предположение, что *утверждение $P(n)$ справедливо для всех значений n, не больших, чем $k \geq n_0$.*

Наконец, на третьем этапе доказывается, что $P(n)$ будет справедливо при $n = k + 1 > n_0$. Из этого следует, что утверждение $P(n)$ справедливо для *любого* $n \geq n_0$.

Действительно, проверим, что утверждение $P(n)$ справедливо при некотором $n = M$, таком, что $M > n_0$. Вначале положим $k = n_0$. Согласно третьему этапу доказательства утверждение будет справедливо при $n = n_0 + 1$. Применив еще раз третий этап доказательства при $n = n_0 + 1$, получим справедливость утверждения $P(n)$ при $n = n_0 + 2$ и т.д., пока не дойдем до $n = M$.

Проиллюстрируем применение метода математической индукции на примере алгоритма вычисления факториала.

Алгоритм 1.2. Вычисление факториала n. Доказать, что если справедливо предусловие $n = M$, $M \geq 0$, то на выходе алгоритма

$f = 1$; **for** $i := 1$ **to** n **do** $f := f * i$;

будет справедливо постусловие $f = M!$.

▷ **Доказательство правильности.** Завершимость алгоритма очевидна, так как цикл **for** при границах изменения параметра цикла от 1

до $n \geq 0$ исполнится ровно n раз. Справедливость постусловия докажем методом математической индукции

Базис. Если $M = 0$, то после того, как переменная f примет значение 1, больше ничего выполняться не будет (цикл от 1 до 0 не проработает ни разу). Как известно, $0! = 1$, т.е. базис справедлив.

Предположение. Предполагаем, что при $M = k$, $k \geq 0$ справедливо постусловие $f = k!$.

Вывод. Все отличие выполнения алгоритма при $M = k + 1$ от выполнения при $M = k$ состоит в том, что вначале алгоритм выполнится в точности так же, как при $M = k$, а затем цикл будет *обязательно* исполнен еще один раз для $i = k + 1$. Так как по предположению перед последним выполнением цикла $f = k!$, то после последнего выполнения цикла $f = k!(k + 1) = (k + 1)! = M!$, что и требовалось доказать. ◁

Метод доказательства с помощью инварианта состоит в следующем. Пусть предусловие записано как $(P \textbf{ and } Q)$, постусловие — как $(S \textbf{ and } Q)$, т.е. в них имеется общая часть Q, которая и называется инвариантом. Доказательство состоит в проверке истинности инварианта после выполнения алгоритма. Обычно инвариант применяют при доказательстве правильности оператора цикла. Если доказать, что однократное исполнение цикла сохраняет истинность инварианта, то отсюда и из завершимости цикла следует истинность инварианта *после* завершения цикла.

Применение инварианта рассмотрим на примере вычисления факториала, но алгоритм немного изменим.

Алгоритм 1.3. Вычисление факториала n. Доказать, что если после выполнения начальных присваиваний $i := 1$; $f := 1$; справедливо предусловие $n = M$, $M \geq 0$, $i = 1$, $f = (i - 1)!$, то после выполнения алгоритма

while $i <= n$ **do begin** $f := f * i$; $i := i + 1$ **end**

будет справедливо постусловие $i = M + 1$, $f = (i - 1)!$.

▷ **Доказательство.** Завершимость здесь также очевидна, так как до выполнения цикла $i = 1$, $n \geq 0$, внутри цикла i увеличивается на 1, а когда цикл закончится, $i = n + 1$, т.е. цикл проработает ровно n раз.

Инвариантом здесь является условие $f = (i - 1)!$. Действительно, после начальных присваиваний инвариант справедлив, а после однократного

исполнения цикла $f' = f * i = i!$ $i' = i + 1$, где f' и i' обозначают новые значения переменных f и i.

После окончания цикла $i = n + 1$, откуда следует, что $f = M!$. ◁

Доказательство методом инварианта предполагает инвариант *заданным*. Но как же его задавать? Общих рецептов для этого нет, здесь часто необходима догалка. Например, для алгоритма вычисления факториала нужно было догадаться, что постусловие $i = n + 1$, $f = n!$ можно представить в виде $i = M + 1$, $f = (i - 1)!$, а предусловие $i = 1$, $f = 1$ — в виде $i = 1$, $f = (i - 1)!$.

Применение метода эквивалентов рассмотрим на примере алгоритмов 1.2 и 1.3 вычисления факториала. Так как алгоритмы хоть немного различаются, то для них эквивалентность можно доказать *относительно* отдельных переменных. Докажем эквивалентность алгоритмов 1.2 и 1.3 относительно переменной f. А именно докажем, что для обоих алгоритмов при любом $n \geq 0$ значения этой переменной будут одинаковы после завершения алгоритмов.

▷ **Доказательство.** Вначале докажем, что циклы в обоих алгоритмах выполнятся *одинаковое* число раз. Для первого алгоритма цикл **for** при границах изменения параметра цикла от 1 до $n \geq 0$ исполнится ровно n раз. Для второго алгоритма до выполнения цикла $i = 1$, а при каждом выполнении после всех других действий в конце цикла i увеличивается на единицу. Поэтому цикл закончится после того, как $i = n + 1$, т.е. цикл проработает также n раз.

А теперь докажем, что после завершения алгоритмов переменные f в них будут равны. Используем метод математической индукции.

Базис. Если $n = 0$, циклы в обоих алгоритмах ни разу не выполнятся, $f = 1$.

Предположение. Предполагаем, что при $n = k \geq 0$ значения переменной f в обоих алгоритмах равны между собой.

Вывод. Все отличие выполнения алгоритмов при $n = k + 1$ от выполнения при $n = k$ состоит в том, что вначале каждый из алгоритмов выполнится в точности так же, как при $n = k$, а затем цикл будет *обязательно* исполнен еще один раз для $i = k + 1$. Так как по предположению перед последним выполнением цикла f в обоих алгоритмах равны, то при последнем выполнении каждого из циклов переменные f умножаются на одинаковые значения $i = k + 1$, т.е. остаются равными.

Теперь, если известно доказательство правильности алгоритма 1.2, то отсюда следует правильность алгоритма 1.3. ◁

Применение метода абстракции для доказательства правильности сложных алгоритмов будет объяснено в разделе 1.4.

На практике аналитическое доказательство не заменяет, а дополняет машинное тестирование. Имеется ряд причин, по которым нельзя ограничиться только аналитическим доказательством.

Во-первых, очень трудно учесть все ограничения, связанные с конкретным представлением данных в компьютере. Так, например, при использовании в качестве целых переменных двухбайтовых машинных слов правильный результат в алгоритме 1.2 получится, только если $0 \leq n \leq 7$, так как для значения $8! = 40320$ уже недостаточно 16 бит. Если использовать четырехбайтовые целые, то можно правильно вычислить $12!$, однако $13! = 6227020800$ правильно вычислить не удастся. Подобные ограничения сложно учесть потому, что они должны быть справедливы как для окончательного результата, так и *для всех промежуточных значений переменных.*

Во-вторых, при выполнении реального алгоритма на компьютере трудно заранее учесть все нюансы взаимодействия алгоритма со стандартными программами, операционной системой.

В-третьих, в доказательстве могут быть ошибки!

Поэтому только при сочетании тестирования и аналитического доказательства правильности можно добиться того, чтобы алгоритм был действительно безошибочным.

Следует также отметить, что наибольшая практическая польза от доказательства состоит в том, что оно заставляет программиста писать ясные и понятные алгоритмы.

1.3. Доказательство правильности простых алгоритмов

Рассмотрим правила построения пред- и постусловий для основных конструкций языка Паскаль. Обозначим через S некоторый алгоритм, через P — предусловие, а через R — постусловие для него. Сокращенно мы это будем записывать так:

$$\{P\}\ S\ \{R\}.$$

1. Присваивание. Если имеется некоторое условие $P(x)$, в запись которого входит переменная x, то

$$\{P(w)\}\ x := w\ \{P(x)\},$$

где $P(w)$ есть то же самое условие, в котором каждое вхождение переменной x заменено на правую часть присваивания w.

2. Последовательность операторов. Если $S1$ и $S2$ — операторы, для которых выполняются условия:

$$\{P\}\ S1\ \{Q\}\ \text{и}\ \{Q\}\ S2\ \{R\},$$

то

$$\{P\}\ S1;\ S2\ \{R\}.$$

3. Условные операторы. Если $S1$ и $S2$ — операторы, для которых выполняются условия:

$$\{P\ \textbf{and}\ B\}\ S1\ \{Q\}\ \text{и}\ \{P\ \textbf{and not}\ B\}\ S2\ \{Q\},$$

то

$$\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \textbf{else}\ S2\ \{Q\}.$$

Если к тому же $S2$ — пустой оператор, то

$$\{P\}\ \textbf{if}\ B\ \textbf{then}\ S1\ \{Q\}.$$

4. Оператор цикла. Если S — оператор, для которого выполняется условие

$$\{P\ \textbf{and}\ B\}\ S\ \{P\},$$

то

$$\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \{P\ \textbf{and not}\ B\ \}.$$

Условие P здесь является инвариантом цикла.

Используя эти правила, можно значительно облегчить вывод пред- и постусловий. А теперь рассмотрим несколько взаимно эквивалентных конструкций в Паскале.

5. Эквивалентность последовательных операторов. Если $S1$ и $S2$ — операторы, в которых используются *различные* переменные, то эквивалентны последовательности операторов:

$$S1;\ S2\quad \text{и}\quad S2;\ S1$$

6. Эквивалентность операторов цикла. Если S — оператор, не изменяющий переменную i, то эквивалентны циклы:

$$i := a;\ \textbf{while}\ i <= b\ \textbf{do begin}\ S;\ \ i := i + 1\ \textbf{end}$$

и

$$\textbf{for}\ i := a\ \textbf{to}\ b\ \textbf{do}\ S$$

7. Эквивалентность условных операторов **if** *и* **case**. Эквивалентны операторы:

$$\textbf{case}\ i\ \textbf{of}\ a1 :\ S1;\ \ a2 :\ S2\ \textbf{else}\ S3\ \textbf{end}$$

и

$$\textbf{if}\ i = a1\ \textbf{then}\ S1\ \textbf{else if}\ i = a2\ \textbf{then}\ S2\ \textbf{else}\ S3$$

Эквивалентности 5 и 7 доказываются методом перечисления вариантов, а эквивалентность 6 — методом математической индукции. Их правильность читателю предлагается доказать самостоятельно.

Приведенный список взаимно эквивалентных простых алгоритмов далеко не самый полный, его необходимо постоянно пополнять по мере создания новых алгоритмов.

1.4. Об эффективности алгоритмов

Любой алгоритм для своего выполнения требует двух основных ресурсов — пространства и времени. Пространство измеряется объемом памяти для входных и выходных данных, переменных внутри алгоритма, а также памятью, необходимой для размещения самого алгоритма. Память для входных и выходных данных определяется условиями задачи, мы не можем ее произвольно изменять. Память для размещения алгоритма определяется длиной алгоритма, а также транслятором и никак не зависит от размера данных. Обычно она много меньше, чем память для данных, являющихся массивами.

В то же время объем памяти для размещения внутренних переменных в алгоритме может различаться для разных алгоритмов, решающих одну и ту же задачу. Алгоритм, которому требуется меньше памяти, более эффективен *по памяти*. Пусть объем памяти зависит от целочисленного входного параметра n. Если для работы алгоритма потребуется массив размерностью $20n$ элементов, то говорят, что такому алгоритму требуется линейная память, а если $n^2/2$ элементов — то квадратичная память и т.п. При разных

функциях объема памяти более эффективен тот алгоритм, у которого функция с ростом n растет медленнее. Например, для всех $n > 40$ функция $n^2/2$ больше, чем функция $20n$.

Время работы алгоритма определяется количеством выполняемых в нем действий и их сложностью, а также временем выполнения на *данном* компьютере отдельных элементарных действий. Однако при анализе алгоритмов важно лишь, *во сколько раз* один алгоритм будет выполняться быстрее другого при одних и тех же входных данных на одном и том же компьютере. Поэтому для каждого алгоритма необходимо определить функцию зависимости количества элементарных действий от входного параметра. Эту функцию называют *трудоемкостью* алгоритма. Так же как и функция объема памяти, функция трудоемкости может быть линейной, квадратичной и т.п. Исследование алгоритма и состоит в выводе функций объема памяти и трудоемкости.

Если для двух алгоритмов, решающих одну и ту же задачу, получены функции объема памяти и трудоемкости, то как определить, который из алгоритмов более эффективен? Типична ситуация, когда первый алгоритм более эффективен по памяти, зато второй — по трудоемкости. В этом случае, как правило, предпочитают алгоритм, более эффективный по трудоемкости. Дело в том, что у большинства алгоритмов функция объема памяти растет медленнее, чем функция трудоемкости. Кроме того, при выполнении алгоритма на компьютере время является более дефицитным ресурсом. Это связано прежде всего с тем, что с развитием техники объем памяти в компьютерах растет быстрее, чем быстродействие процессоров. Так, за последние 20 лет объем памяти "среднего" компьютера увеличился примерно в 1000 раз, а быстродействие — "всего" в 100 раз.

В качестве примера проанализируем алгоритм 1.2 (вычисление факториала). Объем требуемой памяти в нем — константа, так как в нем используются лишь несколько вспомогательных переменных и не используются массивы, размер которых зависит от n. Трудоемкость определяется количеством умножений в цикле, которое, в свою очередь, равно количеству выполнений цикла, т.е. n.

Мы предполагали, что для алгоритма функции объема памяти и трудоемкости зависят только от одного параметра n. Однако

для очень многих алгоритмов ситуация оказывается гораздо сложнее. Пусть, например, алгоритм обрабатывает числовой массив из n элементов. Часто бывает, что время работы алгоритма зависит не только от n, но и от конкретных значений элементов и от их взаимного расположения. Тогда надо постараться аналитически получить функции объема памяти и трудоемкости для *наихудшего* случая. Пессимист был бы доволен: он все предусмотрел. Однако оптимисту это вряд ли понравилось бы, так как в большинстве случаев оказалось бы, что алгоритм работает гораздо быстрее или тратит гораздо меньше памяти, чем предсказано выведенными функциями.

Что касается объема памяти, то потакать оптимисту не стоит: даже если в одном случае из ста программе не будет хватать выделенной памяти (быть может, всего нескольких байтов!) и программа из-за этого будет аварийно прекращать работу, то это в конце концов не понравится даже оптимисту. В то же время если изредка для получения результата придется подождать подольше, то чаще всего с этим можно смириться. Но для этого необходимо знать *среднее* время работы алгоритма, т.е. функцию трудоемкости от n *в среднем*. Получить аналитически такую функцию очень непросто. Поэтому для алгоритма, реализованного в виде конкретной программы, производят экспериментальные испытания, так называемое *моделирование*, с целью вычислить среднее время работы для различных n. При этом достаточно надежные численные результаты можно получить, лишь применив *теорию вероятностей* и *математическую статистику*. К сожалению, изучение этих интереснейших математических дисциплин выходит далеко за рамки нашей книги. Тем не менее, не претендуя на точные численные характеристики времени работы алгоритма, мы при необходимости все же будем проводить для него пробное моделирование, чтобы получить хотя бы приблизительное представление о времени его работы.

1.5. Разработка сложных алгоритмов и доказательство их правильности

Понятие сложности алгоритма неоднозначно. Опытный программист может считать алгоритм из 10 страниц текста простым, а для начинающего бывает, что алгоритм из 20 строк текста кажется непостижимо сложным. Кроме того, на сложность алгоритма влияет

его структура. Если, например, в алгоритме кроме строго последовательных вычислений по формулам ничего нет, то его можно считать простым независимо от длины. Конечно, и в этом случае его необходимо проверять и тестировать. Однако здесь все определяется самими формулами: если формулы правильны и записаны в алгоритме без ошибок, то и сам алгоритм правильный, поэтому проверка необходима лишь для того, чтобы убедиться, что при переписи формул в алгоритм в них не внесли новых ошибок.

Любая конструкция в алгоритме, изменяющая строго последовательное выполнение операторов, значительно усложняет структуру алгоритма и, соответственно, усложняет его проверку. Особенно "вредным" является оператор **goto**, для которого к тому же не существует удобных методов доказательства правильности. Часто бывает, что алгоритм с двумя – тремя **goto** оказывается гораздо сложнее алгоритма с многими десятками операторов **if – then – else**. Именно поэтому на языке Бэйсик, на котором не обойтись без **goto**, практически невозможно программировать *правильные* сложные алгоритмы.

Простой алгоритм можно придумать сразу весь, разработать более-менее сложный алгоритм таким "способом" не удается. Наиболее удобен для сложных алгоритмов метод *сверху – вниз*, называемый также *методом поэтапной разработки алгоритма*. Идея состоит в том, что внутри сложного алгоритма отдельные его части представляют как вспомогательные алгоритмы второго уровня. И поэтому вначале придумывают структуру из операторов (*верх*), связывающую алгоритмы второго уровня, которые еще не написаны. После этого переходят к детальной разработке и написанию отдельных алгоритмов второго уровня (*низ*). Конечно, уровней может быть и больше двух, большая программа может содержать, например, 8 – 10 уровней. Основное преимущество при этом состоит в том, что на любом промежуточном этапе разработки приходится иметь дело с относительно небольшим алгоритмом независимо от сложности всей программы. При этом про другие части алгоритма нужно временно "забыть", абстрагироваться от них.

При создании сложного алгоритма никакие полезные советы не помогут, если программист не накопил достаточного опыта разработки хотя бы простых алгоритмов. Этому, в свою очередь, можно научиться, лишь тщательно изучив большое количество "типо-

вых" алгоритмов, не слишком сложных, но очень широко встречающихся, как алгоритмы второго уровня, во многих больших программах. Квалифицированный программист всегда пытается разработать большую программу в виде структуры операторов, связывающей известные ему алгоритмы. Эти алгоритмы приходится, как правило, модифицировать, подгонять под конкретные условия. Таким образом, чтобы успешно разрабатывать сложные алгоритмы, программист должен:

1) знать и понимать большое количество типовых алгоритмов;

2) уметь модифицировать типовые алгоритмы;

3) уметь из типовых алгоритмов собирать, как из "кирпичиков", сложные алгоритмы.

В принципе не обязательно, чтобы программист умел придумывать *новые* типовые алгоритмы. Как *этому* научиться, не знает никто. Создание принципиально нового алгоритма сродни открытию, а открытия, как известно, делаются не каждый день.

Тестирование сложного алгоритма также является непростым делом. Его можно значительно облегчить, если придерживаться структуры алгоритма. Можно, например, начинать тестирование с неполного алгоритма, содержащего часть операторов первого уровня и один из алгоритмов второго уровня. Конечно, этот неполный алгоритм должен выдавать осмысленный результат, чтобы его можно было проверить. На следующем этапе к тестируемому алгоритму необходимо добавить еще один алгоритм второго уровня и т.д. Тогда при обнаружении ошибочной работы алгоритма ошибку следует искать в операторах, добавленных на *последнем* этапе.

Что касается доказательства правильности сложных алгоритмов, то основной метод для этого — *абстракция*. Предварительно для алгоритма первого уровня и для каждого алгоритма второго уровня задают предусловия и постусловия. После этого доказательство проводят отдельно для каждого алгоритма второго уровня, абстрагируясь от остальных операторов алгоритма. Наконец, при доказательстве правильности алгоритма первого уровня абстрагируются от структуры алгоритмов второго уровня и используют только предусловия и постусловия для них. Примеры таких многоэтапных доказательств будут приведены в следующих разделах.

Многие алгоритмы второго (и более) уровня являются модификациями типовых алгоритмов с известным доказательством правиль-

ности. Поэтому в основном правильность модифицированных алгоритмов доказывается методом эквивалентов.

При тестировании большой программы полезен еще один прием — *машинная проверка условий*. Его суть состоит в том, что для каждого алгоритма второго уровня добавляют вспомогательные операторы проверки предусловия на входе и постусловия на выходе, которые должны сигнализировать в случае нарушения этих условий. При этом записывают *логическое отрицание* проверяемого условия. Проверки помогут выявить ошибки в доказательстве, иногда даже позволят вообще обойтись без доказательства. В качестве примера рассмотрим алгоритм 1.1 с машинной проверкой условий.

Алгоритм 1.4. Машинная проверка условий. Предусловие для числовых переменных x, y до выполнения алгоритма: $x > 0$, $y > 0$. Постусловие после выполнения алгоритма: $x \geq y > 0$.

```
if (x <= 0) or (y <= 0) then
  writeln('Ошибка предусловия : x =', x,' y =', y);
if x < y then begin z := y; y := x; x := z end
if (x < y) or (y <= 0) then
  writeln('Ошибка постусловия : x =', x,' y =', y);
```

Вопросы и задания для самостоятельной работы

1. Что подразумевается под универсальностью алгоритма?
2. Что является задачей тестирования и почему?
3. В чем состоит доказательство правильности алгоритма?
4. Какие существуют методы доказательства правильности алгоритмов и в каких случаях целесообразно применять тот или иной метод?
5. Доказать эквивалентности 5, 6 и 7.
6. Для цикла **repeat** построить эквивалентную конструкцию из цикла **while**. Доказать их эквивалентность.
7. Что значит: "первый алгоритм эффективнее второго"?
8. В чем заключается разработка алгоритма "сверху-вниз"?
9. Как доказывать правильность сложных алгоритмов?
10. Для алгоритма 1.3 (вычисление факториала) записать машинную проверку условий.
11. Записать алгоритм вычисления суммы квадратов чисел от 1 до n и доказать его правильность методом математической индукции и с помощью инварианта.

Глава 2. ОДНОПРОХОДНЫЕ АЛГОРИТМЫ

К однопроходным относят обычно алгоритмы, в которых основной конструкцией является цикл с параметром, изменяющимся по закону арифметической прогрессии. Примерами являются алгоритмы вычисления факториала из предыдущего раздела. Часто однопроходный алгоритм работает с массивом. Доказательства правильности таких алгоритмов обычно очень просты и строятся на основе метода математической индукции или инварианта при очевидной завершимости.

Исследование эффективности для них также несложно. Объем требуемой памяти определяется используемыми массивами. Трудоемкость определяется количеством выполнений операторов в цикле, т.е. является линейной.

Приведенные ниже алгоритмы, несмотря на всю их простоту, являются полезными "кирпичиками" для построения сложных программ.

Алгоритм 2.1. Вычисление суммы s всех n элементов массива A.

```
s := 0;
for i := 1 to n do s := s + A[i];
```

Алгоритм 2.2. Заполнение массива A из n элементов целочисленными случайными значениями из диапазона от 0 до 4999.

```
randomize;
for i := 1 to n do A[i] := random(5000);
```

Алгоритм 2.3. Вычисление максимального значения max среди всех n элементов массива A.

```
max := A[1];
for i := 2 to n do  if max < A[i] then max := A[i];
```

Алгоритм 2.4. Вычисление номера k элемента массива A с максимальным значением среди всех n элементов.

```
k := 1;
for i := 2 to n do if A[k] < A[i] then k := i;
```

Доказательство правильности этих алгоритмов предлагается выполнить в качестве упражнения.

2.1. Рекуррентные последовательности и однопроходные алгоритмы

Определение 2.1. Числовая последовательность $\{x_k\}$ называется *рекуррентной ранга* p, если

$$\begin{cases} x_k = a_k, & k = 0, 1, \ldots, p-1, \\ x_k = f(k, x_{k-1}, x_{k-2}, \ldots, x_{k-p}), & k = p, p+1, \ldots \end{cases}$$

где $a_0, a_1, \ldots, a_{p-1}$ — константы, а f — функция.

Определение рекуррентной последовательности само является алгоритмом, поэтому для вычисления ее элементов это определение надо просто "переписать" на Паскале.

Алгоритм 2.5. Массив A с нумерацией от 0 до $p-1$ содержит константы $a_0, \ldots, a_{p-1}$, а в массив X с нумерацией от 0 до n будут записываться элементы последовательности. Функция f представляет собой формулу, в которой используются элементы массива X с номерами от $k-p$ до $k-1$.

```
for k := 0 to p - 1 do X[k] := A[k];
for k := p to n do X[k] := f(k, X);
```

▷ Доказательство правильности этого алгоритма излишне, достаточно лишь убедиться в том, что алгоритм работает в строгом соответствии с определением рекуррентной последовательности. Первая строчка в алгоритме соответствует первой строчке определения 2.1 и вычисляет элементы последовательности с номерами $0, 1, \ldots, p-1$. Вторая строчка алгоритма вычисляет последующие элементы с номерами $p, p+1, \ldots, n$. ◁

Заметим, что алгоритм 2.5 на самом деле является "полуфабрикатом", во второй строчке вместо $f(k, X)$ надо записать конкретную формулу, зависящую от величин $k, X[k-1], X[k-2], \ldots, X[k-p]$.

Таким образом, вычисления по рекуррентной последовательности будут правильны, если последовательность задана корректно. Поэтому на самом деле необходимо доказывать правильность определения конкретной рекуррентной последовательности. Иногда удается вывести формулу, непосредственно выражающую n-й элемент последовательности. Доказательство правильности такой формулы можно произвести методом математической индукции.

Например, сумма квадратов чисел от 1 до n задается следующей рекуррентной последовательностью:

$$\begin{cases} s_0 = 0, \\ s_k = s_{k-1} + k^2 \quad k = 1, 2, \ldots, n; \end{cases} \tag{2.1}$$

Докажем, что для нее справедлива формула

$$s_n = \frac{2n^3 + 3n^2 + n}{6}; \tag{2.2}$$

▷ **Доказательство.**
Базис. При $n = 0$ сумма $s_n = 0$, формула (2.2) верна.
Предположение. При $n \geq 0$ формула (2.2) верна.
Вывод. При $n + 1$ по формуле (2.2) получаем:

$$s_{n+1} = \frac{2(n+1)^3+3(n+1)^2+(n+1)}{6} = \frac{2n^3+3n^2+n}{6} + \frac{6n^2+12n+6}{6} = s_n + (n+1)^2,$$

что совпадает с определением рекуррентной последовательности (2.1). ◁

Еще один пример рекуррентной последовательности — числа Фибоначчи:

$$\begin{cases} f_0 = 0, \\ f_1 = 1, \\ f_k = f_{k-1} + f_{k-2}, \quad k = 2, 3, \ldots \end{cases} \tag{2.3}$$

Числа Фибоначчи растут очень быстро. Вот несколько начальных чисел: *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233* ...

Алгоритм 2.6. Вычисление n–го числа Фибоначчи f для $n > 0$.

```
g := 0; f := 1;
for i := 2 to n do
  begin h := g + f; g := f; f := h end
```

Для чисел Фибоначчи справедлива следующая формула:

$$f_k = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k\right). \qquad (2.4)$$

▷ **Доказательство.**
Базис. $f_0 = 0, f_1 = 1$ по рекуррентной формуле (2.3) и по формуле (2.4).
Предположение. При $k \geq 1$ формула (2.4) верна.
Вывод. При $k + 1 \geq 2$ по формуле (2.4) получим:

$$f_{k+1} = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{k+1}\right) =$$

$$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k-1}\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^{k-1}\left(\frac{1-\sqrt{5}}{2}\right)^2\right) =$$

$$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{k-1}\left(\frac{1+\sqrt{5}}{2}+1\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{k-1}\left(\frac{1-\sqrt{5}}{2}+1\right)\right) =$$

$$\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^k + \left(\frac{1+\sqrt{5}}{2}\right)^{k-1} - \left(\frac{1-\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^{k-1}\right) = f_k + f_{k-1},$$

что совпадает с рекуррентной формулой (2.3). ◁

Заметим, что для вычисления чисел Фибоначчи формула (2.4) не всегда удобна, ее целесообразно применять лишь для больших k. При этом из-за квадратных корней вычисления придется выполнять над вещественными (приближенными!) числами (типа **real** в Паскале).

Мы здесь не обсуждаем вопрос о том, как *придумывать* такие формулы — эта весьма непростая проблема не всегда имеет решение. Поэтому единственным *универсальным* методом вычисления элементов рекуррентных последовательностей, задаваемых в соответствии с определением 2.1, является циклический однопроходный алгоритм.

2.2. Алгоритмы с упорядоченными массивами и множествами

Целочисленный массив с расположенными по неубыванию или по невозрастанию значениями элементов называется упорядоченным. Использование упорядоченности позволяет создавать эффективные алгоритмы для решения многих интересных задач.

Задача *слияния* двух входных упорядоченных массивов A и B состоит в формировании упорядоченного выходного массива C, содержащего *все* элементы из входных массивов.

Рассмотрим алгоритм слияния для упорядоченных по неубыванию массивов. Вначале элемент $A[1]$ сравнивается с элементом $B[1]$ и наименьший из них переписывается в массив C. Если наименьшим был $A[1]$, то на следующем шаге сравниваются $A[2]$ и $B[1]$, а если наименьшим был $B[1]$, то будут сравниваться $A[1]$, и $B[2]$ и т.д. Если на очередном шаге окажется, что из одного входного массива все элементы переписаны в C, то переписывается элемент из другого массива. На рис. 2.1 показан промежуточный этап работы алгоритма, когда сравниваются элементы $A[i1]$ и $B[i2]$ и меньший из них пересылается в $C[i]$. При равенстве можно пересылать любой элемент.

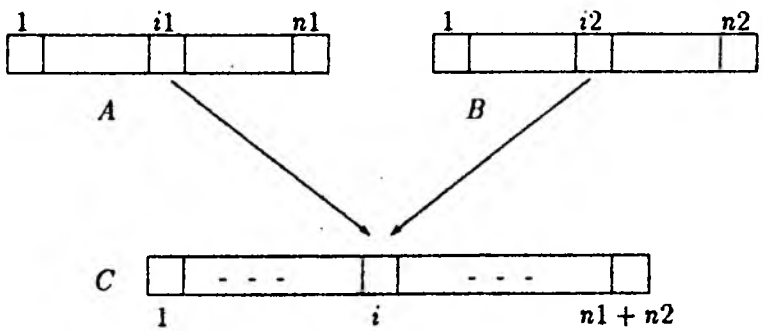


Рис. 2.1

Пример работы алгоритма слияния. В массиве A содержится 3 элемента: $\{5, 13, 14\}$, а в массиве B — 4 элемента: $\{7, 9, 10, 12\}$. В табл. 2.1 по шагам показано изменение переменных $i, i1, i2$ и действия с массивами.

Таблица 2.1

i	$i1$	$i2$	Сравниваются	$C[i]$
1	1	1	$A[1] = 5$ и $B[1] = 7$	5
2	2	1	$A[2] = 13$ и $B[1] = 7$	7
3	2	2	$A[2] = 13$ и $B[2] = 9$	9
4	2	3	$A[2] = 13$ и $B[3] = 10$	10
5	2	4	$A[2] = 13$ и $B[4] = 12$	12
6	2	5	B весь переписан	13
7	3	5	B весь переписан	14

Алгоритм 2.7. Слияние. Даны два входных упорядоченных по неубыванию массива, необходимо создать третий упорядоченный массив, содержащий все элементы из двух входных. Входные массивы A и B содержат соответственно $n1$ и $n2$ элементов, результирующий массив C будет содержать $n1 + n2$ элементов.

```
i1 := 1; i2 := 1;                                   {1}
for i := 1 to n1 + n2 do                            {2}
  if i1 > n1 then                                   {3}
    begin C[i] := B[i2]; i2 := i2 + 1 end           {4}
  else if i2 > n2 then                              {5}
    begin C[i] := A[i1]; i1 := i1 + 1 end           {6}
  else if A[i1] ≤ B[i2] then                        {7}
    begin C[i] := A[i1]; i1 := i1 + 1 end           {8}
  else                                              {9}
    begin C[i] := B[i2]; i2 := i2 + 1 end           {10}
```

▷ **Доказательство правильности.** Согласно *методу эквивалентов* циклу **for** в алгоритме эквивалентен следующий цикл **while**, который мы и будем далее рассматривать:

```
i := 1; while i <= n1 + n2 do begin  ...  i := i + 1 end
```

Инвариантом *этого цикла* будет следующее утверждение:

"*Элементам массива* C: $C[1] \le \ldots \le C[i-1]$ *присвоено значение* $i1 - 1$ *первых элементов из массива* A *и* $i2 - 1$ *первых элементов из массива* B, *при* $i1 \le n1$: $C[i-1] \le A[i1]$, *при* $i2 \le n2$: $C[i-1] \le B[i2]$".

Завершимость алгоритма очевидна, цикл выполнится ровно $n1 + n2$ раза, после его выполнения $i = n1 + n2 + 1$. Если инвариант справедлив, то после выполнения цикла в $n1 + n2$ элементах массива C окажутся упорядоченные значения всех элементов из A и B.

Доказательство инварианта. Перед выполнением цикла в инвариант входит нулевое количество элементов массива C, поэтому истинность инварианта очевидна. При каждом выполнении цикла возможен один из четырех случаев:

1) $i1 = n1 + 1$. Так как инвариант справедлив до выполнения тела цикла, то это значит, что все элементы из массива A переписаны в массив C, откуда следует, что элемент $B[i2]$ не меньше любого из элементов массива A. Так как массив B упорядочен, то $B[i2]$ не меньше всех элементов B, ранее переписанных в массив C, и не больше, чем элементы $B[i2 + 1], \ldots, B[n2]$. В алгоритме исполнятся присваивания в строчке {4}, в результате чего элементу $C[i]$ будет присвоено значение $B[i2]$, для которого $C[i - 1] \le B[i2] \le B[i2 + 1]$. Значения $i2$ и i после этого будут увеличены на 1, и инвариант опять будет справедлив, но уже для новых значений $i2$ и i.

2) $i2 = n2 + 1$. Рассуждения полностью аналогичны приведенным в п. 1) при замене $i1$ на $i2$, A на B и обратно. Надо лишь проверить присваивания в строчке {6}.

3) $i1 \le n1$, $i2 \le n2$, $A[i1] \le B[i2]$. Тогда $A[i1]$ — наименьший элемент из массивов A, B, который еще не переписан в C. Будут выполняться присваивания в строчке {8}, и именно $A[i1]$ перепишется в C. После увеличения $i1$ и i на 1 инвариант снова станет справедлив.

4) $i1 \le n1$, $i2 \le n2$, $A[i1] > B[i2]$. После симметричной замены в обозначениях рассуждения аналогичны п. 3).

Заметим, что невозможно, чтобы $i1 > n1 + 1$ либо $i2 > n2 + 1$ (доказать самостоятельно). ◁

Следующий алгоритм из упорядоченного по неубыванию массива формирует упорядоченный по возрастанию массив.

Алгоритм 2.8. Дан входной упорядоченный массив A длиной n. В массив C по возрастанию переписываются неповторяющиеся элементы, m — их количество.

```
m := 1; C[1] := A[1];
for i := 2 to n do
  if C[m] < A[i] then
    begin m := m + 1; C[m] := A[i] end
```

▷ Доказательство правильности этого алгоритма несложно после перехода к эквивалентному циклу **while**. Инвариантом цикла будет утверждение:

"Все m *различных* по значению элементов $A[1], \ldots, A[i-1]$ переписаны в первые m элементов массива C по возрастанию."

Тогда из постусловия $i = n+1$ немедленно следует правильность алгоритма. ◁

Особый интерес представляют упорядоченные целочисленные массивы с неповторяющимися элементами (их можно получать алгоритмом 2.8). Такие массивы можно интерпретировать как *множества*.

Пусть задано N некоторых объектов (множество объектов). Все объекты перенумерованы числами от 1 до N, т.е. каждый объект полностью определяется своим номером, и любым двум объектам приписаны *различные* номера. Все это множество объектов назовем *исходным*. Некоторые из объектов исходного множества можно каким-либо образом выделить. Тогда образуется множество выделенных объектов, его мы также будем называть множеством. Множество выделенных объектов можно задать, перечислив номера этих объектов, т.е. множество задается целочисленным массивом с неповторяющимися элементами. Для эффективности алгоритмов такой массив должен быть упорядоченным.

Пример 2.1. Задано исходное множество учащихся и некоторые сведения о них:

N	Учащийся	Знает Паскаль	Умеет плавать
1	Иванов Иван	да	да
2	Петров Петр	нет	да
3	Макаров Сергей	да	нет
4	Сергеева Ира	да	да
5	Мишина Мария	нет	нет
6	Устинов Илья	да	нет

Зададим множество A как множество учащихся, знающих Паскаль, и множество B как множество учащихся, умеющих плавать.

Тогда $A = \{1,3,4,6\}$, $B = \{1,2,4\}$.

Над множествами можно производить некоторые *операции*.

Определение 2.2 Объединением множеств A и B называется такое множество C, $C = A \cup B$, содержащее все те элементы, которые имеются либо в A, либо в B, либо в A и в B одновременно, и только эти элементы.

Определение 2.3 Пересечением множеств A и B называется такое множество C, $C = A \cap B$, содержащее все те элементы, которые имеются одновременно в A и в B, и только эти элементы.

Определение 2.4 Разностью множеств A и B называется такое множество C, $C = A \setminus B$, содержащее все те элементы, которые имеются в A, но отсутствуют в B, и только эти элементы.

Определение 2.5 Симметрической разностью множеств A и B называется такое множество C, $C = A \oplus B$, содержащее все те элементы, которые имеются в A, но отсутствуют в B, и все элементы, которые имеются в B, но отсутствуют в A, и только эти элементы.

Пример 2.2. Операции над множествами A и B из примера 2.1:

$C = A \cup B = \{1,2,3,4,6\}$ — множество тех, кто знает Паскаль или хотя бы умеет плавать;

$C = A \cap B = \{1,4\}$ — множество тех, кто не только знает Паскаль, но и умеет плавать;

$C = A \setminus B = \{3,6\}$ — множество тех, кто знает Паскаль, но не умеет плавать;

$C = A \oplus B = \{2,3,6\}$ — множество тех, кто знает Паскаль, но не умеет плавать либо умеет плавать, но не знает Паскаль.

Взяв за основу алгоритм 2.7 (слияние), можно легко реализовать любую из рассмотренных операций. В приведенных ниже алгоритмах массивы A и B задают множества: A из $n1$, B из $n2$ элементов. Результат — в массиве C из m элементов.

Алгоритм 2.9. Объединение множеств. $C = A \cup B$.

```
i1 := 1; i2 := 1; m := 0;
while (i1 ≤ n1) or (i2 ≤ n2) do
begin m := m + 1;
 if i1 > n1 then
   begin C[m] := B[i2]; i2 := i2 + 1 end
 else if i2 > n2 then
   begin C[m] := A[i1]; i1 := i1 + 1 end
 else if A[i1] < B[i2] then
   begin C[m] := A[i1]; i1 := i1 + 1 end
 else if A[i1] > B[i2] then
   begin C[m] := B[i2]; i2 := i2 + 1 end
 else
   begin C[m] := A[i1]; i1 := i1 + 1; i2 := i2 + 1 end
end
```

Алгоритм 2.10. Разность множеств. $C = A \setminus B$.

```
i1 := 1; i2 := 1; m := 0;
while (i1 ≤ n1) do
 if i2 > n2 then
   begin m := m + 1; C[m] := A[i1]; i1 := i1 + 1 end
 else if A[i1] < B[i2] then
   begin m := m + 1; C[m] := A[i1]; i1 := i1 + 1 end
 else if A[i1] > B[i2] then i2 := i2 + 1
 else begin i1 := i1 + 1; i2 := i2 + 1 end
```

Доказательство правильности этих алгоритмов предлагается читателю выполнить самостоятельно.

Рассмотрим еще одну задачу, в которой используется понятие упорядоченности. Задан числовой массив A, содержащий n элементов. Требуется определить начало и конец упорядоченного отрезка в нем, имеющего максимальную длину. Заметим, что такие отрезки не могут перекрываться. Если некоторый отрезок заканчивается на j-м элементе массива, то следующий за ним отрезок (имеющий хотя бы единичную длину) начинается с $(j+1)$-го элемента. Заметим также, что последний отрезок заканчивается концом массива.

Алгоритм 2.11. Дан числовой массив A и количество элементов в нем n. Результат работы: начало a и конец b упорядоченного отрезка максимальной длины. Переменная c обозначает начало, j — конец очередного просматриваемого в цикле отрезка.

```
a := 1; b := 0; c := 1;
for j := 1 to n do
 if (j = n)or(A[j] > A[j + 1]) then
 begin              {обнаружен конец очередного отрезка}
  if (j - c) > (b - a) then
    begin a := c; b := j end;
  c := j + 1
 end
```

▷ Доказательство правильности алгоритма после перехода к эквивалентному циклу **while** можно провести с помощью следующего инвариантного утверждения: "*a — начало, b — конец упорядоченного отрезка максимальной длины среди элементов массива* $A[1], \ldots, A[c-1]$". При этом следует отдельно рассмотреть два случая: 1) j — конец отрезка максимальной длины; 2) j — *не* конец отрезка максимальной длины. ◁

2.3. Алгоритмы с файлами

Обычно в программе предусматривают ввод исходных данных с клавиатуры и вывод вычисленных результатов на экран. Если входные данные имеют большой объем, то удобнее эти данные заранее подготовить и записать в виде файла на диске (или на дискете). После этого можно многократно запускать программу на выполнение, используя этот файл как набор входных данных. Предварительный ввод данных в файл можно произвести с помощью любого подходящего текстового редактора. Полученный при этом файл называется текстовым, каждый байт в нем задает определенный символ.

В Паскале предусмотрены средства работы с файлами, однако они несколько различаются для разных компьютеров и разных трансляторов. Далее мы рассмотрим работу с файлами в Турбо-Паскале для компьютеров типа IBM PC и им подобных.

В программе текстовый файл определяется как особая переменная, имеющая специальный файловый тип *text*. Описание для этого, например, такое:

var F : *text*;

Вначале необходимо связать переменную F с файлом, находящимся на диске (пусть имя этого файла `dan1.dat`). Связь устанавливается процедурой *assign* (назначить):

assign(F, '`dan1.dat`');

Затем необходимо открыть файл для чтения. Для этого предназначена процедура *reset* (установить):

reset(F);

После этого можно читать данные в память процедурой *read*:

read(F, ...).

В конце работы файл нужно закрыть процедурой *close*:

close(F);

Рассмотрим простой пример чтения данных из файла.

Алгоритм 2.12. В файле '`dan1.dat`' записано количество числовых значений, а затем сами числа. Программа вводит эти числа, вычисляет их сумму и выводит результат на экран.

```
var F : text;
    n, i : integer;
    s, a : real;
begin
  assign(F, 'dan1.dat');
  reset(F);
  read(F, n); s := 0;
  for i := 1 to n do
    begin read(F, a);
      s := s + a;
    end;
  close(F);
  writeln('Сумма ', n,' чисел: ', s : 10 : 4);
end.
```

Если количество входных данных в файле заранее неизвестно, то перед вводом очередной порции данных можно проверить, не достигнут ли конец файла. Это можно сделать функцией *eof*(F), которая выдает значение **истина**, если файл прочитан весь, и **ложь** в противном случае. Для текстового файла конец определяется либо

физически, когда очередных данных для чтения в файле больше нет, либо по специальному символу с кодом 27. Во втором случае остаток файла после этого символа читаться не будет.

Мы рассмотрели действия с *входным* файлом. *Выходной* файл следует открывать процедурой *rewrite*(F). Если файл с указанным (в процедуре *assign*) именем уже существует, то все имеющиеся в нем данные уничтожаются. Если же такого файла нет, то он создается. В обоих случаях данные будут записываться с самого начала файла. Непосредственно запись данных из памяти в файл производится процедурой *write*(F, ...).

В качестве примера рассмотрим алгоритм чтения символьных данных из файла в массив и одновременного копирования их в другой файл.

Алгоритм 2.13. В файле `'dan2.dat'` записаны произвольные символы (текст). Программа вводит их в массив t, отображает на экране, записывает в файл `'dan3.dat'` и, кроме того, вычисляет количество символов в файле.

```
var F, D : text;
    n : integer;
    c : char;
    t : array[1..10000]of char;
begin
  assign(F, 'dan2.dat');
  assign(D, 'dan3.dat');
  reset(F); rewrite(D);
  n := 0;
  while not eof(F) do
    begin read(F,c); write(D,c);
      write(c);
      n := n + 1; t[n] := c
    end;
  close(F); close(D);
  writeln; writeln('Длина текста: ', n);
end.
```

Информация, записываемая в файл, не всегда бывает текстовой.

Например, целые числа во внутреннем представлении занимают по два байта. Бессмысленно каждый из этих байтов интерпретировать как некоторый символ, хотя символ записывается как раз в одном байте. В этом случае файл должен быть нетекстовым. В программе такой файл описывается как переменная типа *file* (нетипизированный файл). Работа с ним несколько отличается от работы с текстовым файлом.

В процедуре *reset* или *rewrite* при открытии файла указывается второй параметр — длина блока (логической единицы) считываемой или записываемой информации в байтах.

Для чтения и записи используются процедуры *blockread* и *blockwrite* соответственно. В них указываются по три параметра: файл, переменная для чтения или записи, длина читаемой или записываемой информации в байтах.

В качестве примера рассмотрим алгоритм копирования произвольной двоичной информации из одного файла в другой.

Алгоритм 2.14. В файле `'dan2.dat'` записана произвольная двоичная информация. Программа всю ее переписывает в файл `'dan3.dat'`.

```
var F, D : file;
    c : char;
begin
  assign(F, 'dan2.dat');
  assign(D, 'dan3.dat');
  reset(F,1); rewrite(D,1);
  while not eof(F) do
    begin blockread(F,c,1);
      blockwrite(D,c,1)
    end;
  close(F); close(D);
end.
```

Задания для самостоятельной работы

1. Доказать правильность алгоритмов 2.1 – 2.4.

2. Записать рекуррентную формулу суммы s целых чисел от 1 до n, доказать для нее правильность формулы $s = n(n+1)/2$.

3. Доказать правильность алгоритма 2.6 (вычисление чисел Фибоначчи).

4. Написать алгоритм, который в числовом массиве сдвигает ненулевые элементы в начало, а нулевые — в конец массива. Доказать его правильность.

5. Взяв за основу алгоритм 2.8, написать алгоритм удаления повторяющихся элементов непосредственно в упорядоченном массиве. Доказать его правильность.

6. Доказать правильность алгоритмов 2.9 – 2.10 (операции объединения и разности множеств).

7. Написать алгоритмы операций пересечения и симметрической разности множеств и доказать их правильность.

8. Взяв за основу алгоритм 2.11, написать алгоритм поиска упорядоченного как по возрастанию, так и по убыванию отрезка максимальной длины в массиве. Доказать правильность алгоритма.

9. Дополнить алгоритм 2.7 (слияние) вводом данных из двух файлов и выводом результата в третий файл.

10. Взяв за основу алгоритм 2.7 (слияние), написать алгоритм слияния упорядоченных наборов чисел, записанных в двух файлах, с записью результата в третий файл *без использования массивов*.

11. Написать алгоритм, который из файла читает *все* записанные в нем числа, подсчитывает их количество, сумму чисел и сумму квадратов чисел.

Глава 3. АЛГОРИТМЫ СОРТИРОВКИ И ПОИСКА

Алгоритм, который переставляет элементы в массиве таким образом, что массив становится упорядоченным, называется алгоритмом сортировки. Необходимость упорядочения массивов возникает при решении самых разных задач, и очень часто компьютер более половины полезного времени своей работы тратит именно на сортировку. Известно очень много самых разных алгоритмов сортировки, как простых, так и сложных, обладающих различной эффективностью.

3.1. Простые алгоритмы сортировки

Самый простой алгоритм сортировки работает в соответствии с определением упорядоченного массива: проверяет все рядом стоящие элементы массива $A[i]$ и $A[i+1]$, и как только обнаружит, что условие упорядоченности не выполняется, меняет их местами и продолжает проверку с самого начала массива.

Алгоритм 3.1. Простейшая сортировка массива A из n элементов.

```
i := 1;
while i < n do
  if A[i] <= A[i + 1] then i := i + 1;
  else begin t := A[i]; A[i] := A[i + 1]; A[i + 1] := t; i := 1 end
```

▷ **Доказательство правильности.**

Завершимость алгоритма. Введем определение инверсии. Инверсией называется условие $A[k] > A[i]$ при $k < i$. Подсчитаем максимально возможное количество инверсий для всех пар k, i. Для фиксированного i максимально возможное количество инверсий равно $i-1$, поэтому для всех $i \le n$ общее число инверсий равно сумме чисел $1, 2, \ldots, (n-1)$, т.е. $n(n-1)/2$

В алгоритме при обнаружении инверсии элементы переставляются местами, т.е. количество инверсий уменьшается на одну, после чего просмотр возобновляется с самого начала массива. Таким образом, если количество инверсий в массиве равно I, то массив будет начинать просматриваться от начала $I+1$ раз, в наихудшем случае $1+n(n-1)/2$ раз и в любом случае заканчивает работу, после чего в массиве не остается ни одной инверсии. После окончания цикла переменная i становится равной n.

Инвариант: "$A[1] \le A[2] \le \ldots \le A[i]$" (проверить самостоятельно). Так как после окончания алгоритма $i = n$, то массив окажется полностью упорядоченным. ◁

Трудоемкость алгоритма 3.1 в наилучшем случае, когда массив *уже упорядочен* по возрастанию, составляет $n-1$ шагов, а трудоемкость в наихудшем случае будет, если массив упорядочен по убыванию, когда число инверсий $n(n-1)/2$. Оценим число повторений цикла в наихудшем. В соответствии с инвариантом рассмотрим ситуацию, когда $A[1] \le A[2] \le \ldots \le A[i]$, но одновременно $A[i] > A[i+1]$. Чтобы выполнялось условие $A[1] \le \ldots \le A[i+1]$, алгоритму потребуется $i-1$ шагов цикла, чтобы дойти до сравнения $(i-1)$-го и i-го элементов массива, затем еще $i-2$ шагов цикла, чтобы дойти до сравнения $(i-2)$-го и $(i-1)$-го элементов и т.д., всего около $i^2/2$ шагов. Таким образом, чтобы массив стал полностью упорядоченным, алгоритму потребуется около

$$1^2/2 + 2^2/2 + \ldots + (n-1)^2/2 \approx n^3/6$$

шагов (здесь использована формула (2.2)). Это, конечно, слишком много, поэтому модифицируем алгоритм 3.1.

Алгоритм 3.2. Сортировка обменом массива A из n элементов.

```
i := 1;
while i < n do
 if A[i] <= A[i + 1] then i := i + 1;
 else begin t := A[i]; A[i] := A[i + 1]; A[i + 1] := t;
  if i > 1 then i := i - 1
 end
```

Для этого алгоритма инвариант в точности такой же, как и для алгоритма 3.1. Однако чтобы для наихудшего случая ситуацию

$A[1] \le A[2] \le \ldots \le A[i]$, $A[i] > A[i+1]$ превратить в ситуацию $A[1] \le \ldots \le A[i+1]$, алгоритму 3.2 потребуется всего около $2i$ шагов, т.е. суммарная трудоемкость упорядочения массива из n элементов будет около n^2 шагов.

Это намного лучше, чем $n^3/6$, однако все равно алгоритм 3.2 делает лишнюю работу. Действительно, перестановка элемента $A[i+1]$ на первое место потребует i шагов, но чтобы после этого перейти от сравнения элементов $A[1]$ и $A[2]$ к сравнению элементов $A[i+1]$ и $A[i+2]$, алгоритму потребуется выполнить i заведомо бесполезных шагов. Поэтому представляет интерес улучшенная версия сортировки обменом, имеющая трудоемкость $n(n-1)/2$ шагов цикла в наихудшем случае.

Алгоритм 3.3. Улучшенная сортировка обменом массива A из n элементов.

```
for i := 1 to n - 1 do
begin j := i;
  while (j > 0) and (A[j] > A[j + 1]) do
    begin t := A[j]; A[j] := A[j + 1]; A[j + 1] := t; j := j - 1 end
end
```

Рассмотрим еще один алгоритм сортировки — сортировку выбором. Этот алгоритм основан на идее алгоритма 2.4, вычисляющего номер элемента массива с максимальным значением. Затем максимальный элемент переставляется с последним в массиве. Такие действия повторяются в цикле сначала для n элементов массива, затем для $n-1$ первых элементов и т.д. до двух элементов.

Алгоритм 3.4. Сортировка выбором массива A из n элементов.

```
for i := n downto 2 do
begin k := 1;
  for j := 2 to i do if A[j] > A[k] then k := j;
  t := A[k]; A[k] := A[i]; A[i] := t
end
```

Для этого алгоритма количество сравнений в наихудшем и в наи-

лучшем случаях одно и то же — $n(n-1)/2$.

Известно много других простых алгоритмов сортировки. В качестве самостоятельного задания предлагаем запрограммировать алгоритм "пузырьковой сортировки". Свое название алгоритм получил из-за аналогии движения переставляемых в конец массива элементов, имеющих наибольшее значение, с всплыванием пузырьков пара в кипящей воде.

Алгоритм "пузырьковой сортировки" состоит из двойного цикла просмотра и сравнения пар соседних элементов. Если некоторая пара элементов расположена не по возрастанию, то эти элементы обмениваются значениями.

3.2. Сортировка слиянием

Добиться гораздо меньшей трудоемкости сортировки можно, используя идею слияния двух упорядоченных массивов (алгоритм 2.7). Для работы алгоритма сортировки кроме входного массива из n элементов потребуется еще один вспомогательный массив такой же длины. Вначале предполагается, что входной массив содержит n упорядоченных кусков, каждый из которых содержит по одному элементу. Слияние производится поэтапно.

На первом этапе слияние производится отдельно для первого и второго куска, затем для третьего и четвертого, и т.д. При слиянии образуются упорядоченные куски по два элемента, которые переписываются последовательно во вспомогательный массив. По окончании этапа во вспомогательном массиве сформируются упорядоченные куски по два элемента.

На втором этапе входной и вспомогательный массивы меняются местами и слияние продолжается. После второго этапа упорядоченные куски будут содержать по 4 элемента, после третьего — по 8 элементов и т.д. Наконец, наступит момент, когда весь массив будет представлять собой один упорядоченный кусок, и тогда алгоритм закончит работу. При этом упорядоченный массив может оказаться как на месте входного, так и на месте вспомогательного массива.

Для использования алгоритма слияния 2.7 в сортировке его необходимо модифицировать следующим образом:

1) вместо входных массивов сливаться должны соседние куски в массиве A, начало первого куска $b1$, конец первого куска $e1$, начало второго куска $e1+1$, конец второго куска $e2$;

2) выходным массивом должен быть кусок массива B, его начало $b1$, конец — $e2$;

3) алгоритм слияния должен быть оформлен в виде процедуры, вызов которой следующий: $S(A, B, b1, e1, e2)$.

Алгоритм 3.5. Сортировка слиянием. Входной массив A из n элементов, B — вспомогательный массив, p — признак размещения результата, если $p = 1$, результат в массиве A, если $p = 2$ — результат в B.

```
l := 1; p = 1;                                                    {1}
while l < n do                                                    {2}
begin b1 = 1;                                                     {3}
  while b1 <= n do                                                {4}
  begin e1 := b1 + l - 1;                                         {5}
    if e1 > n then e1 := n;                                       {6}
    b2 := e1 + 1; e2 := e1 + l;                                   {7}
    if e2 > n then e2 := n;                                       {8}
    if p = 1 then S(A,B,b1,e1,e2) else S(B,A,b1,e1,e2)            {9}
    b1:=e2+1                                                      {10}
  end;                                                            {11}
  p := 3 - p; l := l + l                                          {12}
end                                                               {13}
```

Переменная l в алгоритме содержит длину упорядоченных кусков. Цикл с заголовком в строчке 2 управляет переписыванием со слиянием из массива в массив, переменная p определяет направление переписывания: при $p = 1$ из A в B, при $p = 2$ — наоборот. В конце цикла (строчка 12) изменяется значение p и удваивается значение l. Условие прекращения цикла: $l \geq n$, когда массив полностью упорядочен.

Цикл с заголовком в строчке 4 перебирает пары упорядоченных кусков для слияния. Переменная $b1$ (начало первого куска) управляет работой этого цикла. В конце цикла (строчка 10) ее значение становится на 1 больше, чем конец второго куска (переменная $e2$). Цикл прекращается, когда очередной пары кусков нет ($b1 > n$), т.е.

массив полностью переписан. В начале цикла (строчки 5 – 8) на основе значения $b1$ вычисляется конец первого куска ($e1$), начало второго ($b2$) и конец второго куска ($e2$). Проверки (строчки 6 и 8) обеспечивают правильные значения этих переменных на границе массива, когда длина первого или второго куска меньше, чем l. Второй кусок может иметь нулевую длину, тогда $e2 = b2 - 1$. Собственно слияние двух кусков производится через вызов процедуры S (строчка 9).

Полное доказательство правильности предлагается читателю провести самостоятельно.

Для вычисления трудоемкости алгоритма заметим, что после m-го этапа величина l равна 2^m (доказать самостоятельно!). Работа заканчивается, когда $l \geq n$, поэтому количество переписываний $m = \lceil \log_2 n \rceil$, где угловые скобки отмечают округление до ближайшего целого сверху. Так как трудоемкость одного переписывания равна n, то общая трудоемкость C будет равна в шагах цикла слияния:

$$C = n\lceil \log_2 n \rceil. \tag{3.1}$$

Сравним трудоемкость алгоритма сортировки слиянием с алгоритмом 3.4 (с сортировкой выбором) и с алгоритмом 3.1 (с простейшей сортировкой). Следует учесть, что в наихудшем случае один шаг в цикле сортировки слиянием, в котором выполняется три сравнения, по времени примерно соответствует трем шагам в цикле сортировки выбором. Таким образом, будем сравнивать значения функций $3n\lceil \log_2 n \rceil$, $n(n-1)/2$ и $n^3/6$.

Таблица 3.1. Число сравнений в трех алгоритмах сортировки

n	$3n\lceil \log_2 n \rceil$	$n(n-1)/2$	$n^3/6$
10	120	45	166
100	2100	4950	$1.66 \cdot 10^5$
1000	$3 \cdot 10^4$	$5 \cdot 10^5$	$1.66 \cdot 10^8$
10000	$4.2 \cdot 10^5$	$5 \cdot 10^7$	$1.66 \cdot 10^{11}$
100000	$5 \cdot 10^6$	$5 \cdot 10^9$	$1.66 \cdot 10^{14}$

Из таблицы видно, что для малых размерностей выигрывает более простой алгоритм сортировки выбором, однако с ростом n кар-

тина резко меняется: при $n = 100000$ алгоритм сортировки слиянием быстрее в тысячу раз! Что касается алгоритма простейшей сортировки, то он плох всегда, а при $n = 100000$ медленнее алгоритма сортировки слиянием в 30 миллионов раз. Как легко, оказывается, мощный компьютер превратить в неповоротливого тугодума, если бездумно использовать плохие алгоритмы! Но ничего не дается даром: алгоритму сортировки слиянием требуется дополнительный массив из n элементов.

3.3. Поиск

Задача поиска формулируется следующим образом. Задан массив A из n элементов и поисковое значение p. Требуется найти такой номер i элемента массива, для которого $A[i] = p$. В результате поиска может получиться одна из трех ситуаций:

1) существует единственный элемент с номером i, для которого $A[i] = p$;

2) $A[i] \neq p$ при любых $i = 1, 2, \ldots, n$;

3) существует несколько элементов с номерами $i_1, i_2, \ldots i_m$ таких, что $A[i_1] = p$, $A[i_2] = p$, ..., $A[i_m] = p$.

В первом и третьем случае поиск считается успешным, причем в третьем случае очень часто достаточно найти *хотя бы один* элемент, равный поисковому значению. В дальнейшем будем считать, что если поиск успешный, то $1 \leq i \leq n$, а если нет, то $i = 0$.

Для неупорядоченного массива задачу поиска *одного элемента* решает следующий однопроходный алгоритм.

Алгоритм 3.6. Поиск в неупорядоченном массиве A из n элементов первого среди значений $A[i]$, равного p. Если такого значения нет, то $i = 0$.

```
i := 0; k := 1;
while k <= n do
  if A[k] = p then begin i := k; k := n + 1 end
  else k:=k+1;
```

▷ **Доказательство правильности.**

Завершимость алгоритма очевидна, т.к. i при каждом выполнении цикла i увеличивается либо на 1, либо сразу до $n + 1$. После окончания цикла $k = n + 1$.

Правильность постусловия. Предусловие: $i = 0, k = 1$. Рассмотрим два случая:

1) поискового значения в массиве нет, тогда проверка $A[k] = p$ всегда будет давать ложь, поэтому постусловие $i = 0$, $k = n + 1$;

2) пусть $A[1] \neq p$, $A[2] \neq p$, ..., $A[l-1] \neq p$, $A[l] = p$, тогда проверка $A[k] = p$ будет давать ложь для всех $k < l$, цикл будет продолжаться, и только для $k = l$ проверка даст истину, после чего цикл закончится с постусловием $i = l$, $k = n + 1$. ◁

В алгоритме 3.6 для наихудшего случая цикл поиска будет выполняться n раз.

Оказывается, поиск можно значительно ускорить, если массив упорядочен. Идея быстрого алгоритма поиска состоит в том, что в массиве выделяется область "подозрительных" элементов, т.е таких, что если искомый элемент в массиве существует, он обязательно будет внутри этой области. Каждое сравнение выполняется таким образом, чтобы область сокращалась вдвое. Пусть область "подозрительных" элементов $A[b], A[b+1], \ldots, A[e]$. Вначале $b = 1$, $e = n$. Сравнение поискового значения p производится с элементом $A[c]$, где $c = (a + b)\mathbf{div}\ 2$, т.е. с элементом в середине области. При этом возможны два исхода (см. рис.3.1):

1) $A[c] < p$. Искомый элемент, если он существует, расположен среди $A[c+1], \ldots, A[e]$.

2) $A[c] \geq p$. Искомый элемент, если он существует, расположен среди $A[b], \ldots, A[c]$.

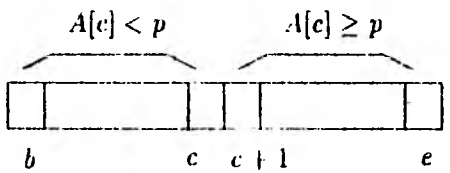


Рис. 3.1

В первом случае можно положить $b = c + 1$, во втором $e = c$. Сравнения следует продолжать, пока "подозрительная" область не сожмется до одного элемента. После этого придется провести срав-

нение с этим единственным элементом, так как может оказаться, что в массиве нет ни одного элемента, равного p.

Так как каждое сравнение уменьшает область в два (или почти в два) раза, алгоритм получил название "дихотомический поиск", или "поиск методом деления пополам".

Алгоритм 3.7. Дихотомический поиск в упорядоченном массиве A из n элементов значения $A[i]$, равного p. Если такого значения нет, то $i = 0$.

```
b := 1; e := n;
while b < e do
begin c := (b + e)div 2;
  if A[c] < p then b := c + 1 else e := c
end;
if A[b] = p then i := b else i := 0;
```

▷ **Доказательство правильности.**

Завершимость алгоритма. Если область содержит более двух элементов, то $b < c < e$ и при любом варианте сравнения размеры области уменьшаются. Если же область содержит ровно два элемента, то $c = b$ и в этом случае при любом варианте сравнения становится $b = e$, после чего цикл прекращается.

Инвариант цикла "область элементов $A[b], A[b+1], \ldots, A[e]$ подозрительна". (Доказать самостоятельно.) После окончания цикла эта область занимает только один элемент $A[b]$, поэтому при истинности проверки $A[b] = p$ результат i будет равен номеру искомого элемента, а в противном случае, когда искомого элемента в массиве нет, $i = 0$. ◁

Оценим трудоемкость дихотомического поиска. Пусть целое m таково, что $2^{m-1} < n \leq 2^m$. Так как при четном n размеры области уменьшаются в два раза, а при нечетном — в два с округлением в большую или меньшую сторону, то понадобится не более m сравнений для того, чтобы длина области стала равна 1. Учитывая еще одно сравнение после цикла, получим общее количество сравнений C в наихудшем случае:

$$C = \lceil \log_2 n \rceil + 1. \qquad (3.2)$$

Таким образом, алгоритм дихотомического поиска работает на-

много быстрее последовательного просмотра. Так, $C = 5$ при $n = 10$, $C = 11$ при $n = 1000$, $C = 18$ при $n = 100000$.

На практике чаще всего массивы сортируют как раз для того, чтобы ускорить последующий *многократный* поиск. Заметим, что если поиск будет проводиться всего один раз, то, сортируя массив, мы получим не ускорение, а замедление работы, так как трудоемкость лучших алгоритмов сортировки $n \log_2 n$ шагов, что намного больше трудоемкости поиска в неупорядоченном массиве.

Оба рассмотренных алгоритма поиска находят только один элемент массива, равный поисковому значению, причем с наименьшим номером. Для алгоритма последовательного просмотра это очевидно, для дихотомического поиска это легко доказать. Иногда требуется найти *все* такие элементы. Нетрудно догадаться, что в упорядоченном массиве они будут располагаться *подряд*, поэтому их легко найти просмотром после дихотомического поиска.

3.4. Косвенная упорядоченность и поиск в таблицах

Иногда необходимо так упорядочить массив, чтобы сохранить и исходный порядок элементов. Решению этой задачи поможет *косвенная упорядоченность*, когда в дополнение к массиву A создается вспомогательный массив индексов S, содержащий такую перестановку чисел $1, 2, \ldots, n$, что

$$A[S[1]] \le A[S[2]] \le \ldots A[S[n]]. \qquad (3.3)$$

Алгоритмы сортировки и поиска легко переделать для косвенной упорядоченности. Для этого, во-первых, задается массив индексов S, элементам которого вначале присваиваются числа $1, 2, \ldots, n$, так что $S[i] = i$. Во-вторых, все сравнения с элементом $A[i]$ заменяются на сравнения с элементом $A[S[i]]$, а вместо обмена значениями элементов $A[i]$ и $A[j]$ при сортировке делается обмен значениями элементов $S[i]$ и $S[j]$. Поэтому при модификации алгоритма сортировки слиянием для косвенной упорядоченности кроме исходного массива сортируемых элементов понадобятся два массива ссылок.

Особенно удобно использовать косвенную упорядоченность при работе с данными, представляющими *таблицу* (не путать с массивом!). Таблица содержит данные, размещенные по строкам и столбцам, и в этом походит на двумерный массив. Данные в одном столбце

должны быть одного и того же типа, в разных же столбцах данные могут быть различных типов. Например, в одном столбце — целые числа, в другом — строки символов. Столбцам в таблице приписывают *названия* (имена), строки таблицы обычно нумеруют целыми числами. Пусть, например, задан список учащихся (табл. 3.2).

Таблица 3.2. Список учащихся

N	Учащийся	Рост	Вес	История	Алгебра
1	Иванов Иван	170	68.5	5	2
2	Петров Петр	190	61	4	4
3	Сидоров Сергей	168	79	4	5
4	Макарова Мария	176	61.4	5	3

Данные таблицы можно разместить в нескольких массивах: каждый столбец в своем массиве. Список учащихся разместим следующим образом: данные столбца "учащийся" — в массиве A, "рост" — в B, "вес" — в C, "история" — в D, "алгебра" — в E. Номера строк размещать нигде не нужно, так как переставлять строки в таблице мы не будем. Данные строки номер i представляются набором следующих значений этих массивов: $\{i, A[i], B[i], C[i], D[i], E[i]\}$.

При практической работе с таблицами возникает множество задач. Вот некоторые примеры для таблицы "Список учащихся":

1) выдать фамилии и имена учащихся, получивших пятерки по истории,

2) выдать фамилии и имена учащихся, имеющих четверки и пятерки по всем предметам;

3) вычислить средний вес учащихся;

4) вычислить средний балл по истории для тех, кто имеет "четыре" и "пять" по алгебре.

5) выдать фамилии и имена учащихся, рост которых находится в пределах от 170 до 180 см.

Простой алгоритм решения какой-либо из перечисленных задач можно написать в виде однопроходного с одним циклом, трудоемкость его определяется числом строк в таблице. Если таблица занимает не 4, а, например, 4000 строк, то для быстрого поиска в

ней данные каждого столбца следует косвенно упорядочить, используя вспомогательные массивы индексов, для каждого столбца свой индекс. Тогда можно проводить быстрый поиск по данным какого-либо столбца. Следует лишь учесть, что в большинстве задач нельзя ограничиться поиском только одного элемента массива, как правило, необходимо отыскивать *все* элементы, удовлетворяющие условию задачи.

Возможности косвенной упорядоченности рассмотрим на примере следующей задачи.

Задача "спортивное двоеборье". Входные данные:

1) таблица спортсменов (строка содержит порядковый номер и фамилию спортсмена), количество строк в таблице n;

2) таблица результатов соревнований по первому виду двоеборья (строка содержит номер спортсмена согласно первой таблице и его баллы по первому виду спорта), количество строк в таблице $n1$, причем $n1 \leq n$;

3) таблица результатов соревнований по второму виду двоеборья (строка содержит номер спортсмена согласно первой таблице и его баллы по второму виду), количество строк в таблице $n2$, $n2 \leq n$.

Вторая и третья таблицы могут содержать меньше строк, чем первая, потому что некоторые спортсмены могли не участвовать в каком-либо из видов соревнований. Кроме того, строки во второй и третьей таблицах *не упорядочены!*

Требуется определить места, занятые спортсменами в двоеборье, *по минимальной сумме мест в отдельных видах.*

Примечания.

1) Считается, что чем больше баллов по одному из видов набрал спортсмен, тем более высокое место по *этому* виду он занял.

2) Определять победителя по *сумме баллов* двух видов нельзя: это все равно, что к килограммам прибавлять метры.

3) В общий зачет должны входить только те спортсмены, которые участвовали в обоих видах.

4) Если несколько спортсменов набрали одинаковое число баллов по одному из видов, то им следует приписать *одинаковое среднее место* по этому виду. Например, четыре спортсмена, занявшие ме-

ста после второго, набрали одно и то же число баллов. Тогда следует считать, что все они имеют место $(3 + 4 + 5 + 6)/4 = 4.5$. Место следующего за ними спортсмена — 7.

5) Если у нескольких спортсменов одинаковая сумма мест по обоим видам, то им следует приписать *одинаковое среднее место* по двоеборью.

6) Все данные о спортсменах сведем в одну таблицу, см. пример в табл. 3.3.

Теперь, прежде чем продолжать чтение, попробуйте самостоятельно составить алгоритм решения задачи.

Таблица 3.3. Пример соревнований по двоеборью

N	Спортсмен	Балл вид 1	Балл вид 2	Место вид 1	Место вид 2	Сумма мест	Общее место
1	Абрамов	105.5	8	3.5	2	5.5	2
2	Волков	110	7.2	2	3	5	1
3	Зайцев	120	6.3	1	5	6	3.5
4	Иванов	–	7.1	–	4	–	–
5	Петров	99	11	5	1	6	3.5
6	Сидоров	105.5	–	3.5	–	–	–
7	Тимофеев	83.5	4.3	6	6	12	5

Решение задачи "спортивное двоеборье". Чтобы было проще работать с пропущенными значениями в таблице, условимся считать, что:

1) нуль баллов означает, что спортсмен не участвовал по одному из видов спорта;

2) очень большое занятое место (больше чем n, например, 10000 и больше) означает, что спортсмен выбыл из борьбы

Данные столбцов таблицы разместим в следующих массивах: фамилии спортсменов — в массиве F, баллы по 1-му виду — в $B1$, остальные данные — соответственно в массивах $B2$, $M1$, $M2$,

Ms, *M0*. Длина массивов должна быть достаточна для размещения n элементов. Пусть тип данных для этих массивов определен как

type *mreal* =**array**[1 . . 1000]**of** *real*;

Алгоритм запишем "по частям", вначале — ввод данных.

```
{Ввод исходных данных. Конец ввода — по значению "0"}

n := 0; p := 1;
writeln('Введите фамилии спортсменов ');
while p = 1 do
begin readln(str);
 if str =' 0' then p := 0
 else begin n := n + 1; F[n] := str end
end;

for i := 1 to n do   {заполнение баллов нулями}
 begin B1[i] := 0;  B2[i] := 0 end
end;

p := 1;
writeln('Введите баллы по 1-му виду');
while p = 1 do
begin write('Номер спортсмена, баллы: '); readln(i, b);
 if i = 0 then p := 0 else B1[i] := b
end;

p := 1;
writeln('Введите баллы по 2-му виду');
while p = 1 do
begin write('Номер спортсмена, баллы: '); readln(i, b);
 if i = 0 then p := 0 else B2[i] := b
end;
```

Для косвенной сортировки нам потребуется вспомогательный массив ссылок S. Его тип определим как

type *mint* =**array**[1 . . 1000]**of** *integer*;

Будем считать, что имеется процедура косвенной сортировки по возрастанию: $ssort1(A,S,n)$, где A — массив значений, S — массив ссылок, n — их длина. Имеется также процедура косвенной сортировки по убыванию $ssort2(A,S,n)$. Кроме того, имеется процедура вычисления мест $place(A,S,M,n)$, которая из массива значений A, отсортированного ссылками S, формирует в массиве M места, занятые каждым спортсменом.

```
for i := 1 to n do S[i] := i;  {первоначальное заполнение ссылок}
ssort2(B1,S,n);      {косвенная сортировка по убыванию}
place(B1,S,M1,n);                    {подсчет мест по виду 1}

for i := 1 to n do S[i] := i;  {первоначальное заполнение ссылок}
ssort2(B2,S,n);      {косвенная сортировка по убыванию}
place(B2,S,M2,n);                    {подсчет мест по виду 2}

for i := 1 to n do Ms[i] := M1[i] + M2[i];
                                {вычисление суммы мест}

for i := 1 to n do S[i] := i;  {первоначальное заполнение ссылок}
ssort1(Ms,S,n);  {косвенная сортировка по возрастанию}
place(Ms,S,M0,n);                    {подсчет общих мест}
```

Рассмотрим алгоритм процедуры $place(A,S,M,n)$. Если бы не было одинаковых и пропущенных значений в A, то было бы достаточно присвоить $M[S[k]] := k$ для всех $k = 1,2,\ldots,n$.

В алгоритме внешний цикл **while** выделяет очередную группу одинаковых элементов массива баллов A по косвенной упорядоченности S. Переменная $i1$ отмечает начало этой группы, переменная i, увеличивая значение во внутреннем цикле **while**, отмечает конец этой группы. Далее в цикле **for** элементам массива M, которые соответствуют этой группе, присваивается среднее место. Особо вы-

деляется случай, когда количество баллов нулевое: тогда присваивается место 10000.

```
procedure place(var A : mreal;var S : mint;
        var M : mreal; n : integer);
var i, i1, k : integer;
begin i := 1;
  while i <= n do
  begin i1 := i;
    while (i < n)and(A[S[i]] = A[S[i + 1]]) do i := i + 1;
            {выделена группа с одинаковыми местами с i1 по i}
    for k := i1 to i do
      if A[S[i1]] > 0 then M[S[k]] := (i1 + i)/2.0;
      else M[S[k]] := 10000;
    i := i + 1
  end
end
```

Задания для самостоятельной работы

1. Доказать правильность алгоритма 3.3 — улучшенной сортировки обменом.

2. Какой из алгоритмов сортировки (3.3 или 3.4) работает быстрее в наихудшем случае и почему? Обосновать ответ.

3. Записать алгоритм "пузырьковой сортировки", доказать его правильность и определить трудоемкость. Сравнить с другими алгоритмами сортировки.

4. Записать на Паскале текст процедуры S для алгоритма сортировки слиянием.

5. Написать на Паскале три программы, каждая из которых вводит размер n, генерирует массив из n случайных чисел и сортирует его (первая — алгоритмом 3.1, вторая — алгоритмом 3.4, третья — алгоритмом 3.5). Испытать на компьютере эти алгоритмы при различных n (например, для $n = 100$, $n = 500$, $n = 1000$, $n = 10000$) и сравнить между собой время их работы. Для определения времени в Турбо-Паскале можно использовать процедуру *gettime* из библиотеки *Dos*.

6. Доказать, что алгоритм дихотомического поиска отыскивает элемент массива *с наименьшим номером* среди всех, равных поисковому значению.

7. Преобразовать алгоритм дихотомического поиска так, чтобы он отыскивал элемент массива *с наибольшим номером* среди всех, равных поисковому значению. Доказать правильность алгоритма.

8. Дополнить алгоритм дихотомического поиска так, чтобы он отыскивал *все* элементы массива, равные поисковому значению. Доказать правильность алгоритма.

9. В массиве из n элементов требуется m раз произвести поиск. Определить, что лучше: 1) выполнять поиск без упорядочения массива; 2) вначале упорядочить массив улучшенной сортировкой обменом, а затем выполнять поиск.

10. Решить задачу 9 для сортировки слиянием.

11. Преобразовать один из простых алгоритмов сортировки так, чтобы он осуществлял косвенную сортировку.

12. Модифицировать алгоритм сортировки слиянием так, чтобы он осуществлял косвенную сортировку.

13. Преобразовать алгоритм дихотомического поиска так, чтобы он отыскивал элемент в массиве с косвенной упорядоченностью.

14. Преобразовать алгоритм задания 8 так, чтобы он отыскивал элементы в массиве с косвенной упорядоченностью.

15. Написать алгоритм решения понравившейся вам задачи для таблицы "Список учащихся".

16. Доказать правильность процедуры *place*.

17. Дописать на Паскале программу решения задачи "спортивное двоеборье": вставить ввод исходных данных из файлов, вывод результатов в удобной форме. Определить трудоемкость алгоритма в зависимости от n.

Глава 4. РЕКУРСИВНЫЕ АЛГОРИТМЫ И БЭКТРЕКИНГ

Алгоритм можно всегда записать в виде процедуры (или процедуры-функции), зависящей от входных параметров. Более-менее сложный алгоритм содержит в себе алгоритмы второго уровня, которые внутри него можно записать в виде вызовов процедур.

Алгоритм можно записать даже так, чтобы он вызывал сам себя! Такой алгоритм называют *рекурсивным*. На первый взгляд кажется, что получится порочный круг и алгоритм на компьютере никогда не выполнится до конца. Действительно, при выполнении алгоритм вызывает сам себя, при выполнении этого вызова он снова вызывает сам себя и так далее. Чтобы последовательность вызовов когда-нибудь закончилась, предусматривают, чтобы при каждом вызове хотя бы один входной параметр *изменялся*. В алгоритме производится проверка этого параметра для того, чтобы при определенном значении повторный вызов *не исполнялся*.

Алгоритмы решения многих задач можно записать в рекурсивном виде. Часто одна и та же задача может быть решена как рекурсивным, так и нерекурсивным алгоритмом. Для некоторых задач рекурсивный алгоритм можно легко придумать, в то время как нерекурсивный алгоритм просто не удается создать.

Существуют задачи, для решения которых необходимо просматривать большое количество вариантов. Рекурсивный алгоритм решения такой задачи после выбора и проверки некоторого варианта отбрасывает его, чтобы выбрать другой вариант. Это действие — ход назад (по-английски *backtraking*) определило название класса алгоритмов: *бэктрекинг*.

4.1. Рекуррентные последовательности и рекурсивные алгоритмы

Рекурсивный алгоритм можно записать непосредственно по определению рекуррентной последовательности. Для примера рассмо-

трим определение факториала:

$$\begin{cases} n! = 1, & n = 0, \\ n! = n(n-1)!, & n > 0. \end{cases}$$

Алгоритм 4.1. Рекурсивный алгоритм вычисления факториала.

```
function F(n : integer) : integer;   {1}
begin                                {2}
  if n = 0 then F := 1               {3}
  else F := n * F(n - 1)             {4}
end                                  {5}
```

Заметим, что в записи рекурсивного алгоритма оператора цикла нет, хотя вычисления носят циклический характер.

Доказательство правильности подобного алгоритма основывается на доказательстве корректности самого определения рекуррентной последовательности. Требуется лишь удостовериться, что при записи алгоритма в него не привнесли ошибок.

Записать рекурсивный алгоритм легко, однако чтобы выяснить его эффективность, необходимо разобраться, как этот алгоритм выполняется *исполнителем*. Пусть в программе имеется вызов:

$$d := F(3);$$

Вычисления будут следующими:

1) вызывается функция F и переменная n заменяется на 3;

2) проверка в операторе {3} дает ложь, вычисляется {4}, в ходе чего вызывается $F(2)$;

3) проверка в операторе {3} снова дает ложь, вычисляется {4}, т.е. вызывается $F(1)$;

4) проверка в операторе {3} еще раз дает ложь, вычисляется {4}, и теперь вызывается $F(0)$;

5) проверка в операторе {3} дает истина, вычисляется $F := 1$;

6) $F(n-1) = 1$ при $n = 1$ подставляется в формулу $F := n * F(n-1)$, в результате $F = 1$;

7) теперь $F(n-1) = 1$ уже при $n = 2$ подставляется в формулу $F := n * F(n-1)$, в результате новое $F = 2$;

8) новое $F(n-1) = 2$ при $n = 3$ подставляется в формулу $F := n * F(n-1)$, окончательный результат $F = 6$.

9) наконец, вычисленное значение $F = 6$ присваивается переменной d при первом вызове.

В первых четырех шагах вычисления как бы идут "вглубь", а в следующих четырех — возвращаются "вверх".

Для запоминания промежуточных значений в последовательности рекурсивных вызовов исполнителю требуется память. В качестве такой памяти используется *стек* — структура данных со специальным доступом. Стек можно представить как массив неограниченной длины и переменную — указатель стека. Со стеком можно производить операции двух видов: записывать в стек и извлекать из стека некоторое значение. Если массив стека обозначить как L, а указатель стека — как p, то операции будут выглядеть следующим образом:

1) $p := p + 1$; $L[p] := x$; — запись в стек значения x;

2) $x := L[p]$; $p := p - 1$; — извлечение из стека значения и запись его в переменную x.

До начала выполнения алгоритма $p = 0$, т.е. стек считается пустым.

При вычислении $F(3)$ в стеке размещаются те значения n, для которых вызывается $F(n)$, а также вычисленные значения F. Заполнение стека и извлечение значений из него показано на рис. 4.1.

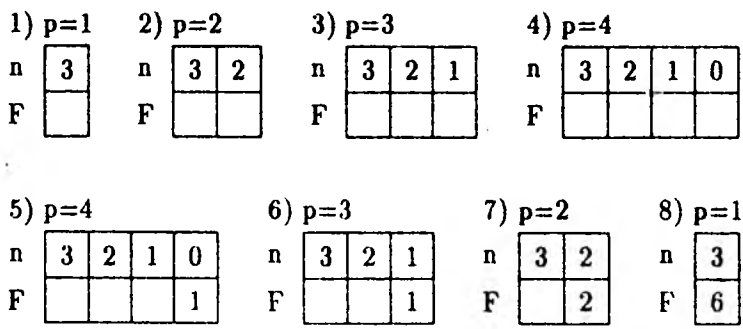


Рис. 4.1

Итак, для выполнения рекурсивного алгоритма требуется дополнительная память для стека, хотя программист может и не знать об этом. Если выделенная исполнителем память для этого недостаточна, то алгоритм не может быть выполнен до конца: произойдет

его аварийное завершение и результат вычислений не будет получен. Поэтому важно уметь оценивать объем памяти, необходимой для стека.

При каждом рекурсивном вызове требуется разместить в стеке следующую информацию:

1) значения параметров процедуры,

2) возвращаемое значение,

3) *внутренние* переменные в процедуре (если они есть),

4) *адрес возврата* — номер команды в программе, которая расположена сразу за командой вызова процедуры, и куда следует вернуться после выполнения процедуры.

Число таких размещений равно *глубине рекурсии* — количеству рекурсивных вызовов до получения первого результата процедурой. Общий объем памяти равен произведению глубины рекурсии на количество памяти, необходимой для выполнения одного вызова процедуры. Так, например, для рассмотренного алгоритма глубина рекурсии равна $n+1$, поэтому требуемый объем памяти в несколько раз больше, чем память для целочисленного массива длиной $n+1$. Особенно много памяти потребуется, если внутри рекурсивной процедуры определен массив большого размера.

Что касается времени выполнения рекурсивного алгоритма, то ясно, что оно пропорционально *количеству* рекурсивных вызовов. Например, для алгоритма вычисления факториала время пропорционально $n+1$. Если сравнить его с временем выполнения нерекурсивного (циклического) алгоритма вычисления факториала, то фактически время работы рекурсивного алгоритма окажется в 2–3 раза больше, так как каждое повторение цикла требует от исполнителя меньше действий, чем рекурсивный вызов процедуры. Из рассмотренного примера видно, что при прочих равных условиях рекурсивный алгоритм менее эффективен, чем циклический, и единственным его преимуществом является лаконичность определения.

А теперь рассмотрим рекурсивный алгоритм для более сложной рекуррентной последовательности на примере чисел Фибоначчи (см. формулу (2.3)).

Алгоритм 4.2. Рекурсивный алгоритм вычисления чисел Фибоначчи.

```
function Fib(n : integer) : integer;      {1}
begin                                     {2}
  if n = 0 then Fib := 0                  {3}
  else if n = 1 then Fib := 1             {4}
  else Fib := Fib(n - 1) + Fib(n - 2)     {5}
end                                       {6}
```

Глубина рекурсии алгоритма при $n > 0$ равна n, так как рекурсивный вызов в строке {5}: $Fib(n-2)$ происходит *после* завершения вычисления $Fib(n-1)$. Поэтому объем требуемой памяти пропорционален n.

Обозначим количество рекурсивных вызовов $R(n)$. Из алгоритма видно, что числа $R(n)$ подчиняются следующему рекуррентному соотношению:

$$\begin{cases} R(n) = 1, & n = 0,\ n = 1 \\ R(n) = 1 + R(n-1) + R(n-2), & n > 1. \end{cases}$$

Числа $R(n)$ растут гораздо быстрее чисел Фибоначчи. Сравните начальные числа Фибоначчи с начальными числами $R(n)$: *1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, ...* Таким образом, время вычисления оказывается катастрофически большим! Это происходит потому, что при вычислении $R(n-2)$ *не используются* промежуточные результаты, полученные при вычислении $R(n-1)$.

Из всего этого можно сделать вывод о том, что *рекуррентную последовательность следует вычислять с помощью цикла, а не рекурсии!*

4.2. Рекурсивная сортировка слиянием

Тем не менее использование рекурсии вполне оправдано тогда, когда дополнительные расходы памяти и увеличение времени работы алгоритма относительно невелики. Для примера рассмотрим рекурсивный вариант алгоритма сортировки слиянием.

Предположим, что алгоритм слияния 2.7 оформлен в виде процедуры $S(A, B, b1, e1, e2)$ так, как это описано в разделе 3.2 (алгоритм копирует со слиянием два упорядоченных куска из массива A с номерами от $b1$ до $e1$ и от $e1+1$ до $e2$ соответственно в массив B с номерами элементов от $b1$ до $e2$). Тогда алгоритм сортировки можно определить очень просто:

1) если длина сортируемого массива 1, то ничего не делается;

2) в противном случае массив делится на две одинаковые (или почти одинаковые) части, к каждой части применяется алгоритм сортировки, после чего эти упорядоченные части сливаются в один упорядоченный кусок.

Алгоритм 4.3. Рекурсивный алгоритм сортировки слиянием. Константа N определяет максимальную длину массивов. На вход процедуры сортировки поступает неупорядоченный кусок массива A с номерами элементов от b до e, на выходе — тот же кусок, но упорядоченный. Массив C — вспомогательный.

```
type mas =array[1..N]of integer;
procedure rsort(var A,C : mas,b,e : integer);
  var i,d : integer;
begin
  if b < e then
  begin d := (b + e)div 2;
    rsort(A,C,b,d); rsort(A,C,d + 1,e);
    S(A,C,b,d,e);
    for i := b to e do A[i] := C[i]
  end
end
```

Правильность алгоритма легко доказать методом математической индукции, применив постусловие, что на выходе алгоритма кусок массива A с номерами от b до e упорядочен.

Вызов процедуры *rsort* для упорядочения массива X из n элементов (Y — вспомогательный массив):

$$rsort(X, Y, 1, n);$$

Из сравнения алгоритмов 3.5 и 4.3 видно, насколько рекурсивный алгоритм проще нерекурсивного (в котором два вложенных цикла!). Но не слишком ли велика плата за простоту? Оценим требуемую память и время выполнения рекурсивного алгоритма по сравнению с нерекурсивным. Нетрудно видеть, что глубина рекурсии алгоритма 4.3 равна $\lceil \log_2 n \rceil$. Действительно, если $2^{m-1} < n \leq 2^m$, то, чтобы размер упорядочиваемого куска массива в очередном рекурсивном вызове процедуры уменьшился до одного элемента, потре-

буется m вложенных рекурсивных вызовов. Даже с учетом того, что для каждого рекурсивного вызова требуется память для значений 6 переменных, т.е. объем памяти для рекурсии имеет порядок $6\lceil\log_2 n\rceil$, для больших n это намного меньше, чем объем памяти для массивов X или Y.

Время выполнения состоит из времени работы всех вызовов процедуры слияния S, времени собственно рекурсивных вызовов процедуры *rsort* и времени обратного копирования из массива C в массив A. Общее время работы процедуры слияния S совпадает с временем работы этой же процедуры в алгоритме 3.5, которое примерно равно $3n\lceil\log_2 n\rceil$ (см. раздел 3.2). На каждую пересылку при слиянии приходится по одной пересылке при обратном копировании, поэтому общее время на это копирование — $n\lceil\log_2 n\rceil$. Подсчитаем общее количество рекурсивных вызовов. Пусть для простоты $n = 2^m$. Тогда при первом вызове *rsort*, когда длина куска массива 2^m, выполнится два вызова процедуры с длиной кусков по 2^{m-1}, при всех вызовах с длиной кусков по 2^{m-2} — четыре вызова и т.д. Общее число вызовов:

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^m = 2^{m+1} - 1 = 2n - 1,$$

что намного меньше, чем $n\lceil\log_2 n\rceil$, — общего количества копирований элементов массива при всех отдельных слияниях. В целом время выполнения рекурсивного алгоритма оказывается больше, чем у нерекурсивного, процентов на 40–50.

Таким образом, эффективность рекурсивной сортировки слиянием не так уж и плоха по сравнению с нерекурсивной.

4.3. Бэктрекинг

Существует ряд задач, которые без рекурсии решить практически невозможно. К ним относятся, в частности, комбинаторные задачи, в которых ищется решение, состоящее из комбинации значений некоторых переменных. Иногда требуется найти не одно, а несколько, быть может, даже все возможные решения. В качестве примера рассмотрим задачу нахождения всех перестановок натуральных чисел от 1 до n. Как известно, количество перестановок равно $n!$. Задачу будем решать следующим образом:

1) в качестве первого числа выбираем любое из чисел $1, \ldots, n$;

2) в качестве второго числа выбираем любое из чисел $1, \ldots, n$, *кроме того числа, которое выбрано первым*;

3) третьим числом выбираем одно из чисел, которое не выбрано первым или вторым;

...

Этот процесс продолжаем до тех пор, пока не выберем последнее, n-е число для перестановки. Чтобы получить *все возможные* перестановки, надо на каждом этапе выбора k-го числа последовательно перебирать все допустимые числа, однако перейти к следующему варианту можно, только если найдены полные перестановки для всех предыдущих вариантов. Такой процесс можно представить схемой типа изображенной на рис. 4.2 для $n = 3$. Кружками на схеме обозначены "состояния" процесса, стрелками — переход к выбору очередного числа. В каждом из кружков записано выбираемое число.

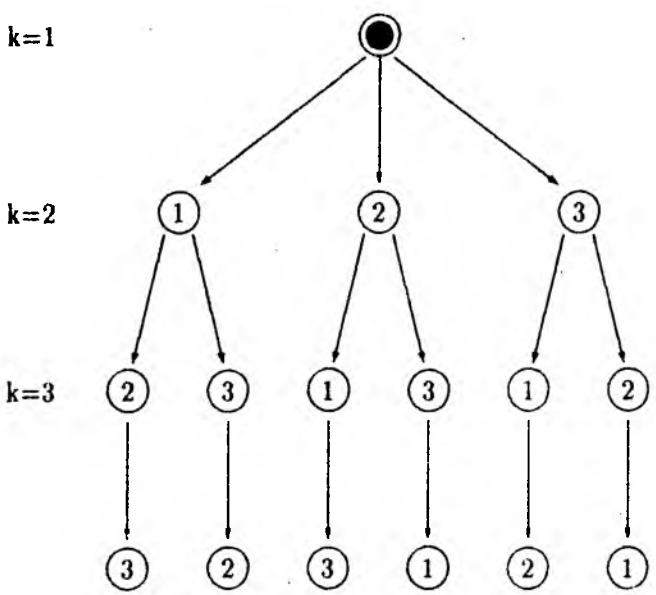


Рис. 4.2

В соответствии со схемой вначале будет выбрана перестановка

1-2-3, затем *1-3-2*, *2-1-3* и т.д. При выборе очередного числа движение по схеме идет по стрелке вниз, а при отказе от этого выбора (чтобы выбрать другое число) — обратное движение (бэктрекинг). Этот процесс легко реализовать рекурсивным алгоритмом.

Алгоритм 4.4. Рекурсивный алгоритм генерации перестановок. Перестановки генерируются в глобальном массиве P, содержащем n элементов. Массив R (тоже из n элементов) содержит признаки включения чисел в перестановку: если $R[i] = 1$, то число i включено в перестановку, а если $R[i] = 0$, то нет. Рекурсивная процедура $per(k)$ выбирает для перестановок все оставшиеся числа, начиная с k-й позиции. Вспомогательная процедура $out(P, n)$ выводит n элементов массива P. Вначале всем элементам массива R присваивается нулевое значение, после чего вызывается $per(1)$.

```
procedure per(k : integer);
  var i : integer;
begin
  for i := 1 to n do
    if R[i] = 0 then
    begin R[i] := 1;  P[k] := i;
      if k = n then out(P, n)
      else per(k + 1);
      R[i] := 0
    end
end
```

Правильность алгоритма нетрудно доказать методом математической индукции. Заметим лишь, что присваивание $R[i] := 0$ после рекурсивного вызова обеспечивает обратный ход — бэктрекинг для поиска другого решения.

Глубина рекурсии алгоритма равна n, так как при каждом вложенном рекурсивном вызове параметр k увеличивается на единицу, а возврат из рекурсии происходит при $k = n$. Величина n не может быть большой из-за огромного количества перестановок (например, $10! = 3628800$), поэтому затраты памяти в алгоритме невелики.

Время выполнения алгоритма определяется, прежде всего, количеством сгенерированных перестановок. Правда, алгоритм делает и "лишнюю" работу: при $k = 2$ из n шагов цикла один будет беспо-

лезным, а при $k = n$ только один шаг цикла из n будет полезным. Однако при $k = n$ этот один шаг перевешивает все бесполезные $n - 1$ шагов, так как при нем выводится *вся* перестановка из n чисел. Практически эта "лишняя" работа увеличивает общее время выполнения незначительно.

Методом бэктрекинга можно решать многие интересные задачи, например, головоломки. В качестве примера рассмотрим одну из них.

Задача "магические числовые квадраты". В квадратной таблице $n \times n$ так расставить числа $1, 2, \ldots, n^2$, чтобы суммы по всем строкам, столбцам и главным диагоналям были одинаковы. На рис. 4.3 приведены примеры магических квадратов для $n = 3$ и $n = 4$. Задача имеет решение лишь для $n \geq 3$.

2	7	6
9	5	1
4	3	8

8	1	6
3	5	7
4	9	2

1	2	15	16
12	14	3	5
13	7	10	4
8	11	6	9

Рис. 4.3

Будем искать *все* возможные магические квадраты для заданного n. Квадратную таблицу "вытянем" в одну строку с нумерацией ее элементов от 1 до n^2. Тогда решением будет такая перестановка чисел $1, 2, \ldots, n^2$, которая удовлетворяет условию магического квадрата. Поэтому за основу алгоритма можно взять алгоритм 4.4 генерации перестановок, дополнив его проверкой условия.

Алгоритм 4.5. Рекурсивный алгоритм генерации магических числовых квадратов. Квадраты генерируются в виде перестановок n^2 чисел в глобальном массиве T. Массив R из n^2 элементов содержит признаки включения чисел в перестановку: если $R[i] = 1$, то

число i включено в перестановку, а если $R[i] = 0$, то нет. Рекурсивная процедура $sq(k)$ выбирает для перестановок все оставшиеся числа, начиная с k-й позиции. Вспомогательная процедура-функция $test(T, n)$ проверяет условие магического квадрата и выдает единицу, если условие выполняется, а процедура $out(T, n)$ выводит n^2 элементов массива T в виде квадрата. Вначале всем элементам массива R присваивается нулевое значение, после чего вызывается $sq(1)$.

```
procedure sq(k : integer);
  var i : integer;
begin
  for i := 1 to n * n
    if R[i] = 0 then
    begin R[i] := 1;  T[I] := i;
      if k = n * n then
      begin
        if test(T, n) = 1 then out(T, n)
      end
      else sq(k + 1);
      R[i] := 0
    end
end
```

Величину суммы для проверки условия определим из следующих соображений. Сумма всех чисел квадрата равна

$$1 + 2 + 3 + \ldots + n^2 = n^2(n^2 + 1)/2.$$

Эта сумма делится на n одинаковых частей (по количеству строк в квадрате). Таким образом, искомая сумма для проверки условия равна $n(n^2 + 1)/2$. Ниже приведена процедура-функция проверки условия магического квадрата. Тип *mas* — массив целых чисел.

В отличие от алгоритма 4.4 здесь решениями будет лишь небольшая часть всех сгенерированных перестановок. Так, для $n = 3$ только 8 перестановок из $9! = 362880$ будут решениями, а для $n = 4$ — только 7040 из $16! = 20922789888000$. На компьютере средней мощности (типа АТ-286) для $n = 3$ алгоритм будет выполняться несколько минут, а для $n = 4$ — несколько сотен лет!

```
function test(var T : mas, n : integer) : integer;
var b, i, j, s, sl : integer;
begin
  b := 1;
  sl := n * (n * n + 1)div 2;
  i := 1;                                {проверка строк}
  while (i <= n)and(b = 1) do
  begin
    s := 0;
    for j := 1 to n do
      s := s + T[(i - 1) * n + j];
    if s <> sl then b := 0;
    i := i + 1
  end;
  j := 1;                                {проверка столбцов}
  while (j <= n)and(b = 1) do
  begin s := 0;
    for i := 1 to n do s := s + T[(i - 1) * n + j];
    if s <> sl then b := 0;
    j := j + 1
  end;
  if b = 1 then
  begin s := 0;      {проверка диагонали слева-направо}
    for i := 1 to n do s := s + T[(i - 1) * (n + 1) + 1];
    if s <> sl then b := 0;
    if b = 1 then
    begin s := 0;    {проверка диагонали справа-налево}
      for i := 1 to n do s := s + T[(i - 1) * (n - 1) + n];
      if s <> sl then b := 0;
    end
  end;
  test := b
end;
```

Попробуем ускорить работу алгоритма. Заметим, что по мере построения очередной перестановки, когда выбраны несколько первых чисел, появляется возможность проверки того, что *все* перестановки, начинающиеся на *эти* числа, бесперспективны, и тогда отсечь все эти варианты перестановок. Это соответствует тому, что при

движении по схеме (см. рис. 4.2) мы из некоторого состояния на одном из верхних уровней сразу же делаем ход назад. В алгоритме же при бесперспективности перестановки просто *не вызывается* рекурсивно процедура $sq(k+1)$. Рассмотрим, как реализовать такие проверки. Во-первых, проверку очередной строки квадрата можно делать сразу, как только она сформирована, условие того, что проверка нужна: $k \bmod n = 0$. Аналогично проверку сформированного столбца квадрата можно делать при $k > n^2 - n$. Наконец, проверку диагонали справа-налево можно делать при $k = n^2 - n + 1$. В результате полных перестановок будет формироваться гораздо меньше. Так, для $n = 3$ при первой проверке вместо 362880 будет формироваться 2592 перестановки, при первой и второй — 72, а при всех трех проверках — только 24 перестановки. В результате время работы алгоритма сократится в тысячи раз. При этом становится реальным вычисление всех магических квадратов и для $n = 4$. К сожалению, этих усовершенствований недостаточно для решения задачи при $n = 5$, число $25! \approx 1.55 \cdot 10^{25}$ останется непомерно большим, даже если его разделить на миллион или миллиард!

Можно изменить задачу: искать не все, а только m *первых* магических квадратов. Тогда требуется после нахождения m первых решений прекратить поиск других решений, для чего произвести завершение всех вложенных рекурсивных вызовов. Это можно сделать следующим образом:

1) завести глобальную переменную z и до вызова процедуры $sq(1)$ присвоить ей нулевое значение;

2) в процедуре sq, как только будет найдено решение, увеличить z на единицу;

3) в процедуре sq цикл **for** выбора следующего числа заменить на эквивалентный цикл **while** с условием: $(i <= n*n)\mathbf{and}(z < m)$.

Это соответствует тому, что при движении по схеме рис. 4.2 после нахождения m решений обратный ход будет выполняться подряд вплоть до выхода на самый верхний уровень и завершения самого первого вызова $sq(1)$.

А теперь кратко рассмотрим общие принципы метода бэктрекинга для решения задач переборного характера:

1) вначале следует определить такие структуры данных (массивы, переменные), которые обеспечат возможность перечисления *всех* возможных вариантов решения задачи;

2) затем записать основную рекурсивную процедуру с циклом, в котором производится перебор вариантов движения вниз на одну ступень по схеме из текущего состояния, либо выдачу сгенерированного очередного варианта;

3) в процедуре предусмотреть проверку корректности полного сгенерированного варианта и по возможности проверку бесперспективности частично сгенерированных вариантов и их отсечения.

В заключение рассмотрим использование метода бэктрекинга для решения еще одной головоломки.

Задача "поиск пути в лабиринте". В прямоугольном массиве T из n строк и m столбцов задан лабиринт. Если $T[i,j] = 0$, то в этой позиции проход есть, если $T[i,j] = 1$ — то нет (стенка). Двигаться можно только в соседнюю клетку по горизонтали или вертикали, если есть проход. Найти путь из некоторой начальной позиции $T[i0, j0]$ в конечную $T[ik, jk]$.

Алгоритм 4.6. Рекурсивный алгоритм поиска пути в лабиринте. В глобальном массиве T из n строк, m столбцов записан лабиринт, путь запоминается в виде координат клеток в глобальных массивах Pi и Pj, текущая длина пути вычисляется в переменной l. Путь также отмечается двойками в массиве T. Вспомогательная процедура *out* выводит найденный путь. Процедура $lab(i,j)$ вычисляет путь от клетки $T[i,j]$ до клетки $T[ik, jk]$.

```
procedure lab(i, j : integer);
begin T[i, j] := 2;  l := l + 1;  Pi[l] := i;  Pj[l] := j;
  if (i = ik)and(j = jk) then out
  else begin
    if (i > 1)and(T[i − 1, j] = 0) then lab(i − 1, j);
    if (i < n)and(T[i + 1, j] = 0) then lab(i + 1, j);
    if (j > 1)and(T[i, j − 1] = 0) then lab(i, j − 1);
    if (j < m)and(T[i, j + 1] = 0) then lab(i, j + 1);
  end
  l := l − 1;  T[i, j] := 0
end
```

Вызов этой процедуры в главной программе: $lab(i0, j0)$. Перед вызовом необходимо счетчику длины пути l присвоить нуль. В про-

цедуре производится перебор всех возможных вариантов движения, на каждом шаге их не более четырех. При выборе очередной клетки для движения проверяется, нет ли там стенки или края поля лабиринта, а также не проходит ли через эту клетку текущий вариант маршрута. Поэтому исключаются маршруты с самопересечениями или с повторением движения по одним и тем же клеткам. Тем самым исключается зацикливание маршрута. Длина любого из найденных маршрутов не превышает количества свободных клеток в лабиринте, которое, в свою очередь, не больше чем $n \cdot m$. Этой же величиной определяется и максимальная глубина рекурсии. Однако общее количество вариантов маршрута может быть очень велико: верхняя граница их количества равна $(n \cdot m)!$. Заметим, что существует принципиально другой алгоритм, гораздо более эффективный. Но он очень сложен, поэтому здесь не обсуждается.

4.4. Логические игры с противником

В алгоритме игры компьютера против человека обычно содержатся следующие три части:

1) диалог с человеком;

2) оценка игровой ситуации;

3) выбор лучшего хода.

При реализации третьей части можно использовать метод бэктрекинга. В качестве примера рассмотрим одну знаменитую игру.

Игра крестики-нолики 3×3. Рабочее поле игры представим в виде массива T из девяти элементов. Первые три элемента соответствуют первому ряду клеток поля, вторые три — второму, третьи три — третьему ряду. Пустая клетка отмечается числом 0, крестик (ход человека) — числом 1, нолик (ход компьютера) — числом 2.

Главная часть алгоритма — диалог с человеком — состоит из цикла, в котором вначале делает ход человек, затем производится проверка игровой ситуации и, если игра не завершена, делает ход компьютер. Переменная d управляет работой цикла. Если $d = 0$, игра продолжается, если $d = 1$ — выиграл человек, если $d = 2$ — выиграл компьютер, если $d = 3$ — ничья. Переменная s содержит количество заполненных клеток.

Вспомогательные процедуры алгоритма:

in(q) — вводит ход человека, переменная q — номер элемента $T[q]$, куда человек ставит крестик;

out(*q*) — выводит ход $T[q]$, сделанный человеком или компьютером, на экран;

test(*q*) — вычисляет оценку ситуации после хода $T[q]$: 0 — если игра не закончена, 1 — если выиграл человек, 2 — если выиграл компьютер, 3 — если ничья;

step(g, q, k, d) — выбирает лучший ход $T[q]$ для игрока g ($g = 1$ для человека, $g = 2$ для компьютера), d — прогноз ситуации (как в процедуре *test*), k — количество оставшихся ходов для прогнозируемой ситуации.

```
{********* Алгоритм диалога *********}
for i := 1 to 9 do T[i] := 0;
d := 0;  s := 0;
while d = 0 do
begin in(q);
  T[q] := 1;  s := s + 1;
  out(q);
  d := test(q);
  if d = 0 then
  begin step(2, q, k, d);
    T[q] := 2;  s := s + 1;
    out(q);
    if k > 1 then d := 0
  end
end;
```

От реализации процедур *in*(*q*) и *out*(*q*) зависит внешний вид игры, удобство ее практического применения. Здесь читатель может воплотить самую смелую фантазию: сделать экранный ввод-вывод в графическом режиме с управлением клавишами стрелок или даже мышью, дополнить диалог различными звуковыми и динамическими эффектами и т.п.

Процедура *test*(*q*) вычисляет номер игрока, сделавшего ход $T[q]$, определяет номер строки и столбца, куда был сделан ход, и проверяет наличие выигрышной для этого игрока комбинации. Если таковой нет и все клетки заняты, то результатом работы процедуры будет ничья (число 3).

Для выбора очередного хода можно использовать самые разные игровые стратегии. Наука *теория игр* в качестве наилучшей (оптимальной) предлагает *минимаксную* стратегию. Эта стратегия

очень проста: необходимо *минимизировать максимально* возможный выигрыш противника.

В процедуре *step* минимаксная стратегия реализуется методом бэктрекинга. Работу процедуры можно проиллюстрировать все той же схемой на рис. 4.2. В начальном состоянии последовательно перебираются все незанятые клетки поля. Для каждого возможного хода в незанятую клетку может быть одна из двух ситуаций: 1) игра закончена; 2) игра продолжается. Во втором случае вызывается процедура *step*, но уже для определения наилучшего хода с точки зрения противника. Наилучший ход определяется по правилам:

1) если среди возможных ходов есть такой, после которого гарантируется выигрыш, то выбирается ход, при котором выигрыш будет достигнут за *наименьшее* число последующих ходов;

2) если среди ходов есть только такие, после которых может быть ничья или проигрыш, то выбирается ход, гарантирующий ничью с *максимальным* числом последующих ходов;

3) если среди ходов все проигрышны , то выбирается ход с *максимальным* числом последующих ходов.

При рекурсивном вызове процедуры для определения наилучшего хода с точки зрения противника применяется такая же стратегия, т.е. всегда игроком выбирается наилучший ход для наилучшего со стороны противника ответа.

В теории игр доказано, что минимаксная стратегия оптимальна: игрок, придерживающийся этой стратегии, получит лучший результат, чем игрок, от нее отклоняющийся. Однако для большинства игр такую стратегию применить не удается. Для игры крестики-нолики 3×3 минимаксную стратегию можно реализовать только потому, что перебор сравнительно невелик: после первого хода человека глубина рекурсии просмотра ответного хода 8, поэтому общее число просматриваемых комбинаций $8! = 40320$. Такой перебор может занять несколько секунд времени на компьютере средней мощности.

Если мы попытаемся применить минимаксную стратегию для более сложной игры, то сразу же столкнемся с такими огромными числами комбинаций, что реализация игры потеряет всякий смысл и никакие супермощные компьютеры не спасут положение. Поэтому обязательно следует придумать и реализовать способы отсечения заведомо плохих вариантов ответных ходов. К сожалению, универ

сальных способов отсечения не существует, для каждой игры нужно придумывать свои.

```
{********* Алгоритм оценки ситуации *********}
function test(q : integer) : integer;
  var ii, i, j, g, te : integer;
begin g := T[q];                          {кто сделал ход?}
  i := (q + 2)div 3;  j := (q - 1)mod 3 + 1;
  ii := (i - 1) * 3 + 1;          {i — строка, j — столбец}
  te := 0;
  if (T[ii] = g)and(T[ii + 1] = g)and(T[ii + 2] = g)
  then te := g              {если строка занята игроком g}
  else if (T[j] = g)and(T[j + 3] = g)and(T[j + 6] = g)
  then te := g              {если столбец занят игроком g}
  else begin
    if (q = 1)or(q = 5)or(q = 9) then
    begin                   {проверка диагонали слева-направо}
      if (T[1] = g)and(T[5] = g)and(T[9] = g) then te := g
    end;
    if (q = 3)or(q = 5)or(q = 7) then
    begin                   {проверка диагонали справа-налево}
      if (T[3] = g)and(T[5] = g)and(T[7] = g) then te := g
    end;
  end;
  if (te = 0)and(s = 9)then te := 3;     {проверка ничьей}
  test := te
end;
```

Часто даже с отсечениями обычно остается слишком много перебираемых комбинаций, поэтому для ограничения времени расчета ответного хода необходимо искусственно ограничить глубину рекурсии (с помощью глобальной переменной — счетчика) Качество стратегии от этого, конечно, пострадает, но зато и у человека появится шанс выиграть!

Иногда для некоторой игры удается провести тонкий анализ игровой ситуации, оценивая ее в баллах. Тогда можно или совсем отказаться от рекурсии, или после нескольких рекурсивных вызовов использовать оценку ситуации. Тем самым можно при ограничении глубины просмотра улучшить качество выбора ответного хода.

Именно так реализуют, например, шахматные программы.

```
{********* Алгоритм выбора лучшего хода *********}
procedure step(g : integer;var j, q, d : integer);
  var i, d1, g1, q1, j1 : integer;
begin g1 := 3 - g;  d := g1;  q := 0;
  for i := 1 to 9 do
    if T[i] = 0 then
    begin
      T[i] := g;  s := s + 1;                    {пробуем сделать ход}
      d1 := test(i);  q1 := 0;
      if d1 = 0 then      {не конец игры, пусть ходит противник}
        step(g1, j1, q1, d1);
      {***                анализ хода противника                ***}
      if d1 = g then                                  {наш выигрыш}
      begin
        if d <> d1 then              {выигрыш, которого не было}
          begin d := d1;  q := q1 + 1;  j := i end
        else if q1 + 1 < q then  {выигрыш был, сейчас быстрее}
          begin q := q1 + 1;  j := i end
      end
      else if d1 = 3 then                                  {ничья}
      begin
        if d <> g then                {ничья и не было выигрыша}
        begin
          if d = g1 then                        {ничья, был проигрыш}
            begin d := d1;  q := q1 + 1;  j := i end
          else if q1 + 1 > q then  {ничья была, но сейчас дольше}
            begin q := q1 + 1;  j := i end
        end
      end
      else                                                {проигрыш}
        if (d = g1)and(q1 + 1 > q) then  {но сейчас больше ходов}
          begin q := q1 + 1;  j := i end
      {***            запомнен лучший вариант хода              ***}
      T[i] := 0;  s := s - 1;                              {ход назад}
    end
end
```

Задания для самостоятельной работы

1. Записать рекурсивный вариант однопроходных алгоритмов 2.1 —2.4.

2. Реализуйте рекурсивный вариант алгоритма дихотомического поиска (см. алгоритм 3.6).

3. Дополните задание 5 главы 3 алгоритмом 4.3 (рекурсивной сортировкой слиянием).

4. Модифицируйте алгоритм 4.3 (рекурсивную сортировку слиянием) для получения косвенной упорядоченности

5. Рекурсивный алгоритм вычисления значения максимального элемента в массиве методом дихотомии. Если в массиве один элемент, то результатом будет его значение. В противном случае массив делится пополам, для каждой его части вычисляется значение максимального элемента, максимум из которых и будет результатом. Доказать правильность алгоритма. Подсчитать количество сравнений и глубину рекурсии.

6. Игра "Ханойские башни". Даны три вертикально укрепленных стержня. На первый стержень нанизаны n дисков, все *разного* диаметра, в виде башни: внизу лежит самый большой диск, на нем второй диск поменьше и т.д. Требуется переложить всю башню из дисков с первого стержня на третий (второй используется для промежуточных перекладываний), не нарушая следующих правил: 1) за один шаг перекладывается один диск; 2) диск можно класть либо на пустой стержень, либо на диск большего диаметра. Запишите рекурсивный алгоритм перекладывания башни. Докажите, что n дисков можно переложить за $2^n - 1$ перекладываний.

7. Реализуйте всевозможные отсечения и усовершенствования в алгоритме решения задачи "магические квадраты" и попытайтесь получить хотя бы по одному квадрату для $n = 5$ и $n = 6$.

8. Реализуйте в алгоритме поиска выхода из лабиринта нахождение самого короткого пути. Что будет, если в этом алгоритме убрать присваивание $T[i, j] := 0$ в конце процедуры $lab(i, j)$?

9. Предусмотрите сокращение перебора в алгоритме поиска самого короткого пути в лабиринте.

10. На шахматной доске $n \times n$ клеток поставлено n ладей так, что никакие две из них не "бьют" друг друга. Написать алгоритм получения всех перестановок, доказать его правильность. Чему равно количество перестановок для любого n?

11. Задача 10, но вместо ладей — ферзи. Написать алгоритм и вычислить количество перестановок для всех n вплоть до $n = 10$.

12. Запрограммируйте и отладьте игру "крестики-нолики" 3×3 с удобным графическим диалогом. Придумайте сокращение перебора для ускорения работы алгоритма

13. Разработайте и запрограммируйте алгоритм вашей любимой игры.

Глава 5. ОБРАБОТКА ТЕКСТА И ТРАНСЛЯЦИЯ

Важное значение в современной информатике имеют *символьные* данные и алгоритмы их обработки. В Паскале такие данные представляют как массив символов: **array of** *char*. Кроме того, имеется и специальный тип *string*, с которым можно производить операции целиком, без выделения отдельных символов. Строки можно присваивать, склеивать (операцией +), сравнивать (операциями = < и т.п.). Иногда приходится иметь дело и с массивами элементов типа *string*, т.е. с массивами строк символов. Пример такого массива — список фамилий спортсменов в задаче "спортивное двоеборье" из раздела 3.4.

Чаще всего входные символьные данные сохраняют на диске в виде файлов, поэтому желательно предусмотреть ввод исходных данных из одного файла и вывод обработанных данных в другой файл (см. пример в алгоритме 2.13).

5.1. Сортировка и поиск в массивах строк

При описании переменной типа *string*, так же как и при описании любого другого массива, указывается количество элементов. Иногда это количество *не пишется*, но всегда *подразумевается*, обычно это 255 символов. Это значит, что под строку выделяется память указанного размера. В отличие от других массивов при выполнении действий над строками автоматически контролируется *текущая* длина строки, которая не может быть больше указанной в описании и которая вначале равна нулю. При сравнении строк используется принцип *лексикографического* порядка. Его можно представить в виде следующего алгоритма.

Алгоритм 5.1. Лексикографическое сравнение строк $s1$ и $s2$. Результат: $r = 0$ при $s1 = s2$, $r = 1$ при $s1 > s2$, $r = -1$ при $s1 < s2$. Строки сравниваются посимвольно, с первых символов. Если очередные символы совпали, то сравниваются следующие.

и т.д. Если обе строки закончились, то они считаются равными, а если закончилась, например, только $s1$, то считается, что $s1 < s2$. Если очередные символы не совпали, то меньшей считается та строка, код символа у которой меньше. Стандартная для Паскаля функция *length* вычисляет текущую длину строки, а *ord* — код символа.

```
n1 := length(s1);  n2 := length(s2);
r := 0;  i := 1;
while (i ≤ n1)and(i ≤ n2)and (ord(s1[i]) = ord(s2[i])) do
  i := i + 1;
if (i < n1)and(i < n2) then
begin
  if ord(s1[i]) < ord(s2[i]) then r := -1
  else r := 1
end
else if (i < n1) then r := -1
else if (i < n2) then r := 1
```

Транслятор Паскаля автоматически использует этот алгоритм при выполнении операций сравнения строк, поэтому строки можно сравнивать так же, как, например, целые числа. Аналогично числовому массиву массив строк может быть упорядоченным или нет. При этом алгоритмы сортировки массивов строк внешне ничем не отличаются от аналогичных алгоритмов для числовых массивов. То же самое относится и к алгоритмам поиска.

Однако при работе со строками возникает следующая проблема. Как известно, в текстах на естественном (русском, английском) языке имеются как строчные (малые), так и прописные (большие) буквы. В алфавите символов им приписаны *различные* коды, однако результатом сравнения, например, "а" и "А" должно быть *совпадение*!

Эту проблему будем решать с помощью *таблицы перекодировки*, которая задается как массив целочисленных констант с нумерацией элементов от 0 до 255 (что соответствует кодам символов длиной 1 байт = 8 бит). Если некоторый символ с кодом $d1$ *не требуется* отождествлять ни с каким другим, то элемент массива номер $d1$ должен содержать значение $d1$. Если же символ с кодом $d2$ необходимо отождествить с символом, имеющим код $d3$, то элемент массива номер $d2$ должен содержать значение $d3$. При

обычном сравнении символов $c1$ и $c2$ фактически сравниваются их коды, получаемые функцией Паскаля *ord*: $ord(c1)$ и $ord(c2)$. При использовании таблицы перекодировки T сравниваются $T[ord(c1)]$ и $T[ord(c2)]$.

Для отождествления всех больших и малых букв латинского и русского алфавитов (а также букв "ё", "е", "Ё" и "Е")в коде ASCII (см. приложение 1) таблица перекодировки T должна быть слецующей:

```
const T:array[0..255]of integer =
  ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
   16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
   32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
   48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
   64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
   80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95,
   96, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
   80, 81, 82, 83, 84, 85, 86, 87,
   88, 89, 90,123,124,125,126,127,
  128,129,130,131,132,133,134,135,
  136,137,138,139,140,141,142,143,
  144,145,146,147,148,149,150,151,
  152,153,154,155,156,157,158,159,
  128,129,130,131,132,133,134,135,
  136,137,138,139,140,141,142,143,
  176,177,178,179,180,181,182,183,
  184,185,186,187,188,189,190,191,
  192,193,194,195,196,197,198,199,
  200,201,202,203,204,205,206,207,
  208,209,210,211,212,213,214,215,
  216,217,218,219,220,221,222,223,
  144,145,146,147,148,149,150,151,
  152,153,154,155,156,157,158,159,
  133,133,242,243,244,245,246,247,
  248,249,250,251,252,253,254,255);
```

Эту таблицу будем использовать для сравнения строк в алгоритме 5.2 (функция *scomp*).

Алгоритм 5.2. Лексикографическое сравнение строк $s1$ и $s2$ с отождествлением больших и малых букв с помощью таблицы перекодировки T. Результат равен нулю при $s1 = s2$, единице при $s1 > s2$, минус единице при $s1 < s2$.

```
function scomp(var s1, s2 : string) : integer;
  var i, r, n1, n2;
begin n1 := length(s1); n2 := length(s2);
  r := 0; i := 1;
  while (i ≤ n1)and(i ≤ n2)and (T[ord(s1[i])] = T[ord(s2[i])]) do
    i := i + 1;
  if (i < n1)and(i < n2) then
  begin
    if T[ord(s1[i])] < T[ord(s2[i])] then r := -1
    else r := 1
  end
  else if (i < n1) then r := -1
  else if (i < n2) then r := 1
  scomp := r
end
```

С применением процедуры *scomp* вместо операций сравнения алгоритм дихотомического поиска будет следующим (см. алгоритм 3.6).

Алгоритм 5.3. Дихотомический поиск в упорядоченном массиве строк A из n элементов. Ищется значение $A[i]$, равное строке p. Если такого значения нет, то $i = 0$. При сравнении строк используется функция *scomp* с отождествлением больших и малых букв.

```
b := 1; e := n;
while b < e do
begin c := (b + e)div 2;
  if scomp(A[c], p) < 0 then b := c + 1
  else e := c
end;
if scomp(A[b], p) = 0 then i := b else i := 0;
```

Функцию *scomp* можно применить в других алгоритмах поиска, а также сортировки для массивов строк с прямой или косвенной

упорядоченностью, см. задания 7–11 главы 3.

5.2. Контекстный поиск

Если приходится иметь дело с символьным массивом большого размера, то в Паскале его нельзя представить как строку (тип *string*), его приходится определять как **array of** *char*. Такой массив будем называть просто *текстом*. Задача поиска состоит в том, что в тексте необходимо найти подстроку, совпадающую со строкой-образцом. Такой поиск называется *контекстным*.

Рассмотрим простой алгоритм контекстного поиска. Пусть S — текст длиной n символов, d — строка-образец. Вначале ищется символ текста $S[i]$, равный первому символу образца $d[1]$. При совпадении сравниваются следующие символы: $S[i+1]$ и $d[2]$, $S[i+2]$ и $d[3]$ и т.д. до конца образца. Если для всех пар обнаружится совпадение, то поиск завершится успешно, если нет, то i увеличится на единицу и сравнение будет продолжаться. В общем случае возможны три ситуации:

1) нет ни одного полного совпадения с образцом;

2) есть только одно совпадение с образцом;

3) существует *более одного* совпадения с образцом, при этом возможно даже их перекрытие, как, например в тексте *"мамама"* для образца *"мама"*.

Приведенный ниже алгоритм находит только одно (первое) совпадение, но его нетрудно изменить для поиска *всех* совпадений.

Алгоритм 5.4. Контекстный поиск в тексте S длиной n по образцу-строке d длиной m. Результат: i — номер символа $S[i]$, совпадающего с первым символом образца. Если полного совпадения нет, то $i = 0$.

```
m := length(d); p := 0  i := 1;
while (p = 0)and(i <= n - m + 1) do
  if (d[1] <> S[i]) then i := i + 1
  else begin j:=1;
    while (j < m)and(d[j + 1] = S[i + j]) do j := j + 1;
    if j = m then p := 1 else i := i + 1
  end;
if p = 0 then i := 0;
```

Алгоритм имеет трудоемкость порядка $n \cdot m$. Действительно, так как несовпадение может обнаружиться при сравнении последнего, m-го символа образца, то для перехода от сравнения первого символа образца с i-м символом текста к сравнению с $(i+1)$-м символом текста потребуется от одного до m сравнений отдельных символов.

Иногда контекстный поиск можно выполнить гораздо быстрее. Пусть, например, текст содержит предложения на русском языке и образцом является слово русского языка. Тогда совпадение может начинаться с такой буквы русского языка, которая является самой первой в тексте либо стоит после пробела или знака препинания.

В этом случае в алгоритме поиска можно вначале выделить очередное слово в тексте, а затем сравнить его с образцом.

Алгоритм 5.5. Поиск слова d в тексте S длиной n. Слова разделяются пробелами и другими символами, не являющимися буквами или цифрами. Результат i — начало слова d в тексте S. Если совпадения нет, то $i = 0$.

```
m := length(d);  p := 0  j := 1;
while (p < 2)and(j <= n - m + 1) do
begin c := S[j];
 if p = 0 then
 begin
  if Y[ord(c)] = 1 then
    begin  d1 := c;  p := 1;  i := j  end
 end
 else begin
  if Y[ord(c)] = 1 then d1 := d1 + c
  else begin  if scomp(d, d1) = 0 then p := 2 else p := 0  end
 end;
 j := j + 1
end;
if p = 0 then i := 0
else if (p = 1) and (scomp(d, d1) <> 0) then i := 0;
```

Для определения вида символа используется вспомогательный массив Y с описанием, аналогичным описанию таблицы T. В массиве Y элемент $Y[i] = 1$, если символ с кодом i является

буквой или цифрой, и $Y[i] = 0$, если символ с кодом i *не является* ни буквой, ни цифрой.

Переменная p в алгоритме имеет смысл *состояния* процесса распознавания. Если $p = 0$, то еще не начато выделение очередного слова, т.е. перед этим были обнаружены символы-разделители. Если $p = 1$, то начато выделение слова и его начальные символы помещены в переменную $d1$. Наконец, если полное выделенное слово совпало с образцом, то $p = 2$ и цикл прекращается. Сравнение с образцом осуществляется функцией *scomp*. Дополнительное сравнение с образцом после цикла необходимо из-за того, что искомое слово может находиться в самом конце текста без последующего разделителя.

Алгоритм имеет трудоемкость порядка $2n$. Действительно, основной цикл исполняется не более n раз (точнее, $n - m + 1$ раз). Внутри цикла обычно производится не более двух сравнений, и только в случае полного выделения очередного слова длиной l, на что предварительно потребуется $l+1$ шагов основного цикла, будет выполняться сравнение с образцом, на которое будет затрачено не более l шагов (в процедуре *scomp*). Таким образом, количество шагов выделения очередного слова и его сравнения с образцом не больше, чем удвоенная длина этого слова.

А теперь рассмотрим более сложную задачу: распознавание в тексте слов по группе образцов (по словарю). В этом случае после выделения очередного слова из текста требуется его сравнение со всеми словами-образцами. Если это сравнение производится последовательно, то алгоритм будет иметь трудоемкость порядка $n \cdot (k+1)$, где k — количество слов-образцов. Трудоемкость можно уменьшить до $n \cdot (\lceil \log k \rceil + 2)$, применив дихотомический поиск выделенного слова в *упорядоченном* массиве образцов.

Алгоритм 5.6. Распознавание слов в тексте S длиной n. Слова-образцы записаны в массиве строк D, их количество k. Алгоритм распознает *все* слова-образцы, информация о каждом распознанном слове выводится в виде:

1) распознанное слово;

2) номер слова-образца в массиве D;

3) номер символа в тексте, с которого начинается слово.

```
p := 0  j := 0;
while j <= n do
begin j := j + 1;
 {проверка для последнего слова без разделителя в конце}
 if j <= n then c := S[j] else c :=' ';
 if p = 0 then
 begin
  if Y[ord(c)] = 1 then
  begin
   d1 := c;  p := 1;  i := j
  end
 end
 else begin
  if Y[ord(c)] = 1 then d1 := d1 + c
  else begin       {слово выделено, поиск совпадения}
   r := 0;  l := 1;
   while (r = 0)and(l <= k) do
    if comp(d1, D[l]) = 0 then
    begin r := 1;
     writeln(D[l], l, i)
    end
    else l := l + 1
  end
 end
end;
```

5.3. Редактирование текста

Самые распространенные программы для персональных компьютеров — текстовые редакторы. Они работают с текстом, представляемым обычно в виде массива символов. Основные действия, которые им приходится выполнять, — это ввод символов в текст, вставки, удаления, замены фрагментов текста, форматирование строк и страниц текста. Иногда нужен контекстный поиск (рассмотренный в предыдущем разделе).

Чтобы редактор был практически полезным, в нем необходимо предусмотреть запись сформированного текста в файл, а также чтение из файла ранее созданного текста для его последующего редактирования.

Операции редактирования должны выполняться под контролем пользователя, т.е. в режиме диалога. Алгоритм диалога можно реализовать в виде "бесконечного" цикла, на каждом шаге которого вначале прочитывается с клавиатуры код введенного символа, а затем выполняется одно из действий, в зависимости от того, что это был за символ. Записать алгоритм можно, лишь учитывая специфику компьютера и транслятора Паскаля. Приведенный ниже пример предназначен для Турбо-Паскаля персональных компьютеров. Используемые в алгоритме процедуры и функции входят в библиотеку процедур *Crt*, их смысл пояснен в комментариях. Некоторые из вводимых с клавиатуры специальных символов кодируются двумя следующими друг за другом байтами, при этом первый байт имеет нулевой код. Такие специальные символы вводятся побайтно двумя подряд выполняемыми операторами чтения символа с клавиатуры. В приложении 2 приведены коды специальных символов для компьютеров типа IBM PC.

В алгоритме 5.7 реализован диалог с отображением на экране вводимых символов и запоминанием их в массиве.

Алгоритм 5.7. Диалог с пользователем. Вначале требуется ввести имя файла для записи текста. Затем программа воспринимает, отображает на экране и запоминает в массиве *tx* символы с кодами ASCII от 32 до 255. Запоминается 2000 символов, которые можно одновременно отобразить на экране (25 строчек по 80 символов). Символы управления курсором (стрелки и перевод строки) перемещают курсор на экране, символ "Esc" прекращает цикл, после чего сформированный текст записывается в файл.

```
uses Crt;        {к программе подключается библиотека Crt}
var ch : char;
   x, y, c, i : integer;
     tx :array[1 . . 2000]of char;          {массив для текста}
     ss : string;
     f : text;
begin
  write('введите имя файла: '); readln(ss);
  for i := 1 to 2000 do tx[i] :=' ';
  clrscr;                                   {очистка экрана}
```

```
  window(1,1,80,25); {установка окна 25 строк, 80 столбцов}
  gotoxy(1,1);          {перемещение курсора в начало экрана}
  x := 1;  y := 1;        {запомнен столбец и строка курсора}
  c := 0;
  while c <> 27 do  {цикл прекращается по коду 27 ("Esc")}
  begin ch := readkey;  c := ord(ch);          {чтение символа}
   if c = 0 then     {если спецсимвол, то читается еще байт}
     begin ch := readkey;  c := 300 + ord(ch) end;
   case c of
   13 : {символ перевода строки}
     begin x := 1; if y < 25 then y := y + 1;  gotoxy(x,y) end;
   375 : {символ стрелка влево}
     begin if x > 1 then x := x - 1;  gotoxy(x,y) end;
   377 : {символ стрелка вправо}
     begin if x < 80 then x := x + 1;  gotoxy(x,y) end;
   372 : {символ стрелка вверх}
     begin if y > 1 then y := y - 1;  gotoxy(x,y) end;
   380 : {символ стрелка вниз}
     begin if y < 25 then y := y + 1;  gotoxy(x,y) end;
   32..255 : {изображаемый символ}
     if x < 80 then
       begin tx[(y - 1) * 80 + x] := ch;  x := x + 1;  write(ch) end;
  end;
  assign(f,ss);  rewrite(f);              {запись текста в файл}
  for i := 1 to 2000 do write(f,tx[i]);
  close(f);
end.
```

Алгоритм 5.7 можно усовершенствовать. Во-первых, предусмотреть в нем предварительное чтение текста из файла, если файл с указанным именем уже существует. Во-вторых, включить в алгоритм управление и другими клавишами ("Home", "End" и т.п.). Наконец, можно добавить и другие действия по редактированию: вставку, удаление (символов, строк) и т.п.

Рассмотрим ряд задач редактирования текста. Первая задача – замена фрагмента текста заданной строкой. Длина заменяющей строки может быть больше или меньше заменяемого фрагмента, поэтому текст, следующий после фрагмента, необходимо весь сдвигать вправо или влево.

Алгоритм 5.8. Замена фрагмента текста. В тексте S длиной n фрагмент, начиная с k-го символа длиной l, заменяется на строку d длиной m. Длина массива, где размещен текст, — $n0$.

```
m := length(d);
if n0 - n < m - l then
  writeln('Недостаточно памяти для текста')
else begin
  if l < m then
  begin
    for i := n downto k + l do S[i + m - l] := S[i]
  end
  else if m < l then
  begin
    for i := k + l to n do S[i + m - l] := S[i]
  end;
  for i := 1 to m do S[i + k - 1] := d[i]
end;
```

А теперь рассмотрим задачу построчного вывода текста, в котором слова разделяются пробелами. В тексте после знака препинания обязательно должен быть пробел, сам же знак, следующий после слова, как бы "склеивается" с ним. Сформированные для вывода строчки текста должны быть не больше максимально возможной длины, которая задается перед началом работы алгоритма. При переходе на новую строчку слова разрывать нельзя, в каждую строчку следует записывать максимально возможное число слов.

В алгоритме 5.9 переменные $j0, j1, j2, j3, i$ отмечают фрагмент текста, который будет выводиться в виде очередной строчки:

$j0$ — начало пробелов до первого слова в строчке;

$j1$ — начало первого слова в строчке;

$j2$ — начало пробелов после последнего слова в строчке;

$j3$ — начало слова, следующего после последнего слова в строчке;

i — номер очередного символа в тексте, при выводе строчки — начало пробелов после слова, следующего после последнего слова в строчке.

Переменная p содержит текущее состояние: $p = 0$, если просматриваются разделители, $p = 1$, если просматривается слово.

Алгоритм 5.9. Построчный вывод текста S длиной n. Максимальная длина строчек — m. Разделители слов — пробелы. Вспомогательная процедура $out(a,b)$ выводит текст от символа $S[a]$ до символа $S[b]$ в виде одной строчки.

```
p := 0;  j0 := 1;  j1 := 1;  j2 := 1;  j3 := 1;
for i := 1 to n do
begin c := S[i];
 if c = ' ' then
 begin
  if p = 1 then
  begin
   if i - j1 > m then          {надо вывести строчку}
   begin
    if j1 = j2 then
    begin
     out(j3, i - 1);  j0 := i;  j1 := i;  j2 := i;  j3 := i
    end
    else begin
     out(j1, j2 - 1);  j0 := j2;  j1 := j3;  j2 := i;  j3 := i
    end
   end
   else begin j2 := i;  j3 := i end;
   p := 0
  end
 end
 else if p = 0 then j3 := i
end;
if n - j1 + 1 > m then  {надо вывести еще одну строчку}
begin
 if j1 = j2 then
  begin out(j3, n);  j1 := 0;  j2 := 0; end
 else begin out(j1, j2 - 1);  j1 := j3;  j2 := n + 1: end
end;
if j1 <= n then       {надо вывести последнюю строчку}
 out(j1, n);
```

В ряде алгоритмов (вставка, замена, удаление фрагмента текста) приходится сдвигать значительную часть текста. Хотя трудо-

емкость сдвига имеет порядок n, из-за большой длины текста на него может уйти заметное время, что недопустимо при диалоговом режиме. Значительно ускорить работу можно, организовав текст не в виде сплошного массива символов, а в виде более сложной структуры.

Будем размещать текст, разделенный на строчки, в двумерном массиве символов T, количество строк массива — $u0$, длина строк массива (количество столбцов) — $l0$. В массиве обычно не все строки заняты текстом, текущее количество занятых строк — u. Занятые строки не обязательно располагаются в начале массива, поэтому задается массив номеров строчек K (длиной $n0$) и переменная u, при этом строчки с текстом располагаются последовательно в строках массива: $K[1], \ldots, K[u]$, а незанятые строки в массиве имеют номера: $K[u+1], \ldots, K[u0]$. Текущие длины строчек текста, вообще говоря, различны, и содержатся в массиве L (длиной $n0$). Вначале массив номеров K заполняется числами $1, 2, \cdots, n0$, и переменной u присваивается нуль. Это значит, что текст не содержит ни одной строчки. В процессе редактирования значения элементов массива K могут переставляться, однако в любой момент совокупность их значений является перестановкой всех чисел от 1 до $u0$. А теперь рассмотрим основные действия по редактированию текста.

Последовательное заполнение текста. В цикле переменная u увеличивается на единицу, новая строчка текста записывается в строку массива T с номером $K[u]$, длина строчки — в $L[K[u]]$, и т.д. Если требуется преобразовать текст из представления в виде сплошного массива символов S, то для этого можно использовать алгоритм 5.8 с соответствующей модификацией процедуры *out*.

Удаление строчки текста с номером j. Эта строчка размещается в строке массива T с номером $K[j]$. Вначале запоминается номер K[j], затем номера $K[j+1], \ldots, K[u]$ сдвигаются влево, после чего запомненный ранее номер $K[j]$ переписывается в $K[u]$, а u уменьшается на единицу.

Вставка новой строчки текста после строчки с номером j. Эта строчка записывается в незанятую до того строку массива с номером $K[u+1]$, длина строчки — в $L[K[u+1]]$, номер $K[u+1]$ запоминается, после чего номера строчек $K[j+1], \ldots, K[u]$ сдвигаются вправо. Затем запомненный ранее номер $K[u+1]$ переписывается в $K[j+1]$, а u увеличивается на единицу.

Редактирование одной строчки текста с номером j производится независимо от других строчек. Эта строчка записана в массиве T с номером $K[j]$, ее длина — в $L[K[j]]$. Некоторые проблемы могут возникнуть лишь при вставке фрагмента в строчку, когда новая длина строчки может превысить максимально допустимую $l0$. В этом случае следует предусмотреть, например, вставку новой строчки текста и перенос части редактируемой строчки в новую.

Последовательный просмотр текста предполагает перебор строк массива T в следующем порядке номеров: $K[1], \ldots, K[u]$.

Как видно, многие действия по редактированию будут выполняться гораздо быстрее, так как алгоритм будет перемещать символы не во всем массиве, а только в одной строке либо перемещать только номера строк, но не их содержимое. Но за повышение скорости придется платить увеличением расхода памяти. Пусть, например, максимальная длина строчек $l0$ равна 200, а средняя длина — 60 символов. Тогда 70% памяти в массиве T не будет занято и как-нибудь использовать ее будет невозможно! Кроме того, потребуется память для массива номеров K и для массива длин строчек L, правда, относительно небольшого объема.

5.4. Кодирование

Преобразование текста, изменяющее представление его символов, называют *кодированием*. Кодирование должно быть *обратимым*, т.е. таким, которое допускает обратное преобразование (*декодирование*). Цели кодирования различны. Если кодируют с целью сделать недоступным текст для непосвященных, то его называют *шифрованием* или даже *секретным шифрованием*. Обратное преобразование называют тогда *дешифрованием*.

Рассмотрим несколько простых методов шифрования. Один из них — метод подстановки. Пусть требуется зашифровать текст, в котором встречаются любые из набора 256 символов. Задается таблица шифрования K, похожая на таблицу перекодировки (см. раздел 5.1), но в ней все элементы различны и имеют значения от 0 до 255. Таким образом, таблица шифрования представляет собой одну из возможных перестановок 256 чисел. Шифрование состоит в том, что код исходного символа заменяется на код, содержащийся в элементе таблицы K под номером исходного символа.

Алгоритм 5.10. Шифрование текста S длиной n методом подстановки по целочисленной таблице шифрования K. Зашифрованный текст Q имеет такую же длину n.

```
for i := 0 to n do
  Q[i] := chr(K[ord(S[i])]);
```

Дешифрование осуществляется таким же алгоритмом, только вместо таблицы шифрования K используется таблица дешифрования L. Ее можно построить по таблице шифрования следующим образом. Вначале заполним таблицу L числами $0, 1, \ldots, 255$. Тогда элемент таблицы шифрования $K[i]$ с номером $i = L[i]$ заменяет в зашифрованном тексте значение кода символа $L[i]$ на $K[L[i]]$. Если отсортировать массив K с одновременной перестановкой элементов массива L, то L и будет массивом дешифрования. На самом деле ничего сортировать не нужно: существует гораздо более простой алгоритм.

Алгоритм 5.11. Построение таблицы дешифрования L по таблице шифрования K.

```
for j := 0 to 255 do
  L[K[j]] := j;
```

Таблица шифрования является ключом шифра, ее следует хранить в секрете. Угадать ее трудно: всего возможно 256! вариантов. Однако возникает другая проблема: вручную составить какой-нибудь вариант таблицы нелегко, на это может потребоваться несколько часов работы, а в полученной таблице могут быть ошибки. Поэтому напишем алгоритм генерации таблицы.

Таблица должна быть *случайной*, чтобы посторонний не смог ее воспроизвести. Поэтому для генерации можно использовать датчик случайных чисел — функцию *random*. Эта функция использует глобальную переменную *randseed*, вначале переменной должно быть присвоено некоторое начальное значение. В Турбо-Паскале переменной *randseed* приписан тип длинного целого (длиной 4 байта), поэтому ее значение может быть любым из диапазона от -2147483648 до $+2147483647$. Каждый вызов функции *random* вырабатывает новое значение этой переменной, которое на самом деле не случай-

ное, так как однозначно определяется предыдущим значением. Однако последовательность этих значений *выглядит как случайная* и повторить ее можно, лишь начав генерацию с *того же самого* начального значения *randseed*. Вызов функции — датчика чисел выглядит так: *random*(256). Это значит, что генерируется целое число из диапазона от 0 до 255.

Первое число в таблице может быть любым из заданного диапазона. Однако уже при генерации второго числа есть опасность, что новое число совпадет с ранее полученным. Поэтому в алгоритме необходимо сделать проверку и при необходимости сгенерировать новый экземпляр числа и т.д. Для проверки удобно использовать вспомогательный массив признаков.

Алгоритм 5.12. Генерация таблицы шифрования в виде перестановки чисел $0,1,\ldots,255$ в массиве K. Массив P содержит признаки генерации: если $P[r] = 1$, то число r уже было сгенерировано, а если $P[r] = 0$, то нет. Перед выполнением алгоритма переменной *randseed* должно быть присвоено начальное значение — ключ шифра.

```
for j := 0 to 255 do P[j] := 0;
for j := 0 to 255 do
begin r := random(256);
  while P[r] = 1 do r := random(256);
  P[r] := 1; K[j] := r
end
```

Второй из рассматриваемых методов шифрования — метод перестановок. В нем используются такая же таблица K. Для шифрования исходный текст разбивается на куски длиной по 256 символов, символы в куске нумеруются числами от 0 до 255. Если последний кусок короче 256 символов, то он дополняется, например, пробелами. Каждый кусок текста шифруется отдельно. При шифровании i-й символ в куске записывается на $K[i]$-е место в зашифрованном куске. При дешифровании символ, находящийся на $K[i]$-м месте в зашифрованном куске текста, переписывается на i-е место в исходном куске текста.

Еще один метод шифрования — логическое наложение. Секретный ключ k должен быть ненулевым символьным кодом из восьми

бит, т.е. числом из диапазона от 1 до 255. Каждый из символов текста также рассматривается как 8-битовая последовательность. При шифровании к каждому i-му биту ключа и i-му биту очередного символа применяется логическая операция "исключающее ИЛИ", называемая также сложением по модулю два, ее обозначение: $\oplus$. Шифрование проводится по формуле $d = k \oplus c$, дешифрование — по формуле $c = k \oplus d$. Несложное доказательство правильности формул наложения предоставляется читателю.

В Турбо-Паскале шифрование наложением можно выполнить оператором: $d := char(ord(c)$ **xor** $k)$; При этом переменная k должна иметь тип *byte* (неотрицательное целое длиной 1 байт).

Можно улучшить метод наложения: сделать так, чтобы одинаковые символы исходного текста в зашифрованном виде выглядели по-разному в различных местах текста. Для этого k необходимо использовать как ключ только для первого символа текста, для $(i+1)$-го символа секретным ключом должен быть зашифрованный i-й символ.

С точки зрения *криптографии* рассмотренные методы шифрования не слишком стойкие от вскрытия прежде всего из-за небольшой длины ключа. В методах подстановки и перестановки при генерации таблицы шифрования исходным ключом фактически является начальное значение в датчике случайных чисел (в Турбо-Паскале оно имеет длину 32 бита). В методе наложения длина ключа еще меньше — 8 бит. Достаточно надежным же может быть ключ длиной порядка 100 бит. Для повышения надежности рекомендуется использовать многоступенчатый алгоритм шифрования, когда на каждой ступени применяются различные методы с разными ключами. Тогда полный ключ будет состоять из совокупности этих отдельных коротких ключей.

Другой вид кодирования — сжатие текста, уменьшающее его длину, но допускающее однозначное восстановление. Такое на первый взгляд фантастическое преобразование оказывается возможным почти для любого текста. Использование специальных очень сложных методов позволяет сжимать произвольные тексты в два-три раза и даже более.

Рассмотрим самый простой метод сжатия — метод исключения повторяющихся символов. В этом методе предварительно задают один из кодов символов — h как служебный. Символ с таким ко-

лом должен встречаться в исходном тексте как можно реже. Группа многократно, n раз подряд встречающихся символов с кодом c заменяется группой из трех кодов: (h, n, c). Если в исходном тексте встречается одиночный символ с кодом h, то он заменяется группой из двух кодов: $(h, 0)$.

Алгоритм 5.13. Сжатие текста S длиной n путем исключения повторяющихся символов. Сжатый текст Q имеет длину m. h — служебный символ.

```
c1 := S[1];  k := 1;  m := 0;
for i := 2 to n + 1 do
begin  p := 0;
  if i <= n then c := S[i];
  if (i <= n)and(c = c1) then
  begin
    if k = 255 then p := 1 else k := k + 1
  end
  else if k >= 4 then p := 1
  else if c1 <> h then
  begin  {здесь  k < 4,  символ просто повторяется}
    for j := 1 to k do Q[m + j] := c1;
    c1 := c;  k := 1;  m := m + k;
  end
  else if k = 1 then
  begin  {кодируется одиночный служебный символ }
    Q[m + 1] := h;  Q[m + 2] := chr(0);
    c1 := c;  k := 1;  m := m + 2;
  end
  else p := 1;
  if p = 1 then
  begin          {кодируется повторение символа }
    Q[m + 1] := h;  Q[m + 2] := chr(k);  Q[m + 3] := c1;
    c1 := c;  k := 1;  m := m + 3;
  end
end
```

В этом алгоритме переменная $c1$ содержит значение предыдущего символа текста, ее значение на каждом шаге сравнивается с

текущим символом $c = S[i]$. Переменная k содержит количество повторений одинаковых символов. В цикле выводится не очередной, а предыдущий символ, поэтому дополнительное выполнение цикла при $i = n + 1$ необходимо для того, чтобы вывести последний, n-й символ. Переменная p введена для облегчения проверки условий: при ее значении 1 в выходной текст записывается группа кодов: (h, k, c).

Алгоритм 5.14. Декодирование текста, сжатого алгоритмом 5.13. Сжатый текст Q длиной m декодируется в исходный текст S длиной n.

```
i := 1;  n := 0;
while i <= m do
begin  c := S[i];
 if c <> h then
   begin  n := n + 1;  S[n] := c;  i := i + 1  end
 else begin c1 := Q[i + 1];
   if (i = m)or((i = m - 1)and(ord(c1) <> 0)) then
   begin writeln('ошибка в символе номер ', i);
     i := m + 1
   end
   else if ord(c1) = 0 then
     begin n := n + 1;  S[n] := h;  i := i + 2 end
   else begin
     for j := 1 to ord(c1) do S[n + j] := Q[i + 2];
     n := nord(c1);  i := i + 3;
   end
 end
end
```

Другие методы сжатия, как правило, более сложны и специфичны. Пусть, например, в тексте содержатся неоднократно встречающиеся слова. Выделяя такие слова, можно сформировать из них словарь (заодно подсчитав и количество повторений для каждого слова). После этого для наиболее часто встречающихся слов заменить их в тексте на специально выбранный служебный символ и номер этого слова в словаре. (Чтобы номер можно было записать в один символ, придется ограничиться только 255 словами.) Чтобы такой

текст можно было декодировать, его необходимо сохранять вместе со словарем, поэтому выигрыш будет тем больше, чем чаще встречаются эти слова и чем они длиннее. Реализовать такой алгоритм предоставляется читателю.

5.5. Трансляция арифметических выражений

Под трансляцией понимают перевод текста с одного языка на другой. В информатике обычно имеют в виду перевод текста, записанного на языке программирования (например, на Паскале), в текст на другом языке, обычно на языке машинных команд. Мы рассмотрим простейший вариант трансляции — перевод арифметических выражений со скобками в бесскобочную форму, называемую *обратной польской строкой*. Ее применяют, например, в некоторых микрокалькуляторах.

Пусть арифметические выражения содержат целые числа, круглые скобки и четыре операции: $+, -, /, *$. В обычной формуле для каждой операции должны присутствовать два операнда: левый и правый. В бесскобочной форме это выглядит иначе: вначале записывается 1-й операнд, затем 2-й и только после операндов — операция. Порядок действий здесь получается однозначным и без скобок. Так, например, формула

$$3 * ((122 - 54)/(314 + 28 * 5))$$

транслируется в бесскобочную форму:

$$3 \;\; 122 \;\; 54 \;\; - \;\; 314 \;\; 28 \;\; 5 \;\; * \;\; + \;\; / \;\; *$$

Более удобным будет такой вариант бесскобочной формы, когда записываются не сами числа-операнды, а их номера, числовые значения операндов помещаются в отдельную таблицу. Кроме того, знаки операций заменяются на их числовые обозначения, например 1001 вместо "+", 1002 вместо "−", 1003 вместо "*" и 1004 вместо "/". Для рассмотренного примера это будет выглядеть так:

$$1 \;\; 2 \;\; 3 \;\; 1002 \;\; 4 \;\; 5 \;\; 6 \;\; 1003 \;\; 1001 \;\; 1004 \;\; 1003$$

а таблица будет содержать операнды по порядку их следования в исходной формуле. Тогда сама формула в бесскобочной форме будет записана в целочисленном массиве.

В бесскобочной форме формулу легко вычислить, или, как говорят, *интерпретировать*. Алгоритм интерпретации основан на использовании стека для числовых значений операндов и результатов вычислений. Работа алгоритма состоит в следующем. В цикле просматриваются элементы массива, в котором записана формула. Если очередной элемент является операндом, то его числовое значение из таблицы записывается в стек. Если же элемент массива является операцией, то из стека извлекаются два верхних значения, над ними выполняется эта операция и результат снова записывается в стек. Окончательный результат вычислений по формуле будет единственным значением в стеке.

Заметим, что результат операции деления может оказаться нецелым даже при целочисленных операндах. Поэтому в качестве стека будем использовать массив вещественных, а при пересылке операндов из таблицы в стек производится их преобразование в вещественный тип.

Алгоритм 5.15. Интерпретация формулы в бесскобочной форме, записанной в целочисленном массиве Q длиной m. Числовые значения операндов записаны в массиве T. В качестве стека используется массив R, указатель стека — переменная u.

```
u := 0;
for j := 1 to m do
  if Q[j] < 1000 then
    begin u := u + 1;  R[u] := T[Q[j]] end
  else if Q[j] = 1001 then
    begin R[u - 1] := R[u - 1] + R[u];  u := u - 1 end
  else if Q[j] = 1002 then
    begin R[u - 1] := R[u - 1] - R[u];  u := u - 1 end
  else if Q[j] = 1003 then
    begin R[u - 1] := R[u - 1] * R[u];  u := u - 1 end
  else if Q[j] = 1004 then
    begin R[u - 1] := R[u - 1]/R[u];  u := u - 1 end;
```

А теперь рассмотрим алгоритм трансляции арифметического выражения в бесскобочную форму. Во-первых, операциям припишем приоритеты: будем считать, что приоритет операций "+" и "−"

равен 1, а операций "*" и "/" равен 2. В этом алгоритме тоже используется стек, но совсем по-другому: здесь в него записываются операции и скобки. Алгоритм на каждом шаге просматривает очередной входной символ. При этом до выполнения цикла в стек заносится открывающая скобка, а после окончания просмотра всех символов формулы в качестве дополнительного входного символа используется закрывающая скобка.

Действия алгоритма на каждом шаге определяются видом входного символа. Если это цифра, то формируется числовое значение очередного операнда. Если это скобка или операция, то вначале в таблицу записывается сформированное значение операнда, а в выходной массив записывается его номер из таблицы. Затем в цикле производятся следующие действия со стеком. Если верхний элемент стека — "(", а входной символ — ")", то скобка из стека удаляется. В других случаях либо входной символ (операция или "(") записывается в стек, либо операция (точнее, ее числовое обозначение) из стека переписывается в выходной массив.

Запись в стек производится, если: 1) входной символ — "(", или 2) верхний символ в стеке — "(", или 3) приоритет операции в стеке меньше приоритета операции — входного символа. В остальных случаях идет перепись из стека в выходную строку.

Алгоритм заканчивает работу, когда все символы входной строки просмотрены, а стек пуст. Из-за ошибки в транслируемой формуле может случиться, что стек окажется пустым до конца просмотра либо просмотр закончен, а стек не пуст.

Для проверки типа символа в алгоритме используется целочисленный массив D с нумерацией от 0 до 255. Он организован по принципу таблицы перекодировки (см. раздел 5.1), в нем для кодов символов предусмотрены следующие значения:

1 — для кодов символов "+" и "−";
2 — для кодов символов "*" и "/";
3 — для кода символа "(";
4 — для кода символа ")";
5 — для кодов символов цифр от "0" до "9";
0 — для кодов всех остальных символов.

Алгоритм 5.16. Трансляция формулы, записанной в массиве символов S длиной n в бесскобочную форму. Результат записы-

вается в целочисленный массив Q, его длина — в m. Числовые значения операндов записываются в массив T, их количество — в k. В качестве стека используется массив символов V, указатель стека — переменная u. Переменная p — признак распознавания числа: $p = 0$, если распознается не число, $p = 1$, если были цифры.

```
u := 1; V[1] :=' ('; p := 0; m := 0; k := 0; i := 0;
while i <= n do
begin i := i + 1;
 if i <= n then c := S[i] else c :=')';
 if D[ord(c)] = 0 then Error;          {недопустимый символ}
 else if D[ord(c)] = 5 then begin                {если цифра}
  if p = 0 then
    begin p := 1; x := ord(c) - ord(0) end
  else x := x * 10 + (ord(c) - ord(0))
 end
 else begin                          {если скобка или операция}
  if p > 0 then begin
   p := 0; k := k + 1; T[k] := x; m := m + 1; Q[m] := k
  end;
  b = 0;
  while b = 0 do                              {действия со стеком}
   if (V[u] =' ('and(c =')') then
     begin u := u - 1; b = 1 end
   else if (c =' ('or(V[u] =' ('or(D[ord(V[u])] < D[ord(c)]) then
     begin u := u + 1; V[u] := c; b = 1 end
   else begin m := m + 1;
     if V[u] =' +' then Q[m] := 1001
     else if V[u] =' -' then Q[m] := 1002
     else if V[u] =' *' then Q[m] := 1003
     else if V[u] =' /' then Q[m] := 1004;
     u := u - 1
     if u < 0 then Error;       {ли шняя операция или ")" }
   end
 end
end;
if u > 0 then Error;        {не хватает закрывающих скобок}
```

Используемая в алгоритме процедура *Error* должна выдавать диагностическое сообщение, номер символа во входной строке i, вызвавшего ошибку, а также присваивать переменной i значение $n+1$ для последующего окончания цикла. Желательно также выдавать несколько символов строки до и после ошибки.

Задания для самостоятельной работы

1. Написать какой-либо алгоритм сортировки строк символов с отождествлением прописных и строчных букв, используя процедуру *scomp*.

2. Изменить алгоритм 5.4 для поиска *всех* совпадений с образцом.

3. Модифицировать алгоритмы 5.5 и 5.6 таким образом, чтобы они отыскивали не только совпадения с образцом, но и более длинные слова, начало которых совпадает с образцом.

4. В алгоритм диалога 5.7 включить действия по нажатию клавиш "Delete", "End", "Insert".

5. В алгоритме диалога 5.7 реализовать построчную запись в файл: пробелы в конце строчки не записывать, в конце каждой строчки записывать символы с кодами 13 и 10, как принято в обычных текстовых файлах.

6. В алгоритме, разработанном в задании 5, предусмотреть ввод файла перед редактированием.

7. В алгоритм, разработанный в задании 6, добавить алгоритм построчного вывода текста 5.9.

8. В алгоритме, разработанном в задании 7, предусмотреть выравнивание текста по правому краю, при этом количество пробелов между словами одной строчки не должно отличаться более чем на один.

9. Написать программы шифрования и дешифрования файла каким-либо из рассмотренных методов. Программа должна требовать ввод ключа в виде числа и автоматически формировать необходимую ей таблицу.

10. Написать программу сжатия файла путем формирования словаря и программу дешифрования сжатого файла. В начале сжатого файла должен располагаться словарь, затем сжатый текст. Исходный текст может содержать только слова из букв (строчных) и пробелов.

11. Используя алгоритмы трансляции формул 5.16 и их интерпретации 5.15, написать программу, которая: 1) вводит запись формулы; 2) транслирует ее в бесскобочную форму, 3) вычисляет и выводит ее числовое значение. При обнаружении ошибки в формуле программа должна выдавать сообщение и кусок текста с ошибкой.

12. В алгоритме, разработанном в задании 11, добавить распознавание чисел с десятичной точкой.

Глава 6. ГЕОМЕТРИЧЕСКИЕ АЛГОРИТМЫ

В этом разделе мы будем рассматривать алгоритмы, оперирующие геометрическими объектами, такими, которые можно изобразить на экране компьютера или на листе бумаги. Все современные трансляторы Паскаля включают библиотеки *графических* процедур, которые могут рисовать графические объекты на экране. К сожалению, в различных трансляторах Паскаля графические процедуры, выполняющие одинаковые действия, могут быть совершенно не похожими друг на друга. Поэтому следует руководствоваться литературой по конкретному, используемому вами варианту транслятора Паскаля.

В целом алгоритм вывода графического изображения должен выполнять следующие действия:

1) включение графики;
2) установку графических параметров;
3) вывод отдельных точек, линий, текстовых надписей;
4) временну́ю задержку;
5) выключение графики.

При построении сложных изображений 2-й и 3-й этапы могут многократно повторяться.

6.1. Системы координат, векторы и отрезки

Изображение на экране компьютера — растровое, т.е. представляет собой матрицу из точек (пикселей). Каждая точка экрана задается двумя неотрицательными целыми координатами (числами): xs, ys. При этом $0 \le xs \le A-1$ $0 \le ys \le B-1$, где A, B — количество рядов точек на экране по вертикали и горизонтали соответственно. Нумерация рядов — слева направо и сверху вниз, поэтому *система координат* экрана отличается от общепринятой: нуль находится в левом верхнем углу экрана, ось X направлена слева

направо, а ось Y — сверху вниз (рис. 6.1). Величины A, B определяются типом компьютера и его графическим режимом. Так, например, в компьютерах типа IBM AT, при включении режима VGA, $A = 640$, $B = 480$.

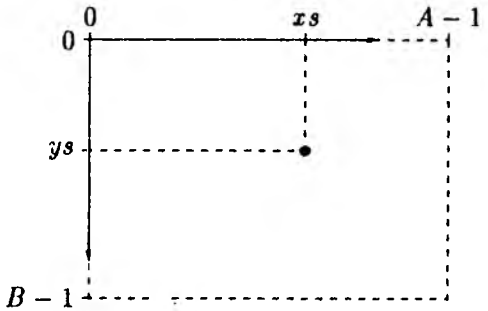


Рис. 6.1

Геометрические алгоритмы имеют дело с точками и отрезками прямых линий. Перед изображением точки ее математические координаты преобразуют с округлением в координаты экрана. Отрезки изображаются в виде ступенчатых линий (рис. 6.2).

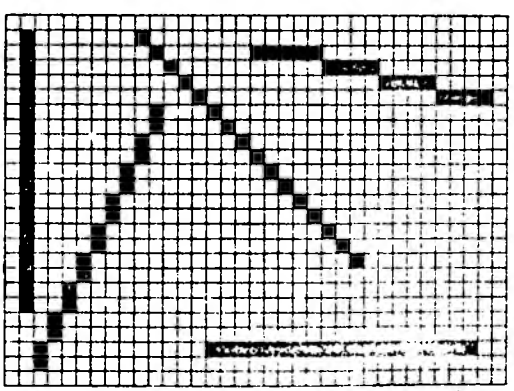


Рис. 6.2

Расчет (интерполяция) и изображение в виде совокупности пикселей для отрезков линий производится процедурами графической библиотеки. Чтобы нарисовать отрезок, необходимо задать его начальную и конечную точки в координатах экрана.

При преобразовании математических координат x, y в координаты экрана xs, ys требуется, во-первых, чтобы для всего изображения хватило места на экране, а во-вторых, чтобы оно занимало возможно бо́льшую часть экрана. Для этого изображение необходимо сдвинуть к началу координат и *масштабировать* — умножить координаты по X и Y на такой коэффициент k, чтобы размеры изображения совпали либо с горизонтальным, либо с вертикальным размером экрана. Обычно изображение занимает либо не весь вертикальный, либо не весь горизонтальный размер. Поэтому его необходимо дополнительно отцентрировать либо по вертикали, либо по горизонтали.

Алгоритм 6.1. Вычисление целочисленных экранных координат xs, ys по вещественным математическим координатам x, y. Заданы вещественные минимальные и максимальные координаты изображения: $xmin$, $xmax$, $ymin$, $ymax$, целочисленные размеры экрана: A, B.

$kx := (A - 0.01)/(xmax - xmin);$
$ky := (B - 0.01)/(ymax - ymin);$
if $kx <= ky$ **then** $k := kx$ **else** $k := ky;$
$xs := (A - trunc((xmax - xmin) * k))$**div** 2 +
$\quad trunc((x - xmin) * k);$
$ys := (B - trunc((ymax - ymin) * k))$**div** 2 +
$\quad B - 1 - trunc((y - ymin) * k);$

В алгоритме вначале вычисляются масштабные коэффициенты отдельно по X и по Y, а затем используется меньший из них. Размеры экрана для вычислений уменьшаются на малую величину (0.01), чтобы при последующем округлении функцией $trunc$ (с отбрасыванием остатка) для максимальных значений математических координат $x = xmax$ и $y = ymax$ получались экранные координаты xs и ys, не превышающие $A - 1$ и $B - 1$ соответственно.

Как известно, каждую точку плоскости можно считать вектором, начало которого находится в точке (0,0). Обозначим координаты точки (вектора) $\vec{a}$ через (a_x, a_y), $\vec{b} = (b_x, b_y)$, $\vec{c} = (c_x, c_y)$. Для векторов определены следующие операции:

1) сумма: $\vec{c} = \vec{a} + \vec{b}$, где $c_x = a_x + b_x, c_y = a_y + b_y$;

2) разность: $\vec{c} = \vec{a} - \vec{b}$, где $c_x = a_x - b_x, c_y = a_y - b_y$;

3) умножение на число k : $\vec{c} = k \cdot \vec{a}$, где $c_x = k \cdot a_x$;

4) скалярное произведение: $\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y$;

5) векторное произведение: $\vec{a} \times \vec{b} = a_x \cdot b_y - a_y \cdot b_x$.

Заметим, что если начальная точка вектора $\vec{a} = (a_x, a_y)$, а конечная $\vec{b} = (b_x, b_y)$, то он является разностью векторов $\vec{b} - \vec{a}$. Отрезок прямой линии в дальнейшем будем задавать как вектор, т.е. отрезок будет *ориентированным.*

Вектор $\vec{a}$ можно задать также в полярной системе координат через его длину (модуль) a и угол α относительно оси OX. Координаты полярной системы координат (a, α) и прямоугольной декартовой (a_x, a_y) связаны соотношениями (рис. 6.3):

$$\begin{cases} a_x = a\cos\alpha, \quad a_y = a\sin\alpha, \\ a = \sqrt{a_x^2 + a_y^2}, \quad \tan\alpha = a_y/a_x. \end{cases} \tag{6.1}$$

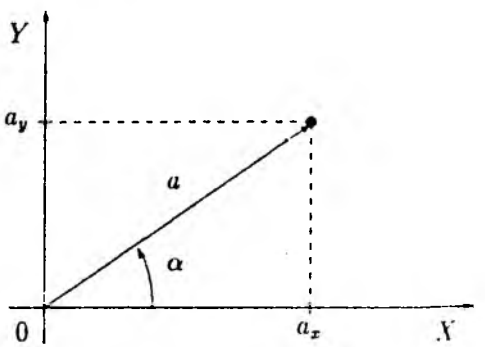


Рис. 6.3

Для скалярного произведения векторов (a, α) и (b, β) справедливо соотношение

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y = \\ a \cdot \cos\alpha \cdot b \cdot \cos\beta + a \cdot \sin\alpha \cdot b \cdot \sin\beta = a \cdot b \cdot \cos(\alpha - \beta), \tag{6.2}$$

а для векторного:

$$\vec{a} \times \vec{b} = a_x \cdot b_y - a_y \cdot b_x = \\ a \cdot \cos\alpha \cdot b \cdot \sin\beta - a \cdot \sin\alpha \cdot b \cdot \cos\beta = a \cdot b \cdot \sin(\beta - \alpha). \tag{6.3}$$

Из этих соотношений следует:

1) скалярное произведение ненулевых векторов равно нулю тогда и только тогда, когда векторы перпендикулярны;

2) векторное произведение ненулевых векторов равно нулю тогда и только тогда, когда векторы параллельны;

3) при общей начальной точке у двух векторов скалярное произведение больше нуля, если угол между векторами острый, и меньше нуля, если угол тупой;

4) при общей начальной точке двух векторов их векторное произведение больше нуля, если второй вектор направлен влево от первого, и меньше нуля, если вправо.

Прямую линию на плоскости можно задать линейным уравнением от двух переменных. Если прямая проходит через две точки (x_1, y_1) и (x_2, y_2), то ее уравнение

$$(x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1). \tag{6.4}$$

Действительно, это уравнение обращается в тождество в точке $(x, y) = (x_1, y_1)$ и в точке $(x, y) = (x_2, y_2)$.

Координаты точек отрезка можно задать двумя параметрическими уравнениями от одной независимой переменной t:

$$\begin{cases} x = x_1 + (x_2 - x_1)t, \\ y = y_1 + (y_2 - y_1)t. \end{cases} \tag{6.5}$$

При $0 \le t \le 1$ точка (x, y) лежит на отрезке, а при $t < 0$ или $t > 1$ — вне отрезка на прямой линии, продолжающей отрезок.

А теперь рассмотрим задачу проверки пересечения двух отрезков. Пусть первый отрезок задан точками $P_1 = (x_1, y_1)$ и $P_2 = (x_2, y_2)$, а второй точками $P_3 = (x_3, y_3)$ и $P_4 = (x_4, y_4)$ (рис 6.4).

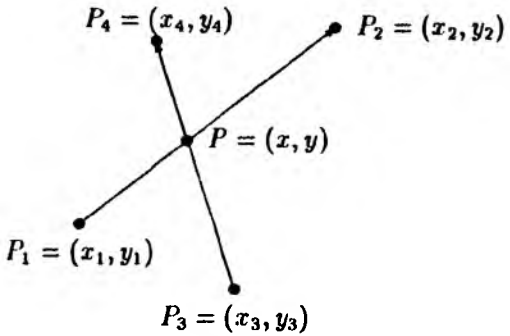


Рис. 6.4

На рисунке точка пересечения отрезков обозначена как $P = (x, y)$. Отрезки пересекаются, если точки P_3 и P_4 расположены по разные стороны относительно вектора $\overrightarrow{P_1P_2}$, а точки P_1 и P_2 — по разные стороны относительно вектора $\overrightarrow{P_3P_4}$. Условие пересечения можно проверить с помощью векторных произведений:

$$\begin{aligned} v_1 &= \overrightarrow{P_3P_4} \times \overrightarrow{P_3P_1}, \quad v_2 = \overrightarrow{P_3P_4} \times \overrightarrow{P_3P_2}, \\ v_3 &= \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_3}, \quad v_4 = \overrightarrow{P_1P_2} \times \overrightarrow{P_1P_4}. \end{aligned} \tag{6.6}$$

Если $v_1v_2 < 0$, $v_3v_4 < 0$, то отрезки пересекаются. Если $v_1v_2 > 0$ или $v_3v_4 > 0$, то отрезки заведомо не пересекаются. Особо следует рассмотреть специальные случаи:

1) если $v_1v_2 \leq 0$, $v_3 = 0$ $v_4 \neq 0$, то точка P_3 расположена на отрезке $\overrightarrow{P_1P_2}$;

2) если $v_1v_2 \leq 0$, $v_4 = 0$ $v_3 \neq 0$, то точка P_4 расположена на отрезке $\overrightarrow{P_1P_2}$;

3) если $v_3v_4 \leq 0$, $v_1 = 0$ $v_2 \neq 0$, то точка P_1 расположена на отрезке $\overrightarrow{P_3P_4}$;

4) если $v_3v_4 \le 0$, $v_2 = 0$ $v_1 \ne 0$, то точка P_2 расположена на отрезке $\overrightarrow{P_3P_4}$;

5) если $v_1 = 0$, $v_2 = 0$, $v_3 = 0$ $v_4 = 0$, то отрезки $\overrightarrow{P_1P_2}$ и $\overrightarrow{P_3P_4}$ расположены на одной прямой, в этом случае требуются дополнительные проверки для выяснения того, перекрываются они или нет.

Для пересекающихся отрезков можно определить точку пересечения. Для этого нужно взять уравнение прямой (6.4), проходящей через точки P_1 и P_2, и аналогичное уравнение прямой, проходящей через точки P_3 и P_4. Решив уравнения как систему относительно неизвестных x и y, получим координаты точки пересечения отрезков.

6.2. Геометрические фигуры. Выпуклость

Мы будем иметь дело с плоскими геометрическими фигурами, причем такими, что их границы являются замкнутыми ломаными линиями. Ломаная задается как последовательность точек: $P_1, P_2, \ldots, P_n$. Подразумевается, что каждая пара соседних точек задает отрезок, т.е. на самом деле ломаная представляет собой последовательность отрезков: $\overrightarrow{P_1P_2}$, $\overrightarrow{P_2P_3}$, $\ldots$, $\overrightarrow{P_{n-1}P_n}$. Для замкнутой ломаной дополнительно задается точка $P_{n+1} = P_1$. Таким образом, под геометрической фигурой будем понимать часть плоскости, ограниченную замкнутой ломаной линией.

Под это определение попадает очень широкий класс фигур, поэтому выделим некоторые частные случаи:

1) треугольник, задаваемый тремя точками: P_1, P_2, P_3;

2) прямоугольник с горизонтальными и вертикальными сторонами, задаваемый двумя точками P_1 и P_2 — соответственно левым нижним и правым верхним углами;

3) фигура с границей, являющейся замкнутой ломаной линией *без самопересечений* (связная фигура);

4) фигура с границей, являющейся *выпуклой* замкнутой ломаной линией (выпуклая фигура).

Выпуклую ломаную можно определить так. Если отрезки в ней просматривать последовательно, передвигаясь по часовой стрелке, то будут получаться только правые повороты.

При работе с фигурами возникает большое число задач: проверка взаимного расположения двух фигур, фигуры и отрезка, фигуры и точки, перестроение фигуры и т.п. Фигуры в алгоритмах будем задавать следующими структурами данных:

1) координаты точек — в виде массивов x и y длиной n;

2) границу фигуры (ломаную линию) — в виде массива T из m номеров точек, являющихся начальными точками m отрезков ломаной; при этом длина массива T должна быть не меньше чем $m + 2$ элемента.

Для начала рассмотрим задачу проверки расположения точки и связной фигуры. Требуется выяснить, расположена заданная точка внутри фигуры или вне ее. Эта задача, на первый взгляд сложная, имеет простое решение. Проведем через точку (x_0, y_0) горизонтальную линию (рис.6.5). Линия пересечет граничные отрезки фигуры несколько раз (причем обязательно четное число раз!). Подсчитаем количество пересечений *слева* от точки. Если оно четно (не забудем, что нуль тоже четное число), то точка лежит *снаружи* фигуры, а если нечетно, то внутри.

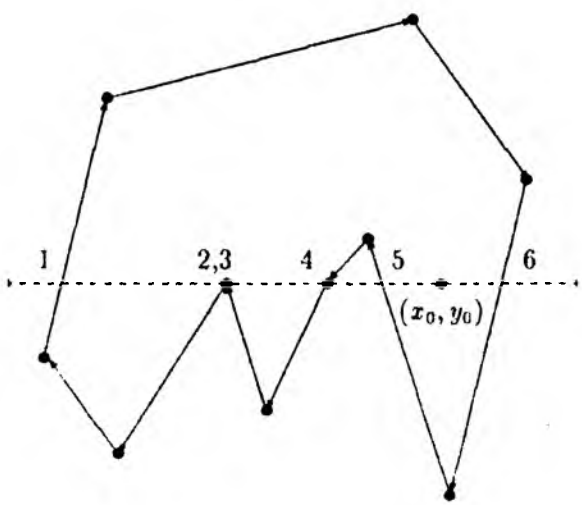


Рис. 6.5

Отдельно рассмотрим особые случаи:

1) точка лежит на i-м отрезке,

2) горизонтальная линия проходит точно через конец i-го отрезка и начало $(i+1)$-го,

3) i-й отрезок лежит на горизонтальной линии.

Для учета второго случая будем считать, что точек пересечения две, если i-й отрезок направлен вверх, а $(i+1)$-й вниз, или i-й отрезок направлен вниз, а $(i+1)$-й вверх. Если же оба отрезка направлены либо вверх, либо вниз, то будем считать, что точка пересечения одна. На рис. 6.5 эти точки обозначены числами $1, \ldots, 6$. Третий случай аналогичен второму, только следует проверять направление $(i-1)$-го и $(i+1)$-го отрезков.

Алгоритм 6.2. Проверка принадлежности точки с координатами $x0, y0$ связной фигуре. Массивы x, y — координаты точек фигуры, массив T — граница фигуры, m — число отрезков границы. Результат: $k = 1$, если точка принадлежит фигуре, $k = 0$, если нет, $k = 2$, если точка лежит на границе фигуры. Процедура *test2* учитывает третий особый случай, а *test3* — второй.

```
T[m + 1] := T[1];
k := 0;  i := 1;
while (k < 2)and(i <= n) do
begin y1 := y[T[i]];  y2 := y[T[i + 1]];
  p1 := (y1 <= y0)and(y0 <= y2);    {отрезок вверх и пересекает}
  p2 := (y1 >= y0)and(y0 >= y2);    {отрезок вниз и пересекает}
  if p1 or p2 then                  {отрезок пересекает линию Y = y0}
  begin x1 := x[T[i]];  x2 := x[T[i + 1]];
    if p1 and p2 then test3                          {учет случая 3}
    else if y0 = y2 then test2                       {учет случая 2}
    else
    begin xp := x1 + (y0 - y1)/(y2 - y1) * (x2 - x1);
      if xp = x0 then k := 2                         {учет случая 1}
      else if xp < x0 then k := 1 - k    {пересечение левее точки}
    end
  end;
  i := i + 1
end
```

Алгоритмы процедур, учитывающих особые случаи *test*2 и *test*3, читателю предлагается написать самостоятельно.

А теперь рассмотрим задачу построения *выпуклой* оболочки связной фигуры. Под выпуклой оболочкой понимается замкнутая выпуклая ломаная минимальной длины, ограничивающая всю фигуру. Для выпуклой фигуры выпуклая оболочка совпадает с ее границей. На рис. 6.6 изображен пример невыпуклой фигуры и ее выпуклой оболочки.

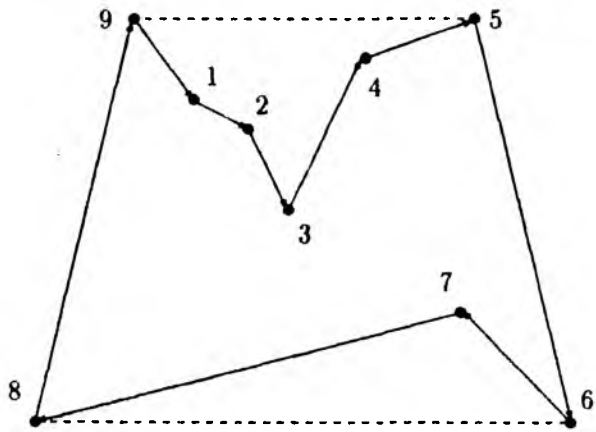


Рис. 6.6

Определим направление обхода по ломаной линии — границе связной фигуры. При переходе от i-го отрезка к $(i+1)$-му направление движения изменяется на некоторый угол по часовой стрелке или против нее. Если считать, что угол по часовой стрелке положительный, а против часовой — отрицательный, то после полного обхода и возврата в начальную точку сумма всех углов будет равна $\pm 360°$. Направление обхода будем считать *правым*, если сумма равна $+360°$, и *левым* в противном случае. Заметим, что для выпуклой фигуры все углы поворота будут иметь одинаковые знаки.

Из этого определения сразу же следует алгоритм построения выпуклой оболочки. Пусть задано правое направление обхода. Просматриваем тройки точек P_i, P_{i+1}, P_{i+2}, начиная от $i = 1$.

Если обнаружим левый поворот от отрезка $P_i\overrightarrow{P_{i+1}}$, к отрезку $P_{i+1}\overrightarrow{P_{i+2}}$, то удаляем точку P_{i+1} и переходим к просмотру точек P_{i-1}, P_i, P_{i+2}. Если поворот правый, то следующими просматриваются точки $P_{i+1}, P_{i+2}, P_{i+3}$. При замыкании обхода после выбора m-й точки требуется выбрать 1-ю, затем 2-ю точку и т.д. Нетрудно доказать, что для правильного замыкания ломаной достаточно обнаружить самую первую тройку точек $P_{m'}, P_{i'}, P_{i'+1}$ с правым поворотом, где штрихами отмечена новая нумерация точек после первого просмотра. Проверить поворот соседних отрезков (векторов!) для тройки точек можно по знаку их векторного произведения – см. формулу (6.3).

Алгоритм 6.3. Построение выпуклой оболочки связной фигуры, заданной ломаной с правым обходом. Массивы x, y — координаты точек фигуры, массив T — граница фигуры, m — число отрезков границы. Выпуклая оболочка формируется в том же массиве T, m — число отрезков оболочки.

```
i := 1;  j := 3;  k := m + 2;
while (i <= k) do
begin T[i + 2] := T[j];
  y1 := y[T[i]];  x1 := x[T[i]];
  x2 := x[T[i + 1]];  y2 := y[T[i + 1]];
  x3 := x[T[i + 2]];  y3 := y[T[i + 2]];
  if (x2 - x1) * (y3 - y2) <= (x3 - x2) * (y2 - y1) then
    begin i := i + 1  j := j + 1 end  {если тройка точек правая}
  else if i > 1 then i := i - 1
  else begin T[i + 1] := T[i + 2];  j := j + 1 end;
  if j > m then                              {замыкание оболочки}
    begin j := 1;  k := i + 1 end;
end;
m := k - j + 3;                              {новое значение m}
for i := 1 to m do T[i] := T[i + j - 1];     {сдвиг в начало T}
```

Рассмотрим пример работы алгоритма для фигуры на рис. 6.6. Ниже по шагам цикла выписаны номера просматриваемых точек в массиве T, проверка выполняется для последних трех точек. Результаты проверки отмечены знаком " + ", если точки образуют

правый поворот, или " – ", если левый. Вертикальная черта отмечает начальную точку оболочки. Для переменных i, j, k указаны те значения, которые они принимают в конце цикла:

1) | 1 2 3 +	$i = 2,$	$j = 4,$ $k = 11$
2) | 1 2 3 4 –	$i = 1,$	$j = 4,$ $k = 11$
3) | 1 2 4 –	$i = 1,$	$j = 5,$ $k = 11$
4) | 1 4 5 –	$i = 1,$	$j = 6,$ $k = 11$
5) | 1 5 6 +	$i = 2,$	$j = 7,$ $k = 11$
6) | 1 5 6 7 +	$i = 3,$	$j = 8,$ $k = 11$
7) | 1 5 6 7 8 –	$i = 2,$	$j = 8,$ $k = 11$
8) | 1 5 6 8 +	$i = 3,$	$j = 9,$ $k = 11$
9) | 1 5 6 8 9 +	$i = 4,$	$j = 1,$ $k = 5$
10) 1 | 5 6 8 9 1 +	$i = 5,$	$j = 2,$ $k = 5$
11) 1 5 | 6 8 9 1 5 –	$i = 4,$	$j = 2,$ $k = 5$
12) 1 5 | 6 8 9 5 +	$i = 5,$	$j = 3,$ $k = 5$
13) 1 5 6 | 8 9 5 6 +	$i = 6,$	$j = 4,$ $k = 5$

Окончательно T: 8 9 5 6

Трудоемкость алгоритма 6.3 линейная, так как он просматривает каждую точку границы выпуклой фигуры не более двух раз.

Следующая задача — построение выпуклой оболочки множества точек. В отличие от предыдущей задачи здесь нет какой-либо исходной ломаной линии, заданы лишь координаты точек. Поэтому, чтобы воспользоваться идеей алгоритма 6.3, вначале построим такую ломаную, которая бы соединяла все точки и не имела самопересечений. Если не требовать от ломаной замкнутости, то построить ее очень легко, достаточно лишь упорядочить все точки по одной из координат. На рис. 6.7 цифрами обозначены точки, упорядоченные по X. Для упрощения задачи вначале будем считать, что нет ни одной пары точек с *совпадающими* координатами по X.

Будем строить оболочку из двух частей — верхней и нижней. При построении верхней части будем просматривать точки от первой к последней, а при построении нижней части — в обратном порядке. При просмотре, так же как в алгоритме 6.3, будем проверять направление поворота для каждой тройки точек. Но так как первая и последняя точки обязательно входят в оболочку, здесь не требуются специальные действия по замыканию оболочки.

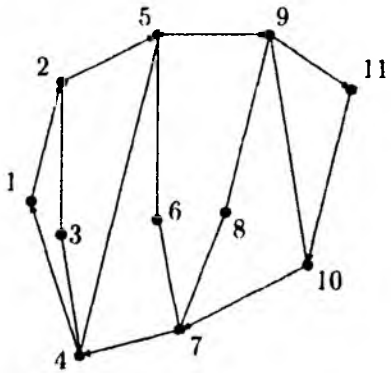


Рис. 6.7

Алгоритм 6.4. Построение верхней части выпуклой оболочки множества точек. Массивы x, y — координаты точек, *упорядоченные по* X, n — их количество. Массив T — номера точек верхней части оболочки, m — их количество.

```
i := 1;  j := 3;  T[1] := 1;  T[2] := 2;
while (j <= n) do
begin T[i + 2] := j;
  y1 := y[T[i]];  x1 := x[T[i]];
  x2 := x[T[i + 1]];  y2 := y[T[i + 1]];
  x3 := x[T[i + 2]];  y3 := y[T[i + 2]];
  if (x2 - x1) * (y3 - y2) <= (x3 - x2) * (y2 - y1) then
    begin i := i + 1  j := j + 1 end   {если тройка точек правая}
  else if i > 1 then i := i - 1
  else begin T[i + 1] := T[i + 2];  j := j + 1 end
end;
m := i + 1;
```

На основе алгоритма 6.4 нетрудно написать алгоритм построения нижней части выпуклой оболочки.

Алгоритм 6.4 обладает линейной трудоемкостью, в нем, так же как и в алгоритме 6.3, каждая точка просматривается не более двух раз. Таким образом, общая трудоемкость решения задачи постро-

ния выпуклой оболочки для произвольного множества точек в основном определяется трудоемкостью алгоритма сортировки и в лучшем случае имеет порядок $n\lceil\log_2 n\rceil$.

6.3. Задача коммивояжера

В этом разделе мы рассмотрим задачу, которая известна как *задача коммивояжера* (бродячего торговца). Вот ее содержательная постановка.

Имеется n городов, между каждой парой городов имеются дороги. Длины всех дорог заданы. Коммивояжер должен разработать такой маршрут своего движения, чтобы, начав путь из какого-либо города, пройти по всем городам, побывав в каждом ровно по одному разу, и вернуться в начало пути. При этом суммарная длина маршрута должна быть минимально возможной. Задача состоит в том, чтобы определить порядок обхода городов.

Задача коммивояжера имеет множество вариантов и интерпретаций. В геометрическом варианте задачи под городами подразумеваются точки на плоскости, под дорогами — отрезки прямых между парами точек. Тогда маршрутом коммивояжера будет замкнутая ломаная минимальной длины, проходящая через все точки.

Насколько сложна эта задача? Во-первых, заметим, что так как маршрут замкнутый, то неважно, с какого города начать движение. Поэтому всегда можно считать, что маршрут начинается с города номер 1. Вторым в маршруте может оказаться любой из оставшихся $n-1$ городов, третьим — один из оставшихся $n-2$ городов и т.д. Таким образом, всего может быть $(n-1)!$ *различных* маршрутов, и обычно только один из них является минимальным. Из этого рассмотрения сразу же следует, что задачу можно решать методом бэктрекинга. За основу возьмем алгоритм 4.4 генерации перестановок чисел.

Алгоритм 6.5. Рекурсивный алгоритм решения задачи коммивояжера. Варианты пути генерируются как перестановки в глобальном массиве P из n элементов. Массив R (тоже из n элементов) содержит признаки включения номеров точек в перестановку: если $R[i]=1$, то номер i включен в перестановку, а если $R[i]=0$, то нет. Рекурсивная процедура $com(k)$ выбирает для перестановок

все оставшиеся точки, начиная с k-й позиции. Вспомогательная функция $dl(i,j)$ вычисляет расстояние между i-й и j-й точками. Вначале всем элементам массива R присваивается нулевое значение, точка номер 1 включается в маршрут: $P[1] := 1$, минимальной величине пути $s0$ присваивается очень большое значение, после чего вызывается $com(2)$. Результатом является путь минимальной длины $P0$ и его длина $s0$.

```
procedure com(k : integer);
  var i, j : integer;
      s : real;
begin
  for i := 2 to n do
    if R[i] = 0 then
    begin R[i] := 1;  P[k] := i;
      if k = n then
      begin s := dl(n, 1);
        for j := 1 to n - 2 do s := s + dl(j, j + 1);
        if s < s0 then begin {новый путь короче s0}
          s0 := s;
          for j := 1 to n do P0[j] := P[j];
        end
      end
      else com(k + 1);
      R[i] := 0
    end
end
```

Этот алгоритм, так же как и алгоритм генерации перестановок, имеет чрезвычайно высокую трудоемкость, так как генерирует *все возможные* варианты маршрутов вне зависимости от того, насколько плох или хорош каждый из вариантов. При улучшении быстродействия алгоритма возникает проблема отсечения перебора, аналогичная той, что решалась в алгоритме 4.5 генерации магических числовых квадратов.

Отсечение перебора в алгоритме 6.5 можно реализовать следующим образом. В алгоритме после проверки $R[i] = 0$ можно предусмотреть вычисление длины s частично построенного маршрута, в который входят точки с номерами: $P[1], P[2], \ldots, P[k], P[1]$. При

этом k-ю точку следует включать в маршрут только при $s < s0$.

Если придумать более точную оценку минимальной длины полного маршрута, который получился бы при достраивании частичного маршрута до конца, то можно добиться отсечения бесперспективных вариантов на более ранних этапах работы алгоритма, когда k невелико. Для этого предварительно для каждой точки вычислим расстояние до ближайшей к ней другой точки и занесем эти ближайшие расстояния во вспомогательный массив. Тогда к длине s частично построенного маршрута необходимо для каждой точки, не вошедшей в маршрут, добавить соответствующее ближайшее расстояние. Написать алгоритм вычисления ближайших расстояний предоставляется читателю.

Однако любые усовершенствования алгоритма не изменят саму природу этой задачи. Доказано, что для нее не существует алгоритма, выполняющегося быстрее чем за экспоненциальное время, т.е. быстрее чем за время порядка c^n, где константа $c > 1$. Рассмотрим следующий пример. Пусть имеется алгоритм, решающий задачу коммивояжера на некотором компьютере за время 4^n микросекунд. Это значит, что при $n = 10$ задача будет решаться за одну секунду ($4^{10} \approx 10^6$). Но уже для $n = 50$ на это потребуется 30 миллионов миллиардов лет! Для справки: возраст Вселенной "всего" около 10 миллиардов лет.

Существует два пути уменьшения времени решения задачи:
1) увеличение мощности компьютера,
2) улучшение алгоритма.

Попробуем первый путь. Будем решать задачу на компьютере, работающем в тысячу раз быстрее. Такие компьютеры уникальны и очень дороги, еще более скоростных компьютеров пока не создали. Тогда задача для $n = 10$ будет решаться за 0.001 секунды, однако для $n = 50$ — за 30 триллионов лет.

Второй путь более обнадеживающий. Пусть для примера алгоритм удалось улучшить настолько, что задача стала выполняться за время 2^n микросекунд. Тогда при $n = 10$ время решения составит 0.001 секунды, а при $n = 50$ — "всего" лишь 30 лет. При выполнении такого алгоритма на компьютере, в тысячу раз более мощном, потребуется тоже немало времени: около 100 дней.

Это значит, что задачу нельзя решить, если n больше нескольких десятков, и никакие наши сверхусилия не спасут положение.

Но что же делать, если задачу надо решать для n, равных, например, 1000 или больше? В практических случаях ничего другого не остается, как вычислить такой путь, длина которого, быть может, не самая минимальная, но значительно меньше, чем у наугад взятого варианта пути. Такое решение называют *приближенным*. Существует много алгоритмов приближенного решения задачи коммивояжера. Рассмотрим один из них, называемый алгоритмом "ближайшего соседа".

Основная идея алгоритма состоит в том, что в качестве очередной, k-й точки маршрута выбирается та точка, которая среди не включенных в маршрут находится ближе всех к $(k-1)$-й точке маршрута.

Алгоритм 6.6. Приближенное решение задачи коммивояжера алгоритмом "ближайшего соседа". Путь формируется в глобальном массиве P из n элементов. Массив R содержит признаки включения номеров точек в путь: если $R[i]=1$, то точка номер i включена в путь, если $R[i]=0$, то нет. Константа max имеет максимально возможное вещественное значение. Функция $dl(a,b)$ вычисляет расстояние между точками с номерами a и b.

```
P[1] := 1;
for i := 2 to n do R[i] := 0;
for i := 2 to n do
begin s0 := max;
  for j := 2 to n do
    if R[j] = 0 then
    begin s := dl(P[i - 1], j);
      if s < s0 then
      begin
        s0 := s; k := j
      end
    end;
  P[i] := k; R[k] := 1
end;
```

Нетрудно видеть, что трудоемкость алгоритма имеет порядок n^2, что позволяет решать задачу для n порядка десятков тысяч. Однако качество решения может оказаться заметно хуже оптимального. На рис. 6.8 приведены примеры оптимального A и приближенного B маршрутов коммивояжера, построенных по одним и тем же точкам.

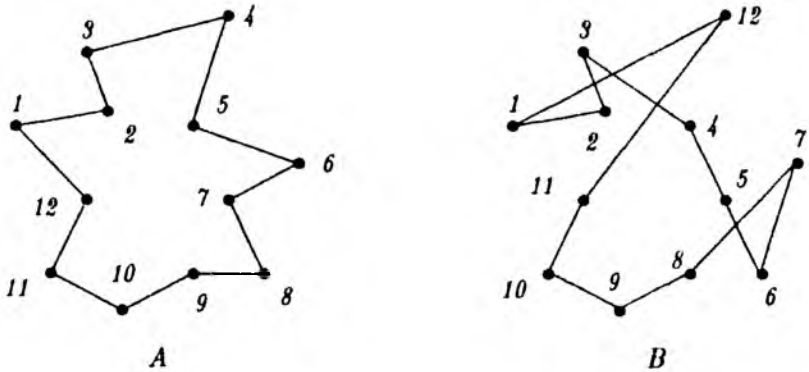


Рис. 6.8

Для алгоритмов, вычисляющих приближенное решение, важно знать, насколько полученное решение *хуже* оптимального. Однако ответ на этот вопрос можно получить, лишь точно зная оптимальное решение. Иногда для него удается вычислить *оценку* в виде неравенства. Получение подобных оценок для различных алгоритмов является интересной и увлекательной задачей, однако выходит далеко за рамки данной книги и здесь не обсуждается.

Задания для самостоятельной работы

1. Вывести формулы для вычисления координат точки пересечения двух отрезков.

2. Написать алгоритм проверки замкнутой ломаной линии является ли она самопересекающейся.

3. Даны два треугольника. Написать алгоритм построения замкнутой ломаной линии, ограничивающей область, общую для обоих треугольников. *Указание*: использовать формулы, полученные в задаче 1.

4. Даны два прямоугольника со сторонами, параллельными осям координат. Написать алгоритм, определяющий их взаимное расположение. Для случая, когда прямоугольники пересекаются, алгоритм должен вычислять:

1) замкнутую ломаную, ограничивающую область, объединяющую оба прямоугольника;

2) замкнутую ломаную, ограничивающую область, общую для обоих прямоугольников.

5. Исследовать работу алгоритма 6.3 для случая, когда в выпуклой оболочке находятся три точки, лежащие на одной прямой. Что изменится, если при проверке тройки точек в алгоритме (правая тройка или нет) нестрогое неравенство заменить на строгое?

6. Исследовать работу алгоритма 6.4 для случая, когда в выпуклой оболочке находятся три точки, лежащие на одной прямой. Что изменится, если при проверке тройки точек в алгоритме (правая тройка или нет) нестрогое неравенство заменить на строгое?

7. Используя алгоритм 6.4, написать алгоритм построения *всей* выпуклой оболочки.

8. В алгоритм 6.5 построения минимального пути коммивояжера добавить сокращение перебора, о котором упоминается в тексте. Экспериментально (на компьютере) проверить, насколько программа стала быстрее работать.

9. Написать программу, которая генерирует точки со случайными координатами, после чего вычисляет минимальный путь коммивояжера алгоритмом из задачи 8, а также приближенный путь алгоритмом 6.6 и выдает длины этих путей. Получить результаты для разных n и для разных наборов случайных точек.

10. Алгоритм "ближайшего города". Похож на алгоритм "ближайшего соседа", но есть отличия:

1) частичный маршрут всегда рассматривается как замкнутый;

2) в качестве очередной точки в маршрут включается та, которая ближе всех других к какой-либо (j-й) точке маршрута, причем включается *между* j-й и $(j+1)$-й точками маршрута.

Написать алгоритм, включить в задачу 9 и экспериментально сравнить его работу с другими алгоритмами вычисления пути коммивояжера.

11. Вывести формулу трудоемкости в зависимости от n для алгоритма задачи 10. Нельзя ли для этого алгоритма добиться трудоемкости порядка n^2 в наихудшем?

ЛИТЕРАТУРА

По языку Паскаль существует необъятное число книг и учебников. Интересны книги автора языка Паскаль Н. Вирта [2, 9]. Для практической работы на компьютере потребуются книги или руководства по конкретному транслятору и операционной системе. Например, для работы с транслятором Turbo Pascal необходима какая-либо из книг по языку и транслятору [7, 8, 15], а также по операционной системе MS DOS или Windows [16].

В данной книге каждая из глав является введением в соответствующий раздел информатики. Ниже приведена литература для более широкого изучения каждого из разделов. Литература, требующая от читателя дополнительной математической подготовки, отмечена звездочками.

К 1-й главе. Об основных идеях доказательства правильности алгоритмов см. [2, 5]. Проблемы тестирования и разработки больших программ изложены в [11]. Развитие теории доказательства правильности см. в [6].

К 2-й главе. О рекуррентных последовательностях и алгоритмах см. в [2]. Алгоритм слияния см. в [3, 10].

К 3-й главе. Об алгоритмах сортировки и поиска см. в [3, 10].

К 4-й главе. О рекурсивных алгоритмах и бэктрекинге см в. [1, 14]. В книге [12] приведены интересные задачи, логические игры, головоломки и обширный список литературы на эту тему.

К 5-й главе. Алгоритмы контекстного поиска есть в книге [3]. Там же можно найти некоторые алгоритмы трансляции. О теории трансляции см. также [4].

К 6-й главе. Алгоритмы вычислительной геометрии см. в [13]. Задача коммивояжера с алгоритмами ее решения подробно изложена в [14].

1. *Абрамов С. А.* Математические построения и программирование. — М.: Наука, 1978. — 192 с.
2. *Вирт Н.* Систематическое программирование. Введение. — М.: Мир, 1977. — 184 с.
3. *Вирт Н.* Алгоритмы + структуры данных = программы. М.: Мир. 1985. 406 с.

*4. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин — М.: Мир, 1975. — 544 с.

5. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. — М.: Мир, 1975. — 245 с.

*6. *Дейкстра Э.* Дисциплина программирования. — М.: Мир, 1978. — 275 с.

7. *Епанешников А. М., Епанешников В. А.* Программирование в среде Turbo Pascal 7.0. — 3-е изд., стер. — М: "ДИАЛОГ-МИФИ", 1995. — 288 с.

8. *Зуев Е. А.* Язык программирования Turbo Pascal 6.0. —М.: Унитех, 1992. —298 с.

9. *Йенсен К., Вирт Н.* Паскаль: Руководство для пользователя и описание языка. — М.: Финансы и статистика, 1982. — 151 с.

*10. *Кнут Д.* Искусство программирования для ЭВМ. **Т. 3.** Сортировка и поиск. — М.: Мир, 1978. — 846 с.

11. *Майерс Г.* Искусство тестирования программ. — М.: Финансы и статистика, 1982. — 175 с.

12. *Поддубная Т. Н., Фукс И. Л.* Информатика в задачах и упражнениях. — Томск: МГП "РАСКО", 1992. — 128 с.

*13. *Препарата Ф., Шеймос М.* Вычислительная геометрия: Введение. — М.: Мир, 1989. — 480 с.

*14 *Рейнгольд Э., Нивергельд Ю., Део Н.* Комбинаторные алгоритмы. Теория и практика. — М.: Мир, 1980. — 476 с.

15. *Фаронов В. В.* Турбо Паскаль. Кн. 1: Основы Турбо Паскаля. — М: МВТУ-ФЕСТО ДИДАКТИК, 1992. — 286 с.

16. *Фигурнов В. Э.* IBM PC для пользователя. Изд. 6-е, перераб. и доп. — М.: "ИНФРА-М", 1995. — 432 с.

Приложение 1

КОДЫ ИЗОБРАЖАЕМЫХ СИМВОЛОВ

В таблице код символа вычисляется как сумма номера строки (от 0 до 15) и числа, которым помечен столбец. Например, код символа "<" равен 12+48 = 60. Коды символов с номерами от 32 до 127 соответствуют стандартным кодам ASCII. Коды символов с номерами от 128 до 241, включающие русские буквы и символы псевдографики, соответствуют так называемой "альтернативной" кодировке, принятой для персональных компьютеров в России.

	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0				0	@	P	‘	p	А	Р	а		└	╨	р	Ё
1			!	1	A	Q	a	q	Б	С	б		┴	╤	с	ё
2			"	2	B	R	b	r	В	Т	в		┬	╥	т	
3			#	3	C	S	c	s	Г	У	г	┃	├	╙	у	
4			$	4	D	T	d	t	Д	Ф	д	┤	—	╘	ф	
5			%	5	E	U	e	u	Е	Х	е	╡	┼	╒	х	
6			&	6	F	V	f	v	Ж	Ц	ж	┤	╞	╓	ц	
7			’	7	G	W	g	w	З	Ч	з	╖	╟	╫	ч	
8			(	8	H	X	h	x	И	Ш	и	╕	╚	╪	ш	
9			)	9	I	Y	i	y	Й	Щ	й	╣	╔	┘	ш	
10			*	:	J	Z	j	z	К	Ъ	к	║	╩	┌	ъ	
11			+	;	K	[	k	{	Л	Ы	л	╗	╦	■	ы	
12			,	<	L	\	l	|	М	Ь	м	╝	╠	▄	ь	
13			-	=	M	]	m	}	Н	Э	н	╜	═	▌	э	
14			.	>	N	^	n	~	О	Ю	о	╛	╬	▐	ю	
15			/	?	O	_	o		П	Я	п	┐	╧	▀	я	

Приложение 2

КОДЫ СПЕЦИАЛЬНЫХ СИМВОЛОВ В IBM PC

Клавиша	Код	+Shift	+Ctrl	+Alt
F1	0+ 59	0+ 84	0+ 94	0+ 104
F2	0+ 60	0+ 85	0+ 95	0+ 105
F3	0+ 61	0+ 86	0+ 96	0+ 106
F4	0+ 62	0+ 87	0+ 97	0+ 107
F5	0+ 63	0+ 88	0+ 98	0+ 108
F6	0+ 64	0+ 89	0+ 99	0+ 109
F7	0+ 65	0+ 90	0+ 100	0+ 110
F8	0+ 66	0+ 91	0+ 101	0+ 111
F9	0+ 67	0+ 92	0+ 102	0+ 112
F10	0+ 68	0+ 93	0+ 103	0+ 113
←	0+ 75		0+ 115	
→	0+ 77		0+ 116	
↑	0+ 72			
↓	0+ 80			
Insert	0+ 82			
Delete	0+ 83			
Home	0+ 71		0+ 119	
End	0+ 79		0+ 117	
PgUp	0+ 73		0+ 132	
PgDn	0+ 81		0+ 118	
BackSpace	8		127	
Tab	9	0+ 15		
Enter	13	13	10	
Esc	27	27	27	
Space	32	32	32	32

Клавиша	+Ctrl	+Alt	Клавиша	+Ctrl	+Alt
A	1	0+ 30	1		0+ 120
B	2	0+ 48	2	0+ 3	0+ 121
C	3	0+ 46	3		0+ 122
D	4	0+ 32	4		0+ 123
E	5	0+ 18	5		0+ 124
F	6	0+ 33	6	30	0+ 125
G	7	0+ 34	7		0+ 126
H	8	0+ 35	8		0+ 127
I	9	0+ 23	0		0+ 129
J	10	0+ 36	-	31	0+ 130
K	11	0+ 37	=		0+ 131
L	12	0+ 38	\	28	
M	13	0+ 50	[	27	
N	14	0+ 49	]	29	
O	15	0+ 24			
P	16	0+ 25			
Q	17	0+ 16			
R	18	0+ 19			
S	19	0+ 31			
T	20	0+ 20			
U	21	0+ 22			
V	22	0+ 47			
W	23	0+ 17			
X	24	0+ 45			
Y	25	0+ 21			
Z	26	0+ 44			

ОГЛАВЛЕНИЕ

Предисловие 3

Глава 1. Разработка алгоритмов и проверка их правильности 5
1.1. Тестирование 6
1.2. Аналитическое доказательство правильности 7
1.3. Доказательство правильности простых алгоритмов 13
1.4. Об эффективности алгоритмов 15
1.5. Разработка сложных алгоритмов и доказательство их правильности 17
Вопросы и задания для самостоятельной работы 20

Глава 2. Однопроходные алгоритмы 21
2.1. Рекуррентные последовательности и однопроходные алгоритмы 22
2.2. Алгоритмы с упорядоченными массивами и множествами 25
2.3. Алгоритмы с файлами 31
Задания для самостоятельной работы 34

Глава 3. Алгоритмы сортировки и поиска 36
3.1. Простые алгоритмы сортировки 36
3.2. Сортировка слиянием 39
3.3. Поиск 42
3.4. Косвенная упорядоченность и поиск в таблицах 45
Задания для самостоятельной работы 51

Глава 4. Рекурсивные алгоритмы и бэктрекинг 53
4.1. Рекуррентные последовательности и рекурсивные алгоритмы 53
4.2. Рекурсивная сортировка слиянием 57
4.3. Бэктрекинг 59
4.4. Логические игры с противником 67
Задания для самостоятельной работы 72

Глава 5. Обработка текста и трансляция **73**
5.1. Сортировка и поиск в массивах строк 73
5.2. Контекстный поиск 77
5.3. Редактирование текста 80
5.4. Кодирование 86
5.5. Трансляция арифметических выражений 92
Задания для самостоятельной работы 96

Глава 6. Геометрические алгоритмы **97**
6.1. Системы координат, векторы и отрезки 97
6.2. Геометрические фигуры. Выпуклость 103
6.3. Задача коммивояжера 110
Задания для самостоятельной работы 114

Литература 116

Приложение 1. Коды изображаемых символов 118
Приложение 2. Коды специальных символов в IBM PC 119

Юрий Леонидович Костюк
Основы алгорнтмизации
Учебное пособие

Редактор Е. В. Лукина

Лицензия 040 – 749 от 18.03.1996 г. Подписано в печать 18.04.1996 г. Формат 60 × 80 1/16. Печать ротапринтная. Бумага офсетная N2. П. л. 7,75, усл. п. л. 7,21; уч.-изд. л. 8,4. Заказ 217. Тираж 1000 экз.

Издательство ТГУ. 634029, г. Томск, ул. Никитина, 4.

Ротапринт ТГУ. 634029, г. Томск, ул. Никитина, 4.