ВИТАЛИЙ ПОТОПАХИН

TURBO PASCAL

РЕШЕНИЕ СЛОЖНЫХ ЗАДАЧ

Комбинаторика

Поиск на графах

Моделирование физических процессов

Рекурсивные и нерекурсивные решения

bhv

Виталий Потопахин

TURBO PASCAL

РЕШЕНИЕ СЛОЖНЫХ ЗАДАЧ

Санкт-Петербург

«БХВ-Петербург»

2006

УДК 681.3.068+800.92Turbo Pascal
ББК 32.973.26-018.1
П64

Потопахин В. В.

П64 Turbo Pascal: решение сложных задач. — СПб.: БХВ-Петербург, 2006. — 208 с.: ил.

ISBN 5-94157-793-1

Книга призвана помочь в овладении искусством программирования тем, кто уже освоил основы составления программ на языке Turbo Pascal. Материал излагается на примере решения 20 практических задач с достаточно сложной логикой по различным темам — комбинаторика, моделирование физических процессов, рекурсивные и нерекурсивные решения. Для каждой задачи анализируются возможный путь к решению, возникающие при этом проблемы, логические ошибки и технические детали. Для большинства задач приведено несколько вариантов решения, для каждого из которых показаны преимущества и недостатки. В процессе анализа выведены некоторые общие правила и принципы программирования.

Для программистов

УДК 681.3.068+800.92Turbo Pascal
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Екатерина Капалыгина*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.01.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 16,77.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-793-1

Оглавление

Введение

Книга написана для тех, кто считает, что программирование — это прежде всего искусство решения логически сложных задач. Главная идея книги — показать процесс мыслительной деятельности таким, какой он есть на самом деле, с ошибками, тупиковыми вариантами, рождением красивой идеи. И что, пожалуй, еще важнее, — показать возможность такой организации своей мыслительной деятельности, при которой поиск решения становится деятельностью системной и планомерной.

Конечно, по прочтению книги у вас не будет рецепта построения решения. Такой рецепт возможен только для трудоемких, но все же логически простых задач, а нас будут интересовать задачи творческие. Единственный метод борьбы с творческими проблемами — это развитый мыслительный аппарат, поэтому 20 глав книги — это описание процесса поиска решения. Именно не описание решения, а описание процесса поиска. Поиск этот не бессистемный. Путь к решению в каждой задаче лежит не только через интуитивные прорывы и даже не столько через них, сколько через логический анализ накопленной информации.

Основной метод, действие которого проявляется на каждом шагу, — это метод борьбы с неопределенностями. Решение каждой задачи представляется как последовательность вопросов "Что нам еще неясно?" и "Как с этим бороться?".

Конечно, трудно ожидать, что первая пришедшая в голову идея будет исчерпывающей, поэтому ряд полученных решений содержат проблемы или даже ошибки. Впрочем, понятие качества решения очень относительно, в силу чего почти в каждой задаче предложено два, а то и три решения и дан небольшой анализ их основных качеств. А какое из них хорошее, это вопрос вкуса и личного стиля.

Возможно, некоторые из использованных задач искушенному "решателю" покажутся простыми, но у нас не было цели максимально осложнить чтение, поэтому задачи разноуровневые и поэтому книга может оказаться полезной для самых разных людей по степени своей подготовки.

Базовый язык книги — язык Паскаль, но все же не стоит воспринимать этот материал как пособие по языку Паскаль. Язык программирования мог бы быть и другим, содержание от этого не претерпело бы серьезных изменений.

Обратите внимание на правила, выведенные в процессе анализа хода решения. Этих правил не много, и они не представляют системы. Создать систему таких правил возможно, но это опасное занятие. В случае удачи подобная система скорее всего привела бы не только к повышению эффективности мыслительного процесса, но и к ограничению ваших творческих возможностей. Поэтому сформулированных правил немного и они не носят характера предписания, как надо мыслить, это скорее всего небольшие советы, корректирующие направление мышления.

Глава 1

Как решается сложная задача

Существует один общий подход к поиску решения сложной задачи, независимо от того, из какой она области: математическая, физическая или программистская. Выражается этот подход в трех простых предложениях:

1. Определим тип задачи.
2. Вспомним, какими методами нам или кому-нибудь другому доводилось решать задачи такого типа.
3. Попробуем применить эти методы к данной задаче.

Этот подход кажется логичным и разумным. Ведь большинство прикладных задач, с которыми мы сталкиваемся, кем-то уже решены, и где-то в базе совокупного знания человечества есть аналоги. Но совокупное знание человечества — это достаточно сложная штука. Оно совсем не так доступно, как хотелось бы, в силу своей огромности. Существуют и другие причины, по которым конкретный человек, сталкиваясь с конкретной задачей, не может или не имеет времени на поиски аналога. Поэтому несмотря на то, что человечество как целое знает и умеет уже довольно много, отдельный человек часто встречается с ситуацией неопределенной и, следовательно, творческой. А в такой ситуации объявленный подход уже не справляется.

Если взять действительно интересную, творческую задачу, то окажется, что определить, к какому типу она относится, довольно сложно. И часто задача будет относиться не к одному типу, а к нескольким. Например, это может быть задача комбинаторного характера с применением графов, или это может быть задача на моделирование физических процессов с использованием графики и методов численной математики.

Вроде бы нет ничего страшного, просто в таких задачах мы имеем дело со сложными типами, и все равно можно действовать по той же схеме, то есть последовательно выполнять действия 1, 2, 3.

К сожалению, не получается. Во-первых, сложных типов можно сконструировать огромное количество, а чем больше типов, тем сложнее будет выполнить пункт первый. Во-вторых, чем больше типов, тем сложнее в них ориентироваться, а в-третьих, тем сложнее отличить, где заканчивается один и начинается другой.

Еще одно разумное предложение звучит так: можно ведь выделить несколько простых типов задач, а сложные конструировать из них.

Эта идея логична тоже только на первый взгляд. Непонятно, что значит "конструировать"? Поясним примером. Пусть имеется два простых типа задач: *Тип первый* и *Тип второй* (не важно, что именно они из себя представляют). Сложный тип можно сконструировать так: *Новый тип* = *Первый тип* и *Второй тип*; а можно и так *Новый тип* = *Первый тип* или *Второй тип*; можно и так *Новый тип* = скорее всего *Первый тип*, но не исключен *Второй*. Из этого простого примера видно, что понятие конструирования очень неопределенное понятие. А если не ясно, как конструировать сложные типы задач из простых, то тем более не ясно, как потом с этим сложным типом сопоставить возможные методы решения.

В общем, как только мы попытаемся составить какую-то классификацию задач, мы столкнемся с таким количество проблем, что невольно придет мысль поискать другой подход.

Попробуем подойти к сложным задачам с другой стороны. Начнем с небольшого, но очень важного замечания. Что бы мы не изобретали, все упирается в рождение идеи. А идея всегда рождается интуитивно, причем независимо от степени ее гениальности и значимости. Происходит это так: вы некоторое время находитесь в состоянии задумчивости, и вдруг вам становится ясно, как решить стоящую проблему, и при этом не ясно, как вы к этой идее пришли. Это называется интуицией.

Все серьезные идеи рождаются интуитивно, и это огорчает, так как совершенно не ясно, каким образом интуицией управлять, но с другой стороны, совершенно не подготовленный человек вряд ли сможет дать красивую идею, и это вселяет надежду. Ведь если для рождения красивой идеи нужна подготовка, значит, интуиция где-то в своей основе содержит систему, какой-то метод, а методу можно научиться.

Что этот метод может из себя представлять? Можно ли вообще описать его точно? Конечно, нельзя ожидать, что общий метод решения творческих задач будет иметь алгоритмическую точность. Невозможно себе представить, что такой метод будет описанием последовательности действий. Конечно, многие люди, когда речь идет о методе, представляют некую последовательность инструкций, выполнив которые мы получим верный результат. Но они глубоко и принципиально неправы.

"Там, где есть асфальт, ничего интересного, а где интересно, там нет асфальта", — братья Стругацкие, "Понедельник начинается в субботу".

Поэтому мы сразу и навсегда откажемся от идеи разработать такой всеобъемлющий алгоритм. Кроме того, мы решительно откажемся от попыток до конца понять тайну творчества. Маловероятно, что интуиция, творческий инсайт, подлежит исчерпывающему логическому анализу. Это то, что мы не будем делать. А попробуем мы разработать метод, системно использующий интуицию, помогающий удержать направление исследования, превратить хаос творчества в осмысленный процесс.

Итак, мы ищем не методы решения задач, а методы организации мыслительной деятельности. Такова наша цель. И чтобы ее достичь, не будем строить теорию, а получим умения из практики. Решая задачи и стараясь отмечать то, что помогло получить решение, как-то обобщать свои наблюдения и, если получится, выводить общие правила.

Для достижения поставленной цели исследуем 20 достаточно сложных задач. Конечно, надо бы исследовать не 20, а 200 задач, но будем надеяться, что и эти 20 принесут свою пользу, а если в конце и не будет исчерпывающего ответа (а его не будет), как решать творческую задачу, то какие-то существенные механизмы мы все же поймем и это будет важный шаг в нужном направлении. Перед тем как перейти к анализу творчества в решении программистских задач, еще раз заметим, что автор этой книги не имеет исчерпывающей или даже просто логически завершенной теории, как работает творческий ум. Более того, автор полагает, что такая теория невозможна по очень глубокой причине — творчество по природе своей бессистемно, и поэтому не может быть описано теорией. Автор только полагает, что существует ряд методов и приемов, позволяющих упорядочить творческий процесс, сделать его более целенаправленным и результативным, так сказать, наложить на творчество логику.

Основное содержание книги — 20 глав, каждая из которых посвящена только одной задаче. Для каждой задачи достаточно подробно показано не только решение, но и возможный путь к нему. Но только *возможный.* Трудно предположить, что два различных человека смогут провести последовательность сложных рассуждений совершенно одинаково.

Там, где было возможно, проведено обобщение и выведено общее правило, сформулирован общий принцип. Общей системы нет, но замеченные методы и принципы обладают различной степенью общности. В этой же главе мы расскажем о нескольких подходах, применимых к процессу решения задач вообще, вне зависимости от их типа.

Пошаговое уточнение

Рассмотрим пример несложной задачи. Пусть требуется разработать алгоритм расчета всех простых чисел, не превосходящих данное число N. Если

для решения не требуется особой эффективности, например верхняя граница не слишком велика (тысячи или десятки тысяч), то можно воспользоваться самой очевидной идеей — алгоритм решения — это цикл, в теле которого для каждого очередного числа выясняется, простое оно или нет, и если простое, то число печатается как результат.

Для того чтобы принять решение относительно простоты текущего числа, необходимо проверить все числа, которые могут быть его делителями, и если хотя бы одно делитель, то проверяемое текущее число не простое, если же ни одного делителя найдено не было, то оно простое.

Идея понятна, но все же вчитаемся в текст более внимательно, все ли здесь действительно понятно. Один неясный пункт есть. Это следующая фраза "надо проверить все числа, которые могут быть его делителями". Можно спросить: "А какие числа могут быть его делителями?" Это и есть уточняющий вопрос. Ответив на него, мы сможем записать идею решения задачи в более точном и развернутом виде.

Еще один пример. Требуется напечатать таблицу умножения. Для того чтобы это сделать, необходимы два множителя, каждый из которых изменяется от 1 до 9. Для каждой пары значений множители перемножаются, и результат выводится в соответствующей позиции экрана монитора.

В этом тексте можно заметить две неопределенности и соответственно задать два вопроса.

1. Что такое соответствующая позиция?
2. Для изменения множителей нужны циклы, множителей два, следовательно циклов также два. Как эти два цикла должны относиться друг к другу? Они должны быть вложены друг в друга или расположены один за другим?

Ответ на эти два вопроса практически даст готовый алгоритм.

Общая схема действий, наверное, ясна. Во-первых, должна быть какая-то идея и, желательно, чтобы идея была верной, уточнение ошибочной идеи приведет в тупик. Затем, записав или как-то иначе выразив то, что уже понятно, мы пытаемся выделить то, что в этой записи не имеет точно определенного смысла. Далее формулируем вопрос, ответ на который должен прояснить смысл, изменяем запись с учетом нового знания и ищем следующую неопределенность.

Есть еще, конечно, сложности с техникой формулировки вопроса. Это хорошо выглядит на простых примерах, таких как были рассмотрены ранее. А для подобных примеров опытному человеку специальные методы анализа и не нужны. Если же взять по-настоящему сложную задачу, то все окажется весьма непростым, и опять на горизонте появится интуиция и творчество. От них, конечно, в этом деле уйти и нельзя, но попробуем дать некоторые наметки.

Если вы внимательно проанализируете свои мыслительные операции, то, наверное, легко сможете заметить два совершенно разных типа деятельности вашего ума. Иногда вы пытаетесь понять, что означает то или иное понятие, а иногда вам требуется определить последовательность операций, выполнение которых приведет к заданному результату. К примеру, в задаче о простых числах мы отвечали только на второй вопрос — как сделать? Это потому, что мы с вами немного знаем математику. Если же за ту же задачу возьмется человек совершенно не искушенный в математике, то первым делом ему придется выяснить для себя: что такое простое число? И что такое делитель числа?

А в более сложных задачах и для искушенного человека может найтись неизвестное понятие. Более того, совершенно не обязательно, чтобы понятие было новым. Проблемы понимания могут быть и со знакомым термином, наш язык многозначен, кроме главного понимания существуют еще оттенки, связанные с контекстом, и может потребоваться более четкое указание, какой смысл данное понятие несет в себе именно в данной задаче.

Приведем пример. Далее в одной из глав рассматривается задача об одиноком путнике. Прочитав условие, любой человек может четко представить себе человека, бредущего в густом тумане по полю с различными препятствиями. Но такое представление не алгоритмизируется. И сразу возникает вопрос, а что такое этот путник? И что такое поле, по которому он движется? Точнее можно спросить так — какими структурами данных можно представить себе поле и одинокого путника?

Из сказанного следует, что процесс уточнения можно разделить на два существенно различных этапа. Во-первых, необходимо выяснить, с чем мы имеем в задаче дело, а лишь затем, как с этим работать. Конечно, такое деление очень условно. Иногда вопросы Что такое? И как делать? возникают попеременно, в самые различные моменты исследования, но все же перед тем как делать, необходимо понять, что скрывается за терминами исследуемой проблемы.

Два подхода к декомпозиции

Предположим, требуется найти сумму факториалов. Допустим, что мы уже знаем, как считать факториалы. Если даже это и не так, то нет ничего страшного, если есть уверенность, что задача счета факториалов разрешима. Итак, пусть мы уже знаем, как их считать. Тогда задача расчета суммы выглядит следующим образом:

```
Сумма = 0
   Для всех чисел от 1 до последнего делать
      Сумма = Сумма + Факториал (от Текущего Числа)
```

Далее мы можем обдумать проблему счета факториалов, уже не думая о том, как считать их сумму. Идея метода этим примером показана исчерпывающе. Вместо того чтобы решать задачу как единое целое, мы предполагаем, что некоторые из подзадач уже решены и оформлены как процедуры или функции, именами которых можно смело пользоваться. Данный метод называется *декомпозицией* задачи на подзадачи. Существенная сложность декомпозиции — это точное определение аргументов функций, решающих отложенные задачи, и точное определение результатов. Существенное преимущество — это решение задачи как единого целого, несмотря на то, что многое еще неизвестно.

Существует еще один подход к декомпозиции, который можно считать общепринятым. Он предполагает разбиение задачи на независимые подзадачи, решение их в отрыве от общей задачи, с последующей компоновкой. В методе, описанном ранее, компоновка выполняется до решения составляющих задач.

С некоторой долей уверенности можно утверждать, что первая форма декомпозиции имеет больше возможностей для применения. Например, она применима в случаях, когда разбиение на независимые задачи затруднительно, главное ее преимущество в постоянном общем видении решения задачи, которое при декомпозиции на независимые задачи может потеряться. А в качестве платы за преимущества, разработчик должен быть способен спрогнозировать некоторые важные свойства будущих функций (аргументы и результат) до их детальной разработки.

Формализация задачи

Формализацию можно рассматривать как составную часть процесса уточнения, в части выяснения смысла используемых понятий. Но так как формализация очень важное понятие само по себе, мы рассмотрим его подробно.

Формализацию задачи можно определить как запись на формальном языке. В отношении задач на программирование *формализация* — это запись задачи в терминах, поддающихся алгоритмизации и последующему программированию на одном из языков программирования. Довольно часто можно встретить мнение, что формализация задачи для последующего программирования — это построение ее математической модели. То есть решение задачи — это выполнение следующих этапов:

1. Построение математической модели.
2. Алгоритмизация.
3. Программирование.

Такая последовательность действий могла бы быть верной, если бы язык математики полностью совпадал с языком программирования, но ни один

язык программирования не совпадает с языком математики. Например, в любом развитом языке программирования есть понятие сложного данного (запись в языке Паскаль, структура в С), в математике этого понятия нет, в математике есть понятие бесконечно малой, в языках программирования этого понятия нет. В задаче о моделировании движения тел в поле сил тяготения мы столкнемся с отсутствием в языках программирования понятий взаимодействия и непрерывности. Вообще надо сказать, что язык математики более богат на понятия, и, следовательно, перевод математической модели на язык программирования может оказаться весьма сложным делом.

Но достаточно часто эта задача вполне разрешима. Некоторые из математических терминов непосредственно соотносятся с терминами, используемыми в языках программирования. Можно уверенно назвать следующие пары схожих понятий:

- Множество — массив;
- Граф — связный список;
- Матрица — двумерный массив.

Можно назвать и другие пары схожих понятий, все множество допустимых пар сильно зависит от личного стиля программиста и конкретной задачи. Например, граф можно представлять в виде матриц инцидентности и смежности, и, следовательно, в языке программирования граф представим двумерным массивом. Но все же есть нечто общее, некий общий принцип. На наш взгляд, этот общий принцип вытекает из различных функциональных возможностей обработки объектов в различных областях знания и обработки структур данных языка программирования. В физике время может изменяться непрерывно, что нельзя реализовать ни с одним типом данных. Сумма ряда может расти неограниченно, для любого числового типа данных существуют определенные ограничения. Сформулируем общий принцип.

==========

Принцип соответствия структур данных объектам

Предположим, в формулировке задачи определяется некоторый объект и для него определен набор операций. Пусть результат выполнения операций нас интересует только с некоторой точностью. Тогда в языке программирования для этого объекта должна быть подобрана структура данных, позволяющая реализовывать аналогичные операции с такой же точностью.

==========

Приведем пример взаимозаменяемых формулировок.

- Дано множество населенных пунктов, связанных между собой дорогами, проезд по которым имеет определенную стоимость.
- Дан неориентированный, взвешенный граф.
- Дан связный список, записи которого содержат числовое поле.

Еще раз обратите внимание, что описанные ранее методы пошагового уточнения и два варианта декомпозиции требуют от разработчика готовой идеи решения. Пусть эта идея будет очень неопределенной и туманной, но она должна быть, именно здесь главная точка приложения интуиции или знания. Результат приложения методов — более точная и более ясная формулировка идеи и возникающих проблем с дальнейшей проработкой решения.

Этап формализации нельзя рассматривать как метод решения. Это необходимый момент любого исследования. Пока мы не представили задачу через структуры, представимые языком программирования, написать реально работающую программу просто невозможно.

Единственно следует заметить, что этап формализации можно проходить в разное время в зависимости от природы задачи. Если поставленная задача обладает высокой степенью логической сложности, то, наверное, более важно проработать логику алгоритма, а представление данных структурами, близкими к языковым, можно и отложить. Задачами такого сорта могут быть задачи поиска на графе, комбинаторные задачи. Эти виды задач традиционно сложны. Очень часто особенной сложностью обладают задачи моделирования каких-либо процессов.

Если же задача носит технический характер, то формализация становится во главу угла. Пример такой задачи — реализация алгоритма деления столбиком двух чисел. Алгоритм общеизвестен, и какого-либо творческого прорыва не нужно, но реализация алгоритма может оказаться достаточно трудоемкой, а в случае неудачного представления данных языковыми структурами даже очень трудоемкой. Еще один пример задачи с техническим характером — это реализация решета Эратосфена. Алгоритм решета также общеизвестен, необходимо только определиться со структурой данных.

Более общих мест в нашей книге не будет. Следующие 20 глав — это детальный анализ конкретных задач и, самое главное, процесса их решения.

Глава 2

Язык записи алгоритмов

Разработка алгоритма — это всегда ответ на два вопроса: из каких действий алгоритм состоит и как это записать. Второй вопрос зачастую не менее важен, чем первый. Наверное многим знакома ситуация, когда понимаешь, что нужно делать, но не можешь ничего объяснить. В деле разработки алгоритма эта проблема серьезно обостряется. Необходимо не просто уметь записать свои мысли, а записать их в соответствии с жесткими правилами написания алгоритма.

Можно без всякого преувеличения сказать, что запись алгоритма — это отдельная наука, требующая специальных навыков, средств и подходов. Не будем перечислять свойства алгоритмов, о которых говорится в школьных учебниках. Добавим к ним только одно важное требование — *запись алгоритма должна быть понятна*. Это требование очень многогранно и многозначно. Например, можно сказать, что запись должна быть наглядна, так как наглядность способствует пониманию. Можно потребовать хорошей структуры алгоритма, так как плохая структура затрудняет понимание. Можно потребовать приближенность к естественному языку. Правда, это требование весьма спорно. Естественный язык отличается нечеткостью и неоднозначностью своих понятий, и, конечно, приближение записи алгоритма к естественной будет одновременно означать потерю однозначности понимания, что плохо.

Вообще если в качестве главного требования к алгоритму взять требование точности и однозначности, то идеальным средством будет язык программирования. Его понятия однозначны, и сам он построен идеально строго. Но тогда говорить о какой-то специальной форме записи алгоритма просто бессмысленно, ведь запись алгоритма и запись программы окажутся практически одним и тем же. Поэтому давайте попробуем разобраться, зачем нужна какая-то специальная форма записи алгоритма, отличающаяся от программы.

Уже было сказано, что общий подход в разработке алгоритмов заключается в постепенном, пошаговом уточнении идеи решения, которая в конечном итоге должна приобрести форму программы. Таким образом, мы утверждаем, что законченный строгий текст должен быть получен только на последнем шаге разработки. Отсюда следует, что алгоритм нам нужен только как промежуточный результат, позволяющий более менее точно сформулировать идею решения, настолько точно, насколько это нужно для перехода к разработке программы.

Сказанное дает нам право пойти на серьезное, с точки зрения теории, нарушение. А именно мы откажемся от строгости в записи в обмен на понятность. В качестве основы для записи алгоритмов возьмем естественный язык (конечно русский), определим набор необходимых конструкций и формы записи этих конструкций.

Простое действие — линейная последовательность команд, смысл которой не нуждается в специальных пояснениях. Имя такой последовательности — это осмысленное выражение или слово, указывающее на смысл простого действия.

Переменная величина — величина, чье значение может изменяться в процессе работы алгоритма. Имя переменной — слово, чей смысл указывает на смысл переменной.

Значимая переменная величина — переменная величина, чье значение велико для работы алгоритма. Имя такой переменной — записанное большими буквами слово, чей смысл указывает на смысл используемой переменной.

Цикл с определенным количеством шагов — конструкция многократного повторения группы операций, состоящая из заголовка и группы выполняемых операций. В заголовке тем или иным образом указывается количество шагов.

Цикл по условию — конструкция многократного повторения группы операций, состоящая из заголовка и группы выполняемых операций. В заголовке указывается либо условие продолжения выполнения группы операций, либо условие прекращения.

Конструкция выбора — описание того, как производится выбор выполняемых действий в зависимости от истинности определенного условия.

Подпрограмма — слово или предложение, указывающие на отдельно записанный алгоритм.

Согласитесь, приведенное описание языка совсем не формальное и каждое из данных определений можно понимать множеством различных способов. Но это не недостаток, это достоинство. Мы просто оставляем большую свободу для формирования личного стиля. В этой книге также есть определен-

ный стиль записи. А для лучшего понимания запишем несколько примеров общеизвестных задач.

Задача 1. Расчет суммы факториалов

```
Сумма = 0
Для всех Целых от 1 до ЧИСЛА
  Сумма = Сумма + Факториал(Целое)
Функция Факториал(Целое)
Произведение=1
Для всех целых от 1 до Целого
  Произведение = Произведение * целое
Вернуть значение Произведения
```

Задача 2. Поиск всех простых чисел, не больших данного целого

```
Для всех Целых от 1 до ЧИСЛА
  Флаг = Истина
  Для всех целых от 2 до Целого -1
    Если Целое делится на целое без остатка то Флаг = Ложь
  Если Флаг есть Истина то Целое есть простое Иначе нет
```

Задача 3. Поиск наибольшего числа

```
Максимальное = Первому
Для всех Чисел от 2 до Последнего
  Если Число больше Максимального То Максимальное = Число
```

Глава 3

Расчет рекуррентной функции

Условие задачи. Вычислить значение функции, заданной следующими условиями:

1. F(1) = 1.
2. F(2N) = F(N).
3. F(2N + 1) = F(N) + F(N + 1).

При этом запрещается использовать массивы в любом виде, в том числе динамические, и запрещается использование рекурсии.

Начнем наши рассуждения. Для начала заметим, что данное определение рекуррентное. Это означает, что для вычисления любого значения функции необходимо определить некоторое количество ее значений от меньших аргументов. То есть мы имеем задачу на вычисление последовательного ряда значений. Самый простой способ вычисления числовых рядов и последовательностей — это моделирование процесса расчетов. Сущность метода моделирования заключается в том, что операторами языка мы описываем процесс расчетов.

Такой метод должен опираться на какие-то особенности ряда. Например, расчет факториала можно описать так:

```
Fact=1;
For I:=2 to N do
  Fact:=Fact*I;
```

Этот фрагмент использует особенность множителей, участвующих в построении факториала, заключающуюся в том, что множители представляют собой последовательность повторяющихся чисел, отличающихся друг от друга на единицу. В нашем случае такой простой закономерности не видно. Можно было бы, конечно, просто запоминать все уже вычисленные значения функции от меньших аргументов, но для этого нужен массив, а массивом

пользоваться запрещено. Кроме того, в момент получения аргумента не известно, какие меньшие аргументы понадобятся в процессе расчетов.

Вторая хорошая идея моделирования опирается на рекуррентный характер функции. Везде, где есть рекуррентные определения, можно строить рекурсивные программы, но рекурсия также запрещена, и поэтому метод моделирования вряд ли применим. Следовательно, нужен иной подход.

Другой подход заключается в поиске хорошей математической закономерности, которая позволила бы упростить проблему расчета. Закономерность можно попытаться найти, анализируя свойства функции, но для этого скорее всего потребуется хороший математический аппарат. А можно просчитать некоторое количество числовых примеров, сопоставить результаты и, может быть, закономерность удастся увидеть. Конечно, закономерность, полученную таким образом, еще придется доказывать (примеры ничего не доказывают), но если закономерность окажется верной для значительного числа примеров, то вероятность ее истинности будет высока. В общем, надо придумать метод счета вручную и просчитать несколько хороших примеров.

Что такое хороший пример? Хороший пример — это серьезная проблема. С одной стороны, хороший пример должен продемонстрировать как можно больше особенностей исследуемого процесса, а следовательно, он должен быть сложным. Но с другой стороны, если пример сложен, то возникает вероятность ошибки, и чем пример сложнее, тем эта вероятность выше. Если же пример прост, то вероятность ошибки невелика или вообще равна нулю, но такой пример может быть очень неинформативен и тогда в нем мало смысла. Сформулируем правило построения хорошего примера.

==========

Правило построения хорошего примера

Для того чтобы получить хороший пример, посмотрим, какие особенности исследуемого вычислительного процесса необходимо продемонстрировать. Сложность и трудоемкость примера должны быть таковы, чтобы исследуемые особенности были представлены полно, но не более того.

==========

Покажем на нашей задаче, как применить полученное ранее правило. Функция определяется тремя правилами, каждое из которых вносит какие-то нюансы в вычислительный процесс. Значит, нужен пример, в котором каждое правило применялось бы хотя бы 2—3 раза. Конечно, больше было бы лучше, но с ростом аргумента будет нарастать трудоемкость вычислений, а значит расти и вероятность ошибки. Далее мы поищем закономерность на одном примере, что, конечно, неправильно. Такое исследование требует большого количества расчетов, но мы все же удовольствуемся одним, чтобы не загромождать текст.

Говоря о правиле расчетов, заметим, что на каждом шагу счета функция может быть представлена либо одной функцией, либо двумя. Это наводит на мысль представить процесс расчетов как дерево уменьшающихся аргументов. Вычислим функцию от 113 (табл. 3.1).

Таблица 3.1. *Расчеты функции от аргумента 113*

<table>
<tr><td colspan="15">113</td></tr>
<tr><td colspan="7">56</td><td colspan="8">57</td></tr>
<tr><td colspan="7">28</td><td colspan="4">28</td><td colspan="4">29</td></tr>
<tr><td colspan="7">14</td><td colspan="4">14</td><td colspan="3">14</td><td colspan="1">15</td></tr>
<tr><td colspan="7">7</td><td colspan="4">7</td><td colspan="3">7</td><td colspan="1">7</td><td colspan="1">8</td></tr>
<tr><td colspan="3">3</td><td colspan="1">4</td><td colspan="3">3</td><td colspan="1">4</td><td colspan="3">3</td><td colspan="1">4</td><td colspan="3">3</td><td colspan="1">4</td><td colspan="1">4</td></tr>
<tr><td>2</td><td>1</td><td>2</td><td>2</td><td>1</td><td>2</td><td>2</td><td>1</td><td>2</td><td>2</td><td>1</td><td>2</td><td>2</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

Результат расчетов F(113) = 13.

В этой таблице уже кое-что видно. Во-первых, видно, что на каждом шагу вычислений мы имеем только два аргумента. Во-вторых, видно, что один из них обязательно четный, а второй обязательно нечетный. Появляется идея. Результат расчетов очевидно равен количеству единичных аргументов. Количество аргументов на каждом шагу — это сумма количеств четных и нечетных аргументов. Следовательно, необходимо поискать закономерность между количествами четных и нечетных. Заметьте, мы смогли увидеть идею только потому, что данные были представлены наглядным образом. Сформулируем правило наглядности.

==========

Правило наглядности

Для того чтобы получить хорошую информацию из результатов численного эксперимента, данные нужно так расположить на бумаге, чтобы взаимосвязи между ними были как можно более наглядны.

==========

Нарушение данного правила может свести на нет вычислительную работу. Например, вычислим значение функции немного иным способом. По определению. Посчитаем значение функции от аргумента 113.

$$F(113) = F(56) + F(57) = F(28) + F(28) + F(29) = 2F(28) + F(29) =$$
$$= 2F(14) + F(14) + F(15) = 3F(14) + F(15) = 3F(7) + F(7) + F(8) =$$
$$= 4F(7) + F(8) = 4F(3) + 4F(4) + F(4) = 4F(3) + 5F(4) =$$
$$= 4F(1) + 4F(2) + 5F(2) = 4F(1) + 9F(2) = 4F(1) + 9F(1) = 13F(1) = 13.$$

Может быть, вам эта запись и не покажется слишком плохой, но согласитесь, она все же намного проигрывает в восприятии в сравнении с таблицей первого метода.

Вернемся к таблице и применим полученное правило еще раз. Распишем количества четных и нечетных на каждом шаге (табл. 3.2).

Таблица 3.2. *Таблица четных и нечетных количеств*

Шаг	Четных	Нечетных
1	1	1
2	2	1
3	3	1
4	1	4
5	5	4
6	9	4
7	0	13

Последний шаг как-то выпадает, здесь появляется ноль, но это последний шаг, он вполне может выпадать из общей схемы расчетов, потому что расчеты уже просто закончены, а вот предыдущие результаты очень любопытны. Обратите внимание, что на каждом шагу, начиная со второго, одно из количеств равно сумме двух верхних, а второе количество равно одному из верхних. Одна из величин равна сумме двух величин предыдущего шага, но иногда равна сумме четных, а иногда сумме нечетных.

Для того чтобы разобраться в том, как именно образуются новые количества четных и нечетных, посмотрим, что происходит на очередном шаге. Пусть на очередном шаге есть два числа ЧЕТНОЕ и НЕЧЕТНОЕ. НЕЧЕТНОЕ всегда распадается на два аргумента: один четный, второй нечетный. Следовательно, нечетное число ничего не изменяет в текущей ситуации. ЧЕТНОЕ уменьшается в два раза. При этом результат деления может стать как четным, так и нечетным. Видимо, здесь и зарыта причина возможных изменений. Рассмотрим эти два варианта развития событий.

- ЧЕТНОЕ опять дает ЧЕТНОЕ. В этом случае общее количество четных увеличивается на количество нечетных, нечетных остается столько, сколько и было. Следовательно, будут верны формулы:
  - Кол-во НЕЧЕТНЫХ = Кол-ву НЕЧЕТНЫХ;
  - Кол-во ЧЕТНЫХ = Кол-во ЧЕТНЫХ + Кол-во НЕЧЕТНЫХ.

- ЧЕТНОЕ дает НЕЧЕТНОЕ. В этом случае общее количество нечетных увеличивается на количество четных, четных становится столько, сколько было нечетных. Следовательно, будут верны формулы:
  - Кол-во НЕЧЕТНЫХ = Кол-во НЕЧЕТНЫХ + Кол-во ЧЕТНЫХ;
  - Кол-во ЧЕТНЫХ = Кол-ву НЕЧЕТНЫХ.

В записанных формулах, конечно же, справа от знака равенства стоят количества текущего (предыдущего) шага, а слева количества следующего (текущего) шага.

Вот в общем-то и все искомые закономерности. Осталось обсудить идею алгоритма и записать сам алгоритм. А идея алгоритма такова: на каждом шаге расчетов мы имеем четыре числа: ЧЕТНОЕ, НЕЧЕТНОЕ и два количества: количество ЧЕТНЫХ и количество НЕЧЕТНЫХ. Шаг работы алгоритма заключается в вычислении по имеющимся четырем числам четырех новых и так до тех пор, пока очередное нечетное число не окажется равно 1. Когда это случится, мы сложим количество четных с количеством нечетных и полученное значение будет искомым ответом.

В своей идее мы исходим из того, что на каждом шаге есть два числа: ЧЕТНОЕ и НЕЧЕТНОЕ. Однако есть один случай, выбивающийся из данного правила. Это первый шаг расчетов, если в качестве исходного аргумента взято четное число. Можно, конечно, попытаться так изменить идею, чтобы этот случай в нее укладывался, но поступим проще. Воспользуемся тем фактом, что в случае с четным аргументом расчет функции не разветвляется. Поэтому не случится ничего страшного, если алгоритм, получив аргумент, будет делить его до тех пор, пока аргумент не станет нечетным, после чего основная идея станет достаточной. Осталось записать алгоритм и текст программы (листинг 3.1).

```
Вводим Аргумент
Пока Аргумент четный делать
  Аргумент = Аргумент / 2
НЕЧЕТНОЕ = Аргумент
Четная сумма =1
Нечетная сумма =1
Пока НЕЧЕТНОЕ больше 1 делать
  Вычислить новое четное и новое нечетное
  Если четное деленное пополам даст четное
    То применить формулы (1)
    Иначе применить формулы (2)
Вывести результат
```

Листинг 3.1

```
program example;
 uses crt;
 var
  c,chet,nechet,sum_chet,sum_nechet:word;
begin
 clrscr;
 readln(nechet);
 while nechet mod 2=0 do nechet:=nechet div 2;
 sum_chet:=1;
 sum_nechet:=1;
 repeat
  chet:=nechet div 2;
  nechet:=nechet - chet;
  if chet mod 2 = 1 then
   begin
    c:=chet;
    chet:=nechet;
    nechet:=c;
   end;
  if ((chet div 2) mod 2)=0 then
   begin
    sum_chet:=sum_chet+sum_nechet;
   end
  else
   begin
    c:=sum_nechet;
    sum_nechet:=sum_nechet+sum_chet;
    sum_chet:=c;
   end;
  writeln(chet,'  ',sum_chet,'  ',nechet,'  ',sum_nechet);
 until nechet=1;
 write(sum_nechet);
end.
```

Решенная задача хорошо иллюстрирует необходимость тщательного математического анализа. В задачах такого типа, а они встречаются очень часто, знание языка не дает почти ничего, чистое же умение алгоритмизации дает немного. Во главу угла здесь встал поиск математической закономерности.

Еще раз повторим, что было существенно важно в поисках закономерности.

- Исходная информация получена из примеров. Логика утверждает, что примеры ничего не доказывают, но примеры могут дать хорошую зацепку

для дальнейшего анализа. В тексте дан только один удачный пример. Конечно, трудно ожидать, что первый попавшийся пример окажется хорошим, поэтому надо быть готовым прорешать их значительное множество.

- Необходимо придумать наглядное представление результатов расчетов, иначе вся масса практической работы может оказаться просто бесполезной.

Между прочим, если разрешить использование рекурсии, то задача становится элементарной и фактически сводится к записи ее рекуррентного определения (листинг 3.2).

Листинг 3.2

```
program example;
 uses crt;
 var
  n:integer;
function Rec(n:integer):integer;
begin
 if n=1 then Rec:=1
  else
   if n mod 2=0 then Rec:=Rec(n div 2)
    else Rec:=Rec(n div 2)+Rec((n div 2)+1);
end;
begin
 clrscr;
 read(n);
 write(Rec(n));
end.
```

В заключение. Конечно, второе решение несравнимо проще, но так бывает не всегда, и необходимо уметь искать математические закономерности. К тому надо заметить, что если нечетных чисел в получаемых разложениях окажется много, то вычислительный процесс начнет стремительно разветвляться, требуя большого объема стековой памяти, и не исключена ситуация, когда ресурсов памяти просто не хватит. Поэтому еще раз повторимся, умение поиска математических закономерностей — это исключительно важное умение.

Глава 4

Перестановки

Условие задачи. Дано произвольное множество символов, построить все перестановки из его элементов.

Из решения предыдущей задачи мы уже знаем, что хорошая идея не появляется на пустом месте. Необходимо четко и ясно представлять себе, как получаются перестановки, и уметь их строить, для чего нужны примеры, много примеров. Но конечно, чтобы не загромождать текст, приведем только один, по возможности удачный.

Таблица 4.1. *Пример перестановок над множеством из 4-х букв*

1	ABCD	7	DABC	13	CDAB	19	BCDA
2	ABDC	8	DACB	14	CDBA	20	BCAD
3	ADBC	9	DCAB	15	CBDA	21	BACD
4	ADCB	10	DCBA	16	CBAD	22	BADC
5	ACDB	11	DBCA	17	CABD	23	BDAC
6	ACBD	12	DBAC	18	CADB	24	BDCA

В табл. 4.1 приведен пример множества перестановок, полученных из четырех элементов, роль которых играют четыре заглавные буквы латинского алфавита A, B, C, D. Этот пример может играть роль определения, из него в принципе видно, что такое перестановки. Для того чтобы закрепить понимание, заметим, что:

- все элементы множества присутствуют в каждой перестановке;
- перемена места двух элементов порождает новую перестановку. Иначе говоря, если в двух перестановках хотя бы в одной позиции стоят разные элементы — это разные перестановки.

Наш пример построен очень удачно. Это, конечно, не случайно, пример специально подобран, что возможно, только если идея уже известна. Получается так, что показан не реальный процесс поиска идеи, а что-то искусственно подобранное. Но это не так. Просто, если демонстрировать все промежуточные шаги и метания, книга может стать безразмерной, а решение каждой задачи — хаотическим нагромождением не вполне понятных действий. Мы многое сокращаем для большей прозрачности. Впрочем, если вы не поленитесь и составите достаточно много примеров перестановок, то в конце концов даже интуитивно вы начнете строить их в каком-то порядке, хотя быть может и не таком, как продемонстрировано в таблице. Способность к упорядочению фундаментальна для нашего интеллекта.

Важное замечание

Вообще очень полезно, перед тем как непосредственно переходить к решению задачи, разобрать в деталях, что означает то или иное понятие. Нарисовать какие-то иллюстрирующие картинки, построить таблицы и диаграммы. Причем это полезно даже в том случае, если вы уверены в своем твердом знании определения. Но твердое знание определения процесса и детальное понимание хода процесса — это не одно и то же. Определение несет в себе минимум информации, а для того чтобы найти хорошее решение, нужен информационный максимум. Грамотная картинка поможет не только более детально разобрать понятие, но и найти хорошую идею решения.

Итак, нам нужна идея. Прочитаем внимательно то, что написано о перестановках. В замеченных свойствах есть одна важная деталь. Сказано, что перемена места элементов порождает новую перестановку. Отсюда возникает мысль — можно построить одну первоначальную перестановку и затем придумать способ построения новой перестановки из уже имеющейся. Этот способ должен быть таков, чтобы:

- пользуясь им, можно было получить все перестановки;
- перестановки не повторялись.

До сих пор наши рассуждения были последовательны, одно вытекало из другого, к сожалению, так не получится до конца, всегда наступает момент, когда приходится положиться на интуицию. Хорошая идея — это всегда интуитивный прорыв, и с этим ничего не поделаешь. Посмотрите внимательно на таблицу, может быть, у вас получится найти идею интуитивно.

Если же вы внимательно посмотрели и у вас не получилось, то давайте вместе. Для начала попробуем придумать вообще любой способ получения новой перестановки из уже имеющейся. Кстати, наверное, это можно записать в виде правила.

==========

Правило упрощения

Если вам необходимо организовать сложный процесс, но пока не получается, попробуйте организовать похожий, но простой. Может быть, позже этот простой удастся усовершенствовать.

==========

А вот простой способ получения перестановок: новая перестановка получается из уже имеющейся перестановкой элементов по кругу: первый на место второго, второй на место третьего, последний на место первого. Для четырех элементов нашего примера это будет выглядеть так: ABCD, DABC, CDAB, BCDA. Все полученные перестановки действительно разные, но если мы попробуем продолжить, то все последующие уже будут повторениями. Но то, что удалось получить несколько разных, вселяет надежду, и можно попробовать метод улучшить. Немного подумаем. Нам не удастся получить новую перестановку, вращая все четыре элемента, но можно вращать три. Возьмем первую перестановку из полученных ранее и прокрутим три последних элемента. Получится вот что: ABCD, ADBC, ACDB. Как видите, три полученные перестановки не повторяются. Если мы повторим точно такую же операцию со всеми четырьмя, то получим 4*3 = 12 различных перестановок.

Дальше проще. Теперь мы возьмем по очереди все 12 перестановок и для каждой из них повращаем уже только две последние буквы. Для примера из первой получим: ABCD, ABDC. Соответственно 12 перестановок превращаются в 24. Вот и все. Теперь можно сказать, что идея решения есть, но выглядит она еще достаточно туманно и, наверное, записать ее в виде алгоритма на данном шаге будет сложно. Поэтому попробуем выразить ее не в виде точного алгоритма, а виде более строгого описания. Помните, в *главе 1* было сказано, что процесс разработки алгоритма можно представить в виде отдельных шагов, на каждом из которых выполняется уточнение уже понятного текста.

Уточнение. Для того чтобы получить все перестановки, берем любую из них за исходную, затем запускаем циклический процесс, в ходе которого на каждом шагу получается новая перестановка прокруткой по кругу элементов текущей. Если в исходной перестановке N элементов, то перед тем как в очередной раз прокрутить N элементов, мы должны N−1 раз прокрутить N−1 элемент. В общем случае, перед тем как в очередной раз прокрутить k элементов, надо k−1 раз прокрутить k−1 элемент. Отчет элементов можно вести слева направо.

Возможно, что-то по-прежнему непонятно, возможно, все кажется понятным, но тем не менее, что-то упущено. Попробуем по нашему описанию отработать еще один пример, попроще. Это нам даст или уверенность, что все ясно, или выявит какую-нибудь скрытую проблему. Возьмем исходное

множество только из трех символов: A, B, C. И построим все возможные перестановки:

1. Исходная перестановка — ABC.
2. Прокрутим два последних и получим новую перестановку — ACB.
3. Мы говорили, что два элемента надо прокрутить два раза. Поэтому выполняем прокрутку еще раз и получаем ABC. Мы действовали по алгоритму и получили повтор. Это плохо, но тем не менее построим все, что получится. Нам нужна информация.

ABC	CAB	BCA
ACB	CBA	BAC
ABC	CAB	BCA

Три перестановки действительно повторились, но похоже, в этих повторениях есть какая-то закономерность. Если мы ее выявим, то сможем лишние перестановки отбрасывать. Конечно, следует заметить, что появление лишних перестановок означает какую-то потерю качества алгоритма (расчет лишних величин — это плохо) и, может быть, следует подумать о другом алгоритме, свободном от этого недостатка, но прежде чем отказываться от идеи лучше проанализировать, так ли это плохо. Может быть, указанным недостатком можно пренебречь.

Примечание

Достаточно часто приходится мириться с недостатком в алгоритме. Машинное время, конечно, важный фактор, но если машина выиграет несколько минут, а вам надо потратить на разработку более эффективного алгоритма несколько часов, то стоит ли его разрабатывать? Кроме того, всегда следует спросить себя, а смогу ли я сделать лучше? И надо ли делать лучше? Что мы выиграем, разработав более эффективный алгоритм? Это вопросы не ленивого человека. Просто представьте себе ситуацию коммерческой разработки. Вам как программисту дали задание. Естественно, кроме содержательной части в этом задании будет указано, за какой срок задание необходимо выполнить. Это в свою очередь означает, что свобода эксперимента самым серьезным образом ограничена.

Конечно, это не означает, что следует выбрать первый попавшийся алгоритм и на нем остановиться. Необходимо искать компромисс между временем разработки и эффективностью алгоритма. Пока у вас мало опыта, нужно меньше жалеть времени, потом с опытом вы сможете за то же время находить более эффективные решения.

А сейчас займемся главной проблемой нашего будущего алгоритма. Предстоит выяснить, когда появляются лишние перестановки, как от них избавиться и по возможности оценить их количество. Анализ последнего примера показывает, что лишние перестановки появляются по одной на каждую прокрутку трех элементов. Причина появления лишней перестановки в том, что мы фактически прокручиваем два элемента три раза.

Видимо, на каждую прокрутку 3 элементов приходится одна лишняя из 2, на каждую прокрутку из 4 — одна лишняя из 3. Далее, прокруток из 3 элементов — 3, следовательно, прокрутки из 3 порождают 3 лишние из 2 элементов. Аналогично 4 прокрутки из 4 порождают 4 лишние из 3 элементов. Таким образом, при N элементах мы имеем N*(N−1)**3 лишних при том, что неповторяющихся перестановок ровно N!. Это означает, что в общей массе перестановок лишних почти столько же, сколько и не повторяющихся. Плохо это или нет, зависит от того, сколько элементов в исходном множестве, на котором строятся перестановки. Надо только заметить, что скорость роста факториала столь велика, а числа факториалов достигают астрономических величин так быстро, что увеличение их еще на треть вряд ли серьезно ухудшает дело — оно и так достаточно плохо. Поэтому можно сделать вывод о целесообразности нашего алгоритма, если количество элементов в исходном множестве не слишком велико.

Все принципиальные моменты построения алгоритма уже обговорены, но для сложных алгоритмов обсуждение только принципиальных вопросов часто бывает недостаточным. Зачастую недоговоренные технические детали создают не меньшие проблемы, чем принципиальные идеи. Поэтому займемся мелочами.

Заметим, что для каждой прокрутки из 4 элементов необходимо построить все перестановки из 3. А для каждой прокрутки из 3 необходимо построить все перестановки из 2 и т. п. То есть можно сказать, что прокрутка из N элементов определяется всеми прокрутками из N−1 элемента. Определения такого вида, когда конструкция или значение определяются значениями или конструкциями той же природы, но отличающимися количественно, называются *рекуррентными*. А если величина определена рекуррентно, то для ее получения можно построить рекурсивную программу. Это очень серьезное утверждение, давайте оформим его в виде правила.

==========

Правило построения рекурсии

Задача, если в ее математическом решении применяются рекуррентные определения, допускает рекурсивное решение, и скорее всего это решение будет наиболее простым и естественным.

==========

Возможно, из того, что было сказано ранее, не совсем понятно, что такое рекуррентное определение. Для пояснения приведем два примера математи-

ческих величин и для каждого два определения: рекуррентное и нерекуррентное (табл. 4.2).

Таблица 4.2. *Примеры рекуррентных и нерекуррентных определений*

Название величины	Нерекуррентное определение	Рекуррентное определение
Факториал	N!=1*2*...*N	N!=N*(N-1)!; 1!=1
Арифметическая прогрессия	$A_N=A_0+d*(N-1)$	$A_N=A_{N-1}+d$; A_0=Число

Если вспомнить, что рекурсивная процедура (или функция) — это такая процедура (или функция), которая в процессе своей работы вызывает сама себя, то становится понятным откуда между рекуррентным определением и рекурсивной процедурой такая хорошая взаимосвязь. У них один принцип работы. Рекуррентное определение не дает значение величины сразу, оно только говорит, какую величину той же природы надо подсчитать, чтобы вычислить данную. Рекурсивная процедура (функция) поступает так же, она не вычисляет величину, она только определяет, с какими значениями запустить процедуру еще раз, чтобы данный вызов мог закончить свою работу. Зачастую, для построения рекурсивного модуля достаточно записать рекуррентное определение. К примеру, тело функции, рекурсивно вычисляющей факториал, будет выглядеть так:

```
If n>1 then factorial:=n*factorial(n-1) else factorial:=1;
```

Обратите внимание на еще одну общую черту. И в рекуррентном определении и в рекурсивном модуле часто необходимо выделить тривиальный случай (то есть простой), на котором расчеты завершаются. А теперь вернемся к нашей задаче, запишем алгоритм и текст программы (листинг 4.1).

Главный алгоритм:

```
Ввести исходное множество — массив длиной N элементов
Вызвать процедуру РЕКУРСИЯ с аргументом N
```

Процедура `РЕКУРСИЯ` — входное значение `N`:

```
Делать N раз
  Построить перестановку на N элементах
  Если N>1 То вызвать процедуру РЕКУРСИЯ с аргументом N-1
```

Листинг 4.1

```
program example;
  uses crt;
  var
```

```
   n_big,m,i:integer;
   a:array[1..10] of char;
   c:char;
procedure printing;
  var
   j:integer;
begin
  m:=m+1;write(m,')  ');
  for j:=1 to n_big do write(a[j],' ');
  writeln;
c:=readkey;
end;
procedure recurs(n:integer);
  var
   i,j:integer;
begin
  for i:=1 to n do
    begin
      c:=a[n];
      for j:=n downto 1 do a[j]:=a[j-1];
      a[1]:=c;
      if i<n then printing;
      if n>1 then recurs(n-1);
    end;
end;
begin
  clrscr;m:=0;readln(n_big);
  for i:=1 to n_big do readln(a[i]);
  clrscr;printing;
  recurs(n_big);
end.
```

Ранее было сказано, что наличие рекуррентного определения означает возможность получения хорошего рекурсивного решения. Это, однако, не означает, что возможно только рекурсивное решение. Для каждой задачи можно придумать как рекурсивное, так и не рекурсивное решение. Решение без рекурсии зачастую более понятно и требует меньше ресурсов. Программирующим на Pascal под DOS, кроме того, необходимо помнить, что рекурсия реализуется посредством стековой памяти, размер которой невелик. А теперь попробуем найти нерекурсивное решение.

Мы построим его на той же идее, что и рекурсивное. В этом нет ничего неожиданного. Рекурсия — это форма организации программы точно так же,

как рекуррентные формулы — это форма организации вычислительного процесса. Поэтому вполне можно ожидать, что и для рекурсии и для обычного решения окажется возможным использовать одну и ту же идею, лишь немного иначе оформленную.

Напомним, что идею кратко можно записать так: алгоритм представляет собой цикл прокруток элементов массива разной длины. Рекурсивное определение перестановки длины N требует построения всех перестановок меньшей длины. Переход к нерекурсивному алгоритму должен это как-то учесть. Заметим, что рекурсивная процедура занимается счетом прокруток. Это делает оператор `if n>1 then recurs(n-1);`. Он вместе с циклом обеспечивает многократное построение меньших перестановок. Отсюда возникает идея счета количества перестановок определенной длины.

Общая идея. Заведем счетчики для перестановок. Счетчик с номером 2 будет считать количество перестановок длины 2, счетчик с номером 3 будет считать перестановки длины 3 и т. д. Тогда ядро алгоритма — это цикл, в теле которого осуществляется поиск счетчика, чье значение меньше его номера. Если, к примеру, будет обнаружено, что счетчик с номером 5 имеет значение 3, то это означает, что есть возможность выполнить прокрутку длины 5.

Проверка значений счетчиков должна выполняться с меньшего номера. Это необходимо для того, чтобы обеспечить условие "Новую перестановку длины N можно строить только тогда, когда построены все перестановки меньшей длины", об этом условии мы уже говорили. Как только построена новая перестановка длины N, опять появляется необходимость в построении всех перестановок меньшей длины, для чего соответствующие счетчики потребуется обнулять.

Счетчик с номером 1 нам не нужен, так как переставлять последний элемент сам с собой нет смысла.

Общая идея сформулирована, важные технические детали описаны, можно записать алгоритм и текст программы (листинг 4.2).

```
Ввести исходное множество
Обнулить счетчики прокруток
Пока значение счетчика с номером N меньше N, делать
  Найти счетчик с наименьшим НОМЕРОМ, такой что его значение
     меньше его НОМЕРА.
  Выполнить прокрутку длиной равной НОМЕРУ
  Увеличить значение найденного счетчика на 1
  Если значение счетчика меньше его НОМЕРА то
    Распечатать перестановку.
    Обнулить все счетчики с меньшими номерами
```

Листинг 4.2

```
program example;
uses crt;
var
 a:array[1..20] of char;
 c:array[1..20] of integer;
 w,n,i,k:integer;
 d:char;
procedure printing;
var
 k:integer;
begin
 w:=w+1;
 write('w=',w,' ');
 for k:=1 to n do write(a[k],' ');
 writeln;
end;
begin
 clrscr;
 readln(n);
 for i:=1 to n do
  begin
   readln(a[i]);
   c[i]:=0;
  end;
 clrscr;
 w:=0;
 printing;
 while c[n]<n do
  begin
   i:=2;
   while c[i]=i do i:=i+1;
   c[i]:=c[i]+1;
   d:=a[1];
   for k:=1 to i-1 do a[k]:=a[k+1];
   a[i]:=d;
   if c[i]<i then
    begin
     printing;
```

```
      for k:=2 to i-1 do c[k]:=0;
    end;
  end;
end.
```

В заключение. Обратите внимание, что при перестройке рекурсивного решения в нерекурсивное пришлось ввести дополнительный массив. Этот массив на самом деле совсем не нов. Он был и в рекурсивном варианте. Стек, используемый рекурсией, и есть дополнительный массив. Разница в том, что за управлением стеком нам следить не приходится, вот почему рекурсивное решение выглядит проще.

Мы дальше еще столкнемся со случаями перестройки рекурсивного решения в нерекурсивное. И всегда организация дополнительного массива, берущего на себя функции стека, будет важным техническим моментом.

Глава 5

Выборки

Условие задачи. Дано множество символов, построить все выборки из данного множества без повторений.

Мы можем немного помочь интуиции анализом условия. Там сказано, что выборки должны быть без повторений. Это означает, что элементы, уже включенные в очередную выборку, необходимо как-то пометить, чтобы ненароком не включить их в эту же выборку еще раз. А это уже конкретное указание на то, что делать. Что же касается пометки, то заметим, что элемент может быть только в двух состояниях: он входит в выборку или не входит. Логично элемент, включенный в выборку, отметить единицей, а не включенный нулем. Для дальнейшего движения зададим себе естественный вопрос: "Что означает новый термин "отметить", куда будет ставиться единица или ноль".

Мы провели хорошую подготовительную работу, теперь выполнить заключительные рассуждения не так сложно. Идея такова: составим двоичное число с длиной, равной длине исходного множества, и тогда если, например, второй разряд двоичного числа равен 1, то второй элемент множества участвует в выборке, а если в некотором разряде находится ноль, то соответствующий элемент множества в выборке не участвует.

Если мы согласны с такой моделью выборки, то сразу появляется и метод получения очередной выборки из уже имеющейся. А именно задача получения очередной выборки сводится к задаче получения очередного двоичного числа из уже имеющегося. Получить же новое двоичное число можно обычной операцией прибавления двоичной единицы.

Приведем пример. Дано:

- исходное множество — Q, W, S, D;
- начальное число — 0001.

Запишем таблицу соответствия чисел и выборок (табл. 5.1).

Таблица 5.1. *Двоичные числа и соответствующие им выборки*

№	Число	Выборка
1	0001	
2	0010	S
3	0011	S D
4	0100	W
5	0101	W D
6	0110	W S
7	0111	W S D
8	1000	Q
9	1001	Q D
10	1010	Q S
11	1011	Q S D
12	1100	Q W
13	1101	Q W D
14	1110	Q W S
15	1111	Q W S D

Алгоритм можно построить как цикл, в начале которого строится число, состоящее из одних нулей, затем на каждом шаге выполнения цикла:

- к уже построенному двоичному числу прибавляется двоичная единица;
- распечатывается очередная выборка, соответствующая полученному двоичному числу.

Построение структур данных не проблема. Исходное множество — это массив, двоичное число — это также массив. Оба массива имеют одинаковую длину.

Если вам уже когда-либо приходилось решать задачу сложения двух двоичных чисел, то сейчас вы можете уже готовую программу сложения оформить в виде процедуры, и часть задачи решена. Это, кстати, очень важное умение использовать готовые результаты. Но мы пойдем другим путем.

Заметим, что у нас нет задачи сложения двух любых двоичных чисел. Перед нами стоит задача прибавить к произвольному двоичному числу двоичную единицу. Это частная задача, а решения частных задач зачастую проще и эффективнее общих.

Действительно, для того чтобы прибавить единицу, достаточно найти первый ноль, начиная с младшего разряда, заменить его на единицу, а затем

все единицы, расположенные правее (или левее в зависимости от того, где находится младший разряд) от только что поставленной, заменить на нули. Если вы желаете наглядно увидеть, как работает это правило, посмотрите внимательно на табл. 5.1, в ней много двоичных чисел, каждое получено из предыдущего прибавлением двоичной единицы, и действие нового правила видно очень хорошо.

Итак, общие идеи мы проговорили достаточно хорошо, теперь запишем алгоритм:

```
Вводим исходное множество, в виде массива.

Формируем массив двоичного числа с длиной, равной количеству элементов
в исходном множестве, и заполняем его нулями.

Пока двоичное число содержит хотя бы один ноль, делаем следующее:

  Прибавляем к двоичному числу двоичную единицу

  Для всех разрядов двоичного числа делаем следующее:

    Если текущий разряд равен единице, то распечатываем элемент

    исходного множества с тем же номером, что и единичный разряд.
```

Несколько замечаний по структуре программы (листинг 5.1).

- Заголовок главного цикла выглядит достаточно сложно. Будет разумно разработать специальную функцию проверки на наличие нулей и вызывать ее в заголовке цикла.
- Задачи прибавления единицы к двоичному числу и распечатки очередной выборки выглядят как достаточно сложные самостоятельные задачи, поэтому разумно их также оформить в виде отдельных процедур, которые будут вызываться поочередно в главном цикле.

Если выполнить эти советы, то главный цикл получится очень компактным и легким для понимания.

Листинг 5.1

```
program example;
 uses crt;
 var
  a:array[1..100] of char;
  flag:array[1..100] of byte;
  i,n:integer;
function proverka:boolean;
var
 i:integer;
 f:boolean;
begin
 f:=false;
```

```
 for i:=1 to n do
  if flag[i]=0 then f:=true;
 proverka:=f;
end;
procedure summa;
var
 i,j:integer;
begin
 i:=1;
 while (flag[i]=1) and (i<=n) do i:=i+1;
 if i<=n then
  begin
   flag[i]:=1;
   for j:=1 to i-1 do flag[j]:=0;
  end;
end;
procedure printing;
var
 i:integer;
begin
 writeln('------------------------------------------------------');
 for i:=1 to n do
  if flag[i]=1 then write(a[i]);
 writeln;
end;
begin
 clrscr;
 readln(n);
 for i:=1 to n do
  begin
   readln(a[i]);
   flag[i]:=0;
  end;
 while proverka do
  begin
   summa;
   printing;
  end;
end.
```

Некоторые важные выводы. Исходная задача является задачей комбинаторной, то есть относящейся к разделу, который считается традиционно сложным

для программирования. Не зря значительное количество задач, предлагаемых на олимпиадах различных уровней, — это задачи именно комбинаторные. Нам удалось перевести задачу совсем в иной раздел, мы переформулировали ее как задачу на сложение двоичных чисел с небольшим довеском в виде исходного множества элементов, обработка которого целиком определяется двоичным числом.

Это говорит о важности разносторонних знаний для человека, занимающегося программированием. В смежных, а иногда и даже отдаленных областях знания можно найти самое неожиданное решение для достаточно сложной задачи. Конечно, надо понимать, что четко и ясно никто не скажет, где может найтись такое простое и неожиданное решение, на то оно и неожиданное, поэтому самой верной стратегией подготовки будет широкая любознательность, изучение самых различных областей знания, но в первую очередь необходимо уделить внимание разделам математики.

Применив свои математические знания, мы действительно нашли в соседней области хорошее и даже очень хорошее решение задачи, но чтобы действительно понять, насколько оно хорошо, его надо с чем-то сравнить. Поэтому сейчас решим нашу задачу еще раз, при этом постараемся, чтобы второе решение также было хорошим, а затем сравним их.

Второе решение. Заметим, что каждый элемент исходного множества может либо входить в выборку, либо не входить. Если для выборки завести отдельный массив, то для каждого элемента массива выборки возможно только два состояния: либо этот элемент пуст, либо он содержит соответствующий элемент исходного множества.

Для первого элемента массива выборки существуют два состояния, для каждого состояния первого элемента возможны два состояния второго элемента и т. д. Введем понятие последовательности. *Последовательностью* длины K будем называть часть массива выборки длины K, начиная с первого элемента. *Множество последовательностей* длины K мы можем определить через множество последовательностей длины K−1 следующим образом: каждая последовательность множества длины K — это последовательность из множества длины K−1, к которой в позицию K добавлен либо пробел, либо K-ый элемент исходного множества.

Тогда множество выборок — это множество последовательностей длины N. Полученное нами определение имеет рекуррентный вид, из этого следует, что мы можем получить рекурсивное решение. Строиться рекурсивное решение на следующих почти очевидных соображениях:

- цель искомой рекурсивной процедуры — за N вызовов построить очередную выборку, следовательно, на N-ом вызове можно завершать построение очередной выборки и выполнять распечатку;

- каждый вызов удлиняет уже построенную выборку на одну позицию, следовательно, в каждом вызове должен определяться очередной элемент выборки;
- следующий элемент выборки можно определить двумя способами (пустой или элемент исходного множества), следовательно, каждый вызов процедуры должен порождать еще два вызова.

Приведем код второго решения задачи в листинге 5.2.

Листинг 5.2

```
program example;
 uses crt;
 var
  a,b:array[1..10] of char;
  n,i:integer;
procedure vb(t,k:integer);
procedure print;var
 i:integer;
begin
 if k+1=n then
  begin
   for i:=1 to n do
    if b[i]<>' ' then write(b[i],' ');
   writeln;
  end;
end;
begin
 if t=0 then b[k]:=' ' else b[k]:=a[k];
 if k<n then
  begin
   vb(0,k+1);
   print;
   vb(1,k+1);
   print;
  end;
end;
begin
 clrscr;
 readln(n);
```

```
 for i:=1 to n do readln(a[i]);
 vb(0,1);
 vb(1,1);
end.
```

В заключение. Нетрудно заметить, что второе решение даже изящнее и короче первого. Это решение, хотя оно и рекурсивное, значительной нагрузки на стековую память не накладывает, так как оба используемых массива являются глобальными, локальных переменных практически нет. Но логика второго алгоритма существенно сложнее для понимания, и с этой точки зрения первый алгоритм более предпочтителен, если есть потребность в быстром получении решения.

На базе второго решения вполне возможно построить третье, нерекурсивное решение, но оно не даст ничего нового, поэтому воздержимся.

Глава 6

Шахматная доска

Условие задачи. Нарисовать шахматную доску, используя минимум графических возможностей.

Задача рисования шахматной доски не особенно сложна. Изюминка нашей формулировки в том, что нужно обойтись как можно меньшим количеством возможностей. Надо сказать, что умение пользоваться минимумом — это очень важное умение. Если вы можете многое с небольшим количеством инструментов, то, добавляя к своему искусству новый инструмент, вы резко увеличиваете свои возможности. Если же вы привыкли пользоваться мощными и разнообразными инструментами, то мощь нового, скорее всего, останется для вас скрытой. Итак, ограничим себя одной графической процедурой — процедурой установки точки.

Сейчас попробуем показать, что этого вполне достаточно, но изящество решения здесь не главное. Попутно мы решим еще одну проблему, а именно продемонстрируем метод решения задач, который назовем *методом пошагового усложнения.*

Представьте себе, что данную задачу решает человек, обладающий хорошим логическим мышлением, но он не опытный программист, и наша задача для него кажется технически очень сложной. А так как решить задачу все же надо, он вырабатывает для себя следующую стратегию поведения: первым шагом он упрощает исходную задачу и доходит до такой, которая выглядит элементарной. Затем решается элементарная задача, после чего начинается обратный процесс усложнения полученного решения.

Начнем рассуждать с позиции такого человека.

Анализ и решение примитивной задачи. Шахматная доска состоит из колонок (или полосок) квадратов. Если бы удалось построить одну полоску, то,

повторив ее многократно, удалось бы получить и шахматную доску. Полоска состоит из квадратов, если бы удалось построить один квадрат, то, многократно его повторив, удалось бы получить и полоску. Квадрат в свою очередь состоит из отрезков. Если бы удалось построить один отрезок, то многократное его повторение дало бы квадрат. Отрезок состоит из точек, поэтому сейчас наша задача — написать только один цикл, который будет рисовать точки вплотную друг к другу. По завершении такого цикла будет получен отрезок. Это уже простая задача. Приведем текст ее решения:

```
for x:=1 to 50 do
 putpixel(10+x,10,1);
```

Пошаговое усложнение.

Шаг первый — рисование квадрата. Готовый фрагмент программы рисует 50 точек и в результате получается горизонтальный отрезок. Теперь, чтобы получить квадрат, надо этот отрезок прорисовать 50 раз, каждый раз смещая на один пиксел вниз или вверх. Иначе говоря, необходимо написать еще один цикл, который повторит уже имеющийся цикл 50 раз, изменяя вторую координату на 1 на каждом шаге. Вот новый фрагмент программы:

```
for y:=1 to 50 do
 for x:=1 to 50 do
  putpixel(10+x,10+y,1);
```

Шаг второй — рисование полоски квадратов. Сторона квадрата имеет длину 50 пикселов. Поэтому, чтобы получить новый квадрат правее уже имеющегося, его координату *X* необходимо смещать на 50 пикселов. Так как требуется несколько квадратов, то мы опять построим цикл, который несколько раз повторит уже имеющийся фрагмент программы. Вот этот фрагмент:

```
for i:=1 to n do
 for y:=1 to 50 do
  for x:=1 to 50 do
    putpixel(10+x+50*(i-1),10+y,1);
```

Шаг третий — рисование доски. Вообще-то уже видна закономерность, все, что мы делаем с координатой *X*, затем приходится делать и с координатой *Y*, поэтому новый шаг можно было бы выполнить формально, но все же немного порассуждаем. Чтобы получить новую полоску, надо уже имеющуюся сместить на 50 пикселов вниз. Так как нужно несколько полосок, то опять построим цикл, который несколько раз повторит уже имеющийся фрагмент программы, каждый раз изменяя координату *Y* на 50 пикселов так же, как это делалось с координатой *X*. Вот этот фрагмент:

```
for k:=1 to n do
 for i:=1 to n do
  for y:=1 to 50 do
```

```
for x:=1 to 50 do
  putpixel(10+x+50*(i-1),10+y+50*(k-1),1);
```

Шаг четвертый — цвета. Доска получена, но эта доска одноцветная. Цель следующего шага в определении операции перемены цвета. Кстати, обратите внимание, что все выполненные шаги довольно просты, каждое рассуждение не заключает в себе ничего сложного, а программа постепенно, но уверенно усложняется. В этом и заключается изюминка метода.

Сейчас поговорим о перемене цвета. Очевидно, что цвет можно менять только в том случае, если его значение хранится в переменной, сейчас цвет записывается константой. Введем переменную `c`, которая в начале процесса рисования равна 1.

Теперь о главном. Цвет менять необходимо в тот момент, когда завершено рисование квадрата, следующий квадрат должен быть уже другого цвета. Квадрат рисуют два внутренних цикла, следовательно, перемену цвета необходимо выполнять после их завершения. Далее приведен необходимый фрагмент программы:

```
c:=2;
for k:=1 to n do
  for i:=1 to n do
   begin
    for y:=1 to 50 do
     for x:=1 to 50 do
      putpixel(10+x+50*(i-1),10+y+50*(k-1),c);
    if c=1 then c:=2 else c:=1;
   end;
```

Исполняемая часть программы кажется полностью завершенной, мы можем записать все, что не хватает для ее работоспособности. Но, завершив программу, в процессе ее тестирования натолкнемся на интересный эффект. При нечетном количестве клеток, она вполне работоспособна, а при четном доски не получается, а получается несколько однотонных полосок. Мы делали все в соответствии с идеей метода, и даже что-то получилось, но не до конца. Это не означает, что использованный метод плох. Это означает, что метод не обеспечивает окончательной и безусловной победы над задачей и может понадобиться еще что-то. Это естественная ситуация, в нашей науке нет стопроцентных методов поиска решений.

Вернемся к проблеме. Построим две строки в виде таблицы и посмотрим, как происходит перемена цвета при четном (табл. 6.2) и нечетном (табл. 6.1) количестве клеток шахматной доски.

Таблица 6.1. *Нечетное количество*

1	2	1	2	1
2	1	2	1	2

Таблица 6.2. *Четное количество*

1	2	1	2	1	2
1	2	1	2	1	2

Чтобы понять, в чем проблема, достаточно посмотреть на вторую таблицу. В ней цвета меняются от квадрата к квадрату, но при переходе от полоски к новой полоске перемена цвета приводит к плохому эффекту, а при нечетном количестве эта перемена наоборот хороша. Поэтому сделаем так: цвета менять будем, как и меняли, но по завершению рисования полоски, если количество клеток четно, выполним еще одну перемену цвета (эта дополнительная перемена скомпенсирует плохой эффект), а если количество клеток доски нечетно, то дополнительная перемена не нужна.

Полоску прорисовывают три внутренних цикла, следовательно, вторую перемену цвета будем выполнять по их завершению. Теперь запишем всю программу со всеми необходимыми атрибутами (листинг 6.1).

Листинг 6.1

```
program example;
 uses graph;
 var
  dr,md,x,y,i,k,c,n:integer;
begin
 read(n);
 dr:=detect;
 initgraph(dr,md,'');
 c:=2;
 for k:=1 to n do
  begin
   for i:=1 to n do
    begin
     for y:=1 to 50 do
      for x:=1 to 50 do
       putpixel(10+x+50*(i-1),10+y+50*(k-1),c);
     if c=1 then c:=2 else c:=1;
```

```
    end;
   if n mod 2=0 then
     if c=1 then c:=2 else c:=1;
  end;
end.
```

В заключение. Решенная задача — очень удачный пример пошагового усложнения от примитива до сложного исходного условия. Наша задача идеально подходит для демонстрируемого метода. Конечно, так гладко не всегда получится. Скорее можно ожидать, что метод удастся применить с некоторыми издержками, путь к исходной задаче от примитива может оказаться длительным и извилистым. Но пошаговость решения и понятность каждого отдельно взятого шага является хорошей компенсацией за длительные усилия.

Глава 7

Рекурсивная снежинка

Условие задачи. Нарисовать картинку, такую как показано на рис. 7.1.

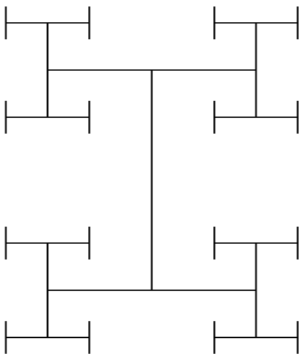

Рис. 7.1

Начнем наши рассуждения с уточнения задачи. Из формы рисунка ясно, что длины линий уменьшаются, но не ясно на сколько. Величину уменьшения можно определить самостоятельно, так как никаких прямых указаний на этот счет в условии нет. Положим, что линии уменьшаются в два раза.

Решение, общий подход. Задача кажется сложной. И метод, которым мы воспользуемся, выглядит весьма необычно. Не генерируя никаких идей, не создавая общего плана поиска решения, мы построим цепочку рассуждений, каждое из которых несложно и, как кажется, лишь немного будет продвигать нас к цели, но в конечном итоге получится требуемый результат. Воздержимся от общих схем, а просто продемонстрируем метод в действии и лишь затем скажем несколько общих слов.

Цепочка рассуждений. Для начала заметим, что каждый отрезок порождает еще два, и это своего рода рекуррентное определение очередного отрезка. Любое рекуррентное определение наводит на мысль о построении рекурсивной процедуры. Отсюда следует, что головная программа может состоять из двух команд: это рисование самого первого отрезка и вызов процедуры, рисующей новый отрезок. Ранее, однако, было сказано, что каждый отрезок порождает еще два, следовательно, в головной программе должно быть два вызова. Запишем то, что уже ясно:

```
line(300,50,300,250);
rec_uzor();
rec_uzor();
```

Далее займемся построением процедуры `rec_uzor`. Обратите внимание, что мы уже построили кусок программы и ни слова не сказали про алгоритм решения или какую-нибудь содержательную идею.

Сейчас попробуем определить, что должна знать процедура для выполнения своей работы. Процедура рисует новый отрезок от вершин уже нарисованного, следовательно, она должна знать координаты вершин предыдущего отрезка. Далее, процедура должна знать, вертикальный или горизонтальный отрезок ей рисовать, для чего необходимо сообщить ей ориентацию предыдущего отрезка. Договоримся сами с собой, что 1 означает вертикальный отрезок, а 2 — горизонтальный. Далее, новый отрезок должен быть в два раза меньше предыдущего, следовательно, длина предыдущего также должна быть известна. Вроде бы вся необходимая информация для построения отрезка уже определена, но есть еще одно небольшое соображение по поводу рекурсивной природы процедуры. Мы знаем, что необходимо позаботиться о завершении цепочки рекурсивных вызовов. Пусть вызовы прекращаются тогда, когда номер очередного вызова достигнет определенного значения. Для обеспечения такой возможности необходимо передавать номер очередного вызова. Сейчас уже возможно записать немного больше текста. Во-первых, можем записать полностью текст головной программы:

```
driver:=detect;
initgraph(driver,mode,'');
line(300,50,300,250);
rec_uzor(300,50,1,200,1);
rec_uzor(300,250,1,200,1);
```

В первом вызове сообщается, что центр очередного отрезка должен находиться в точке (300, 50), предыдущий отрезок был вертикальным, его длина была 200 пикселов и данный вызов первый.

Можем записать заголовок нашей рекурсивной процедуры

```
procedure rec_uzor(x,y,k,l,n:integer);
```

Далее построим тело процедуры, для чего проанализируем, какую информацию несет каждая из переданных величин. Понятно, что каждая величина передается для чего-то, и это что-то выполняется каким-то набором команд, следовательно, если удачно проведем анализ, то нам удастся нарастить текст программы. Начнем с последней переменной, так как она несет самую общую информацию о структуре процедуры. А именно она говорит следующее: "если ее значение не достигло некоторой критической величины, то можно выполнять необходимые действия, иначе нельзя". Пусть это критическое значение равно 10. Тогда сразу появляется следующий текст процедуры

```
if n<10 then
  begin
  end;
```

Между `begin` и `end` находится весь остальной текст процедуры, о котором нам пока ничего не известно. Возьмем следующую переменную. Она несет в себе информацию о длине, про которую известно, что она должна быть уменьшена в два раза. Это можно сделать уже сейчас и фрагмент увеличивается еще на один оператор

```
if n<10 then
  begin
   l:=round(l/2);
  end;
```

Следующая величина говорит о том, что возможны две ситуации (ранее была нарисована вертикальная или горизонтальная линия), для каждой из которых данная величина принимает свое определенное значение. Это означает, что мы можем записать условный оператор, учитывающий два значения величины `k`. Текст процедуры еще немного увеличивается.

```
if n<10 then
  begin
   l:=round(l/2);
   if k=1 then begin
   end
  else begin
  end;
  end;
```

Для тех, кто предпочитает оператор-переключатель, этот фрагмент текста можно переписать следующим образом:

```
if n<10 then
  begin
   l:=round(l/2);
   case k of
```

```
    1: begin
       end;
    2: begin
       end;
   end;
  end;
```

Наше обещание выполняется, до сих пор не было никаких рассуждений об алгоритме решения задачи, мы просто анализируем данные и рассуждаем о том, что надо сделать, чтобы получение процедурой переменных имело смысл, а текст процедуры растет, и каждый промежуточный результат кажется вполне логичным.

Продолжим анализ. Осталось две переменные, имеющие смысл координат. Они могут потребоваться только для рисования текущего отрезка. Если необходимо нарисовать горизонтальный отрезок, то его координата *Y* конечно равна той, которая была получена процедурой, а координаты *X* левого и правого конца пусть отличаются от известной координаты центра на величину длины. Для вертикального отрезка наоборот, координата *X* остается без изменений, а координаты *Y* верхнего и нижнего конца отличаются от координат центра на величину длины. Теперь можно записать две процедуры рисования отрезка.

```
if n<10 then
  begin
    l:=round(l/2);
    if k=1 then begin
                  x1:=x-l;x2:=x+l;
                  line(x1,y,x2,y);
                end
           else begin
                  y1:=y-l;y2:=y+l;
                  line(x,y1,x,y2);
                end;
  end;
```

И последний момент. В самом начале сказано, что каждый нарисованный отрезок порождает два вызова процедуры, для каждого из которых опять так же, как и в начале, простыми рассуждениями мы можем определить передаваемые параметры.

```
if n<10 then
  begin
    l:=round(l/2);
    if k=1 then begin
                  x1:=x-l;x2:=x+l;
                  line(x1,y,x2,y);
```

```
                rec_uzor(x1,y,2,l,n+1);
                rec_uzor(x2,y,2,l,n+1);
              end
          else begin
                y1:=y-l;y2:=y+l;
                line(x,y1,x,y2);
                rec_uzor(x,y1,1,l,n+1);
                rec_uzor(x,y2,1,l,n+1);
              end;
  end;
```

Текст процедуры готов полностью, как и было обещано, мы ни разу не обмолвились об алгоритме, не было никаких общих рассуждений, только небольшие пошаговые построения, но программа удалась. Ее полный текст приведен в листинге 7.1.

Листинг 7.1

```
program example;
       uses crt,graph;
       var
        driver,mode:integer;
     procedure rec_uzor(x,y,k,l,n:integer);
      var
       x1,y1,x2,y2:integer;
     begin
      if n<10 then
        begin
          l:=round(l/2);
          if k=1 then begin
                        x1:=x-l;x2:=x+l;
                        line(x1,y,x2,y);
                        rec_uzor(x1,y,2,l,n+1);
                        rec_uzor(x2,y,2,l,n+1);
                      end
                 else begin
                        y1:=y-l;y2:=y+l;
                        line(x,y1,x,y2);
                        rec_uzor(x,y1,1,l,n+1);
                        rec_uzor(x,y2,1,l,n+1);
                      end;
        end;
     end;
     begin
```

```
      driver:=detect;
      initgraph(driver,mode,'');
      line(300,50,300,250);
      rec_uzor(300,50,1,200,1);
      rec_uzor(300,250,1,200,1);
end.
```

А теперь попробуем поговорить о методе. Сущность его заключается в том, что данные прямо и непосредственно определяют выполняемые над ними действия. В каком-то смысле данные всегда определяют действия, но не всегда так явно. Например, вспомним задачу получения выборок, в ней предыдущая выборка без всякого сомнения определяет последующую, но между предыдущей и последующей выборкой располагается достаточно сложная логика, причем в этой логике возможны варианты, очень сильно отличающиеся друг от друга (в задаче о выборках у нас было два решения). В задаче о снежинке не так. Например, величина длины определяет один простой оператор присваивания, самая первая величина (номер вызова) определяет конструкцию ветвления, которая впоследствии сильно разрастается, но в момент своего написания она достаточно проста, написать ее по-другому можно, но данный вид очень естественен, и даже иное написание (например, с помощью оператора `case`) по сути будет тем же самым.

Конечно, приходится признать, что выбор задачи очень удачен и применить этот метод на произвольной задаче вряд ли получится так же эффективно, но нужно помнить, что *данные всегда в той или иной степени определяют выполняемые над ними операции*. И напоследок сформулируем еще одно очень полезное правило, дополняющее только что записанное.

==========

Правило учета логики данных

Если была введена величина, значит, она была введена для каких-то целей. Следовательно, если в вашей программе есть неиспользованная переменная, то скорее всего в вашей программе есть ошибка.

==========

Это правило уже не имеет прямого отношения к решенной задаче, оно из нее вытекает. Ведь если данные определяют действия, и с каждым данным должна быть сопоставлена группа операторов, то отсутствие этой группы прямым и явным образом указывает на возможность ошибки.

Кстати, это настолько очевидно, что даже зачастую определение неиспользованных переменных берут на себя компиляторы.

Глава 8

Ханойская башня

Это старая классическая задача, встречающаяся во многих учебниках и пособиях по программированию. Мы тоже не пройдем мимо и используем ее для того, чтобы еще раз поговорить о рекурсии. Общая структура рекурсивного решения такова:

1. Дан вычислительный (и не обязательно вычислительный) процесс, в котором результат каждого шага определяется через результат предыдущего.
2. В конечном итоге процесс обязательно приводит к какой-то тривиальной ситуации, в которой результат можно определить явным образом без дальнейших рекурсивных обращений.

Очевидно, что любая рекурсия обладает следующими свойствами: во-первых, в процессе рекурсивных вызовов изменяются какие-то данные, во-вторых, существует конфигурация данных, при которой рекурсивные вызовы прекращаются и что-то делается явным образом (обычно при вычислении каких-либо величин) или действие просто прекращается так, как это было в задаче о рисовании снежинки. Кстати, первое свойство вытекает из второго. Действительно, если ничего не будет изменяться, то говорить о том, что наступит тривиальная ситуация, бессмысленно. Простейший пример, демонстрирующий сказанное, — это программа рекурсивного счета факториала (листинг 8.1).

Листинг 8.1

```
program example;
 uses crt;
 var
  n:integer;
function factorial(n:integer):integer;
begin
 if n=1 then factorial:=1
  else factorial:=n*factorial(n-1);
```

```
end;
begin
 clrscr;
 read(n);
 write(factorial(n));
end.
```

В процессе счета факториала тривиальная ситуация — это случай при `n=1`, а от вызова к вызову `n` уменьшается на 1, что и дает гарантию завершения рекурсивных вызовов. Гарантия завершения процесса рекурсивных вызовов очень важный момент. Надо всегда помнить, что рекурсия не завершается сама по себе и что организованный вами процесс может и не дойти до тривиального случая, что создаст ошибку под названием "переполнение стека".

Если говорить об ошибках, то сразу стоит сказать еще об одной, более тонкой. Обратите внимание, что в той же программе счета факториала нет циклов, хотя нерекурсивное решение обязательно содержит цикл. Это означает, что рекурсия есть форма организации циклических процессов, и если в вашем рекурсивном решении появляются циклы, участвующие в организации основной логики, то это должно навести вас на размышление о верности логики. И, во всяком случае, рекурсивные вызовы внутри циклов всегда сильно усложняют логику, если без них возможно обойтись, то лучше это сделать.

Программа счета факториала полезна только для демонстрации общих правил, вообще-то считать факториал рекурсивной программой вряд ли хорошо. Это обычный линейный процесс, и нерекурсивное решение если не короче, то проще. Но мы немного на нем задержимся, чтобы напомнить еще одну важную деталь — рекурсивному решению всегда сопутствует рекуррентное определение. Например, наше рекурсивное решение задачи о факториале предполагает его рекуррентное определение n!=n*(n-1)!.

Если вспомнить задачу о снежинке, то также можно заметить, что большая снежинка состоит из маленьких, такой же формы, те в свою очередь из еще более маленьких и т. д.

Вернемся к задаче о Ханойской башне. Смысл ее в следующем: дано три подставки, на первой из них лежит некоторое количество дисков, таких что для любой пары дисков тот, что сверху, имеет радиус меньший того, что ниже. Необходимо переложить все диски на третью подставку, не нарушая следующих правил:

- ❒ за один раз можно брать только один диск;
- ❒ диск можно класть только на диск большего радиуса или на пустую подставку.

Начнем наши рассуждения с попытки обнаружить рекуррентную природу задачи. Это легко сделать, если внимательно рассмотреть два рисунка — рис. 8.1 и 8.2. На рис. 8.1 показана исходная задача.

Рис. 8.1. Задача о Ханойских башнях (дано)

Рис. 8.2. Задача о Ханойских башнях (процесс)

Если удастся получить ситуацию, изображенную на рис. 8.2, то исходная задача перемещения пирамиды с левой подставки на правую решится в два шага. Во-первых, диск с левой подставки перемещается на правую, а затем решается задача перемещения меньшей пирамиды со средней подставки на правую. А для того, чтобы получить такую ситуацию, надо сначала решить задачу перемещения ханойской башни без нижнего диска на среднюю подставку. Таким образом, исходная задача о перемещении башни из пяти дисков сводится к двум задачам о перемещении четырех дисков.

Рекуррентный характер задачи налицо, следовательно, поиск рекурсивного решения вполне оправдан. Тривиальный случай для построения рекурсивной процедуры тоже понятен — это башня из одного диска.

Входные данные:

- номера подставок в следующем порядке: подставка, с которой необходимо переместить текущую пирамиду, вспомогательная подставка, и подставка, на которую требуется переместить пирамиду;
- высота перемещаемой пирамиды.

Алгоритм рекурсивной процедуры будет выглядеть так:

```
Если перемещаемая высота равна 1 то
  Переместить один диск с исходной подставки на подставку-цель.
  Иначе
    Вызвать процедуру, перемещающую пирамиду с высотой на единицу
    меньше с исходной подставки на вспомогательную (то есть кроме
```

```
    нижнего диска).
    Переместить нижний диск с исходной подставки на подставку-цель.
    Вызывать процедуру, перемещающую пирамиду с вспомогательной
    подставки на подставку-цель.
```

Текст программы показан в листинге 8.2.

Листинг 8.2

```
program example;
 uses crt,graph;
 type
  mas=record
        krug:array[1..10] of integer;
        h:integer;
      end;
 var
  a:array[1..3] of mas;
  dr,md,i,n:integer;
procedure ris;
var
 i:integer;
begin
 cleardevice;
 for i:=1 to n do
  begin
   if a[1].krug[i]>0 then circle(100,200,a[1].krug[i]);
   if a[2].krug[i]>0 then circle(300,200,a[2].krug[i]);
   if a[3].krug[i]>0 then circle(500,200,a[3].krug[i]);
  end;
end;

procedure hanoy(a1,a2,a3,m:integer);
procedure perest;
begin
    a[a3].h:=a[a3].h+1;
    a[a3].krug[a[a3].h]:=a[a1].krug[a[a1].h];
    a[a1].krug[a[a1].h]:=0;
    a[a1].h:=a[a1].h-1;
    ris;
    readkey;
end;
begin
```

```
 if m=1 then perest
 else
  begin
   hanoy(a1,a3,a2,m-1);
   perest;
   hanoy(a2,a1,a3,m-1);
  end;
end;
begin
 dr:=detect;initgraph(dr,md,'');
 read(n);
 for i:=1 to n do
   a[1].krug[i]:=(n-i+1)*10;
 a[1].h:=n;a[2].h:=0;a[3].h:=0;
 ris;
 readkey;
 hanoy(1,2,3,n);
end.
```

Полученное решение — хороший пример формального подхода. Мы уже говорили о том, что для построения алгоритма весьма желательно выработать хорошее представление о процессе. Попробуйте выработать у себя хорошее представление о процессе перемещения дисков. Для двух дисков это легко, для трех — затруднительно, а попробуйте представить себе процесс для пяти или шести дисков. Это будет уже весьма непросто. Или даже попробуйте, поняв алгоритм на четырех дисках, повторить его на шести. Совершенно не очевидно, что у вас это получится. Кстати, даже в анализе задачи мы нашли существенную сложность. Рекуррентное определение говорит о том, что исходная задача сводится к двум задачам, при этом целевое использование подставок меняется. Вот эта постоянная перемена предназначения подставок сильно осложняет понимание процесса.

Но к процессу поиска решения можно подойти и иначе. Мы уже описали некую общую технологию построения рекурсивных программ. О ней сказано достаточно много, сейчас отметим только, что она состоит из ряда формальных шагов, фактически трех: определение условия завершения, организация рекурсивного вызова, выполнение текущих операций. Если это сделать корректно, то беспокоиться о правильном понимании процесса уже не надо.

В рассмотренной задаче на каждом шагу выполняется элементарное перемещение дисков и два рекурсивных вызова, отличающихся друг от друга назначением подставок (какая из них конечная, а какая вспомогательная). Точное определение назначения подставок при каждом вызове избавляет нас от необходимости понимания всего процесса перемещений. Что-то подобное

уже было в задаче о снежинке. Вспомните, мы строили рекурсивную процедуру шаг за шагом, ни слова не говоря о процессе рисования в целом.

Второе решение. В листинге 8.3 приведен текст второго решения той же задачи. Оно также имеет рекурсивный характер. Текст рекурсивной процедуры несколько сложнее (во всяком случае длиннее), но структура данных, реализующая пирамиду, существенно проще. Мы не будем подробно анализировать эту программу, попробуйте разобраться в принципах ее работы самостоятельно.

Листинг 8.3

```
program example;
   uses crt,graph;
 var
   driver,mode,i,r,n:integer;
   x:array [0..15] of integer;
 procedure gr(x:array of integer);
  var
    c:char;
  begin
   for i:=1 to n do
    begin
     circle(x[i],239,r*i);
     setfillstyle(1,i);
     floodfill(x[i]+r*i-1,239,15);
    end;
   c:=readkey;
   setfillstyle(1,16);
   floodfill(1,1,14);
  end;
procedure abc(a,b,c,a1,b1,c1:integer);
   begin
    if a>1 then  abc(a-2,a-1,a,a1,b1,c1);
    if a=1 then  begin x[a]:=b1; gr(x); end;
    x[b]:=c1; gr(x);
    if a>1 then abc(a-2,a-1,a,b1,c1,a1);
    if a=1 then  begin x[a]:=c1; gr(x); end;
    x[c]:=b1; gr(x);
    if a>1 then  abc(a-2,a-1,a,c1,a1,b1);
    if a=1 then  begin  x[a]:=a1;  gr(x); end;
    x[b]:=b1; gr(x);
    if a>1 then  abc(a-2,a-1,a,a1,b1,c1);
    if a=1 then  begin x[a]:=b1;  gr(x); end;
   end;
```

```
begin
 clrscr;
 writeln('number of circle');
 readln(n);
 r:=round(100/n);
  driver:=detect; initgraph(driver,mode,'');
 for i:=1 to n do  x[i]:=110;
gr(x);
abc(n-2,n-1,n,100,530,320);
end.
```

В заключение. У двух приведенных рекурсивных решений даже на первый взгляд есть что-то общее, но видна и существенная разница. Обратите внимание, что вызов рекурсивной процедуры во втором варианте требует шесть аргументов. Можно найти и другие отличия. Корень различий в представлении подставок и колец структурами данных. Проведите анализ структур данных, который даст понимание, почему эти два решения так сильно отличаются.

Глава 9

График соревнований

Условие задачи. Требуется составить график соревнований по виду спорта, предполагающему парные состязания. Это может быть бокс или фехтование. Вид спорта роли не играет, существенно важно выполнить следующие условия:

1. В каждом бою участвуют два спортсмена.
2. Проигравший спортсмен выбывает из соревнований.
3. В любом поединке встречаются спортсмены, проведшие или равное количество боев или у одного из них количество боев на один меньше.

Пока неясно, за что взяться в построении алгоритма, поэтому попробуем представить себе ход соревнований. Пусть некоторое количество бойцов начинают соревнования. Если их количество равно степени двойки, то тогда все достаточно просто. Выстраиваем спортсменов в один ряд. На каждом этапе в бой вступают два соседа: первый со вторым, третий с четвертым и т. д. В конце концов, останутся два бойца, проведшие по равному количеству боев. Это можно проиллюстрировать следующей картинкой — рис. 9.1.

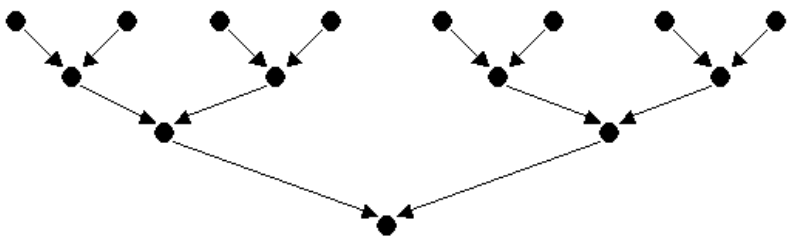

Рис. 9.1

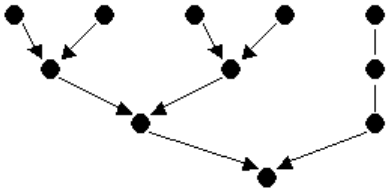

Рис. 9.2

Здесь стрелками помечен выбор пар спортсменов. Очевидно, что два спортсмена, вышедшие в финал, будут иметь одинаковое количество боев. А теперь попробуем нарисовать картинку, в которой соревнования начинает иное количество бойцов — рис. 9.2.

В этой ситуации видна проблема, последний боец при нашем методе построения пар соревнующихся два раза выходит в следующий круг без боя, следовательно, в финале у него будет на два боя меньше, чем у его соперника. Появление проблемы это хорошо, теперь у нас есть за что зацепиться в разработке алгоритма. Но прежде чем идти дальше, сформулируем небольшое правило.

==========

Правило. Проблема как источник информации

Разработка алгоритма — это решение проблем, следовательно, для того чтобы начать разработку, надо выделить какие-то неясности и ошибки. Для того чтобы это сделать, можно взять первое пришедшее в голову решение и просто посмотреть, почему оно не будет работать. Первое пришедшее в голову решение почти очевидно не будет работать, но его анализ даст важную информацию о том, в каком направлении надо думать.

==========

А теперь подумаем, в чем мы были не правы, полагая, что, сводя двух соседей, получим нужный график. Ошибка видна после небольшого раздумья. Если слева от правого крайнего спортсмена находится количество партнеров, равное степени двойки, то они на протяжении всего соревнования будут обходиться без него. По всей видимости, от нас требуется, если таковой спортсмен найдется на правом краю, позаботиться о партнере для него специально.

Заметим сразу, что в описанной ситуации есть две особенности: во-первых, спортсмен остался без партнера, и, во-вторых, количество спортсменов слева от него равно степени двойки. Понятно, что первое условие более общее и легче проверяемое. Поэтому решим так: если остался спортсмен без партнера, то в следующем круге позаботимся о партнере для него специально.

А как? Очень просто, если мы строим пары бойцов слева направо, то лишний боец окажется на правом краю, а мы в следующем круге будем строить пары справа налево. При таком уточнении метода наш боец в последнем примере окажется с партнером (рис. 9.3).

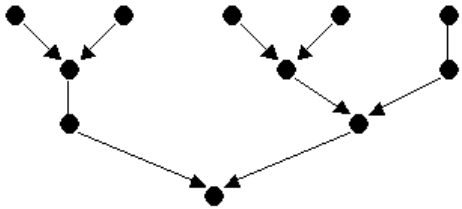

Рис. 9.3

Теперь, кажется, все хорошо, решенная проблема возникла от того, что выбор партнеров слева направо был предпочтительным, теперь мы первый раз идем слева направо, а потом, когда встретилась проблемная ситуация, будем выполнять проход справа налево. И все хорошо. *Так ли?*

Думаю, вы должны заметить серьезное нарушение логики. Мы только что сказали, что движение слева направо не должно быть предпочтительным и сделали таковым движение справа налево. А чем оно лучше? Откуда известно, что при построении пар справа налево плохая ситуация не повторится? Попробуйте построить разные примеры с разным исходным количеством бойцов, и вы быстро убедитесь, что ситуация действительно повторится, а иначе и быть не может, так как два направления (справа — налево и слева — направо) равноправны, и все, что случается в одном из них, может случиться и в другом.

Но решение проблемы уже очевидно, если, строя пары спортсменов слева направо, мы, получив лишнего бойца, изменили направление построения пар на обратное, то опять, получив лишнего бойца, необходимо поступить точно так же, то есть сменить направление.

С данного момента алгоритм, пожалуй, достаточно ясен, и можно приступать к разработке программы, если бы не одно *но*. Многие из нас глубоко убеждены, что алгоритм первичен по отношении к программе. Из этого убеждения вытекает, что если разработка удалась, если полученный алгоритм прозрачен, ясен и прост, то все проблемы, которые могут возникнуть в дальнейшем, — это проблемы техники владения языком. В принципе, это действительно так. Наше обещанное *но* связано с тем, что технические проблемы могут оказаться слишком велики, настолько велики, что прозрачный и предельно ясный алгоритм превратится в очень запутанную программу. Давайте рассмотрим алгоритм с этой точки зрения.

Ядро используемых данных — это, конечно, информация о спортсменах. Самая простая структура, позволяющая хранить информацию одинаковой природы, — это массивы. Следовательно, основной структурой данных будет массив бойцов. Программа может представлять собой цикл обработки массива, на каждом шагу которого массив (длина) уменьшается и содержимое элементов массива как-то изменяется.

Так вот эту процедуру изменения массива нам придется описывать дважды, для прохождения массива справа налево и для прохождения массива слева направо. Те, кто имеет хороший опыт работы с массивами, наверное, согласятся, что от порядка прохождения массива зависит очень многое и такое серьезное изменение, как изменение направления обхода, может сильно усложнить логику. Вероятно, это и не так страшно, может быть изменение логики окажется вполне по силам, но лучше еще немного подумать и изменить алгоритм таким образом, чтобы в любом случае обходить массив в одном и том же направлении. *Попробуем сделать это.*

Причина изменения направления обхода массива заключается в том, что то справа, то слева остается лишний боец и на следующем круге для него нужно обязательно обеспечить партнера. Для чего необходимо, чтобы этот лишний боец на следующем круге обязательно участвовал в распределении. Что же мешает сохранить направление слева направо? Рассмотрим пример из пяти бойцов. Во втором круге остается только трое спортсменов. Если сохранить проход слева направо, то третий (он был лишним) опять окажется лишним, для него опять не будет пары. Но можно начать не с первого, а со второго, тогда без пары окажется первый, а третий будет с парой гарантированно.

Наш новый метод построения пар выглядит так: если лишний боец оказался на правом краю, то проход массива начинается со второго бойца, если лишний боец окажется слева, то проход начинается с первого. Если же количество бойцов четно, то пара есть каждому и проход начинается опять-таки с первого. Эту новую идею запишем в виде алгоритма:

```
Пока бойцов больше чем 1 делать
  Составить очередной набор пар
    Если количество бойцов четное, то пары составлять из соседних,
    начиная с первого
    Если их количество нечетное, и номер этой ситуации нечетный,
    то пары составлять, начиная с первого
    Если их количество четное и номер ситуации четный,
    то пары составлять, начиная со второго
```

Мы ничего не сказали об одной технической, но чрезвычайно важной детали — структуре данных, содержащей информацию о бойцах. Хорошую подсказку о том, как их представить, дает графическое изображение хода

соревнований. В этом дереве боец фактически представлен координатой *X*, а очередной круг — координатой *Y*. Поэтому естественным представлением множества бойцов будет массив координат *X*.

Но техническая деталь заключается совсем не в этом. После каждого круга примерно половина бойцов выбывает из соревнований, а следовательно, массив необходимо переписывать. Здесь как раз и сказывается преимущество прохода массива всегда от начала к концу.

По результатам боя между первым и вторым бойцом победителя естественно записать в первую ячейку массива. При проходе от начала это проблем не вызывает. А при проходе от конца запись победителя пары, в которой участвуют последний и предпоследний, создаст существенную трудность. Место победителя этой пары где-то в середине массива, а в средние элементы массива в этот момент пока записывать никого нельзя, так как они еще только подлежат обработке. Процесс обработки до середины массива еще не дошел, и операция записи изменит ход соревнования.

Полученное решение достаточно хорошо (листинг 9.1), и на нем вполне можно остановиться, но у нас на протяжении предыдущих глав уже сформировалась традиция демонстрировать равенство рекурсивных и нерекурсивных построений. Поэтому если первый вариант нерекурсивный, то второй мы видимо должны разработать с использованием рекурсии.

Листинг 9.1

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  x:array[1..20] of integer;
  i,n,k,tp,m:integer;
procedure ris(m,y:integer);
var
 i:integer;
begin
 for i:=1 to m do
  circle(x[i],y*40,5);
end;
begin
 read(n);
 for i:=1 to n do x[i]:=20*i-10;
 dr:=detect;initgraph(dr,md,'');
 ris(n,1);
 k:=1;
 tp:=0;
```

```
 while n>1 do
  begin
   k:=k+1;
   m:=n div 2;
   if n mod 2=1 then
    begin
     for i:=1+tp to m+tp do
        x[i]:=(x[2*i-1-tp]+x[2*i-tp]) div 2;
     m:=m+1;
     if tp=0 then
      begin
       tp:=1;
       x[m]:=x[n];
      end
     else  tp:=0;
    end
   else
     for i:=1 to m do
        x[i]:=(x[2*i-1]+x[2*i]) div 2;
   n:=m;
   ris(n,k);
  end;
end.
```

Заметим, что процесс построения дерева боев не требует возврата. Дерево строится до последнего круга соревнований, в котором встречаются финалисты, и на этом все прекращается, возврат результатов не нужен. Каждый рекурсивный вызов получает на входе массив, обработанный предыдущим вызовом, выполняет обработку и передает его далее. Дополнительная информация, необходимая процедуре, — это номер круга, без него не обойтись при расчете координаты *Y* и длины передаваемого массива.

Проблема корректного формирования пар в этом варианте решается несколько иначе. Идея прохода только от начала остается. На каждом шаге обрабатывается только четное количество бойцов, а последний обрабатывается в зависимости от того, остался он или нет (а бойцов вообще может быть только четное количество) и четна или нечетна длина переданного массива. Текст программы-решения представлен в листинге 9.2.

Листинг 9.2

```
Program example;
 uses crt,graph;
 type c=array [1..100] of integer;
```

```
 var
  a: c;
  n,i,driver,mode:integer;
  s:string;
procedure ris(i,y:integer);
begin
 line(a[i],y,a[i],y+10);
 line(a[i+1],y,a[i+1],y+10);
 line(a[i],y+10,a[i+1],y+10);
end;
procedure fight(var a:c; n,k:integer);
 var i,y:integer;
begin
 y:=45+10*(k-1);
 i:=1;
 while i<=n-1 do
  begin
   ris(i,y);
   a[round((i+1)/2)]:=round((a[i]+a[i+1])/2);
   i:=i+2;
  end;
  if i=n then
   begin
    line(a[n],y,a[n],y+10);
    a[round((n+1)/2)]:=a[n];
   end;
 if n mod 2<>0 then
  begin
   n:=n div 2+1;
   k:=k+1; y:=45+10*(k-1);
   ris(n-1,y); a[n-1]:=round((a[n-1]+a[n])/2);
   for i:=1 to n-2 do line(a[i],y,a[i],y+10);
   n:=n-1;
  end
 else n:= n div 2;
 if n>1 then fight(a,n,k+1);
end;
begin
 clrscr;
 writeln('количество бойцов');
 readln(n);
 for i:=1 to n do
  a[i]:=43+30*(i-1);
```

```
 driver:=detect;
 initgraph(driver,mode,'');
 for i:=1 to n do
  begin
   str(i,s);
   outtextxy(a[i]-3,40,s);
  end;
 fight(a,n,1);
 while not keypressed do;
end.
```

В заключение. Ядро обоих полученных решений — сложная обработка массива. Массив в процессе работы алгоритма перестраивается и изменяет длину. Такая перестройка кажется чисто техническим затруднением, но его преодоление — изюминка полученных решений.

Глава 10

Поиск прямоугольника наибольшей площади

Условие задачи. Дан двумерный массив, заполненный случайным образом нулями и единицами. Найти прямоугольник наибольшей площади, заполненный единицами.

Очень важное свойство хорошего решения — это достижимость результата за как можно меньшее количество действий. Представим себе такую задачу: пусть в большом числовом множестве надо найти число с заданными свойствами. Это можно сделать простым перебором всех чисел. Если их много, то перебор может занять много времени. Если для получения решения пользователь программы обладает неограниченным временем, то переборное решение будет достаточным, но часто реальный пользователь желает получить решение за ограниченное время. Хорошее решение (способное справиться с задачей за ограниченное время), возможно, удастся получить, если изучить свойства данного множества чисел и построить явную формулу, позволяющую найти нужное число сразу. Но идеал он на то и идеал, что редко бывает достижим, к тому же если бы все задачи решались идеально, то программирование было бы не нужно, все решалось бы математическими методами.

Чаще все-таки встречаются исходные данные со слабо выраженными закономерностями, и задачи на них надо решать перебором. Наша задача — в чистом виде задача переборная. Но все-таки что-то сделать можно. И сейчас мы посмотрим, что можно сделать с полным перебором, если избавиться от него не удается.

Решение в самом общем виде выглядит как полный перебор всех возможных прямоугольников, укладывающихся в данный, и нам нужно решить две проблемы.

- Каким образом вообще организовать полный перебор всех возможных прямоугольников?
- Как организовать этот перебор так, чтобы часть вариантов не рассматривать?

Эти два вопроса выражают самый общий подход к решению переборных задач. Возьмите любую задачу, поищите в ней закономерность, и если оказалось, что задача переборная, то задайте эти два вопроса. Ответ на первый даст решение, ответ на второй поможет полученное решение существенно улучшить.

Кстати, желание не рассматривать часть вариантов заключает в себе серьезное противоречие. Если мы сможем что-то отбросить, то не означает ли это, что нам все-таки удалось найти в исходном множестве нулей и единиц закономерность? Наверное, да, но в условии сказано, что нули и единицы расставлены случайным образом, а следовательно, никакой закономерности нет и быть не может.

Выход из противоречия неожиданно прост. Действительно до начала поиска прямоугольника мы ничего не знаем о структуре данных, но уже после построения и проверки первого вложенного прямоугольника в наше распоряжение попадает вполне определенная информация и чем дальше, тем больше, а на основании этой информации мы можем корректировать свои поисковые действия. Начнем поиск решения с организации полного перебора.

Чтобы установить порядок перебора, необходимо определить, чем могут отличаться два прямоугольника. Отличий может быть три. Они могут отличаться размером, формой и положением в большом прямоугольнике. Вот по этим трем свойствам и будем устанавливать порядок.

Если говорить о размере, то порядок будет таков: начнем с поиска прямоугольника наибольшей площади, затем меньшей, затем еще меньшей и т. д. Очевидно, что наибольшей площадью обладает исходный прямоугольник, а наименьшей — прямоугольник, состоящий из одной единицы. Тогда наибольший прямоугольник — это исходный, а наименьший — это первая попавшаяся единица.

Такой порядок поиска сразу дает возможность оптимизации. Действительно, если найден прямоугольник площадью S, это означает, что поиск можно прекратить, так как прямоугольника большей площади нет (мы его уже искали и не нашли), а меньшая площадь нас не интересует.

Теперь обсудим возможные изменения формы в пределах заданной площади. Предположим, что на некотором шаге площадь равна S. Обозначим стороны проверяемого прямоугольника через A и B. Очевидно, что S = A*B. Пусть величина A изменяется от 1 до S, тогда величина B изменяется от S до 1. Так как величины сторон привязаны друг к другу, то одну из них мы можем сделать независимой величиной, а вторую вычислять через первую. А все возможные формы прямоугольников будут определяться следующей алгоритмической конструкцией:

```
Для A от 1 до S делать
  Если S делится на A нацело, то B=S/A
```

И последнее: необходимо научиться работать с положением проверяемого прямоугольника при том условии, что его площадь известна и форма, определяемая длинами сторон, также известна.

Пусть положение проверяемого прямоугольника определяется его левой верхней точкой. Это допущение нас ни в чем не ограничивает. И пусть его сторона A параллельна горизонтальной стороне искомого прямоугольника, а сторона B — вертикальной стороне исходного. Обозначим координаты левой верхней вершины проверяемого прямоугольника через x, y. Тогда все возможные положения перебираются следующей конструкцией:

```
Для всех x от 1 до длины горизонтальной стороны исходного прямоугольника.
   Для всех y от 1 до длины вертикальной стороны исходного
   прямоугольника
```

И здесь опять появляется возможность оптимизации. Пусть, например, исходный прямоугольник имеет длины сторон 200, 200, то есть это квадрат, и пусть на некотором шаге мы ищем прямоугольник площадью 100 со сторонами 10×10. Очевидно, что в точке x = 191, y = 191 прямоугольник с такой площадью просто не поместится и, следовательно, при данных значениях координат левой верхней точки никаких проверок делать не надо, как, впрочем, и при больших значениях.

Мы достаточно подробно обсудили алгоритм перебора и все возможности сокращения вариантов перебора, сам алгоритм достаточно громоздок, но логической сложности не представляет, поэтому он неинтересен и мы ограничимся программой с подробными ремарками (листинг 10.1).

Листинг 10.1

```
program example;
  uses crt,graph;
  var
   a:array[1..70,1..20] of byte;
   c,s,q,x,y,lx,ly:integer;
function square(x,y,lx,ly:integer):integer;
 var
  i,j,s:integer;
  c:char;
begin
 s:=0;
 textcolor(2);
 for i:=x to x+lx do
  for j:=y to y+ly do
   if a[i,j]=0 then s:=1;
 if s=0 then
```

```
   for i:=x to x+lx do
     for j:=y to y+ly do
       begin
         gotoxy(i,j);write(a[i,j]);
       end;
 if s=0 then square:=1 else square:=0;
end;
begin
 textcolor(15);
 clrscr;
 randomize; {Заполнение массива нулями и единицами}
 for y:=1 to 20 do
  for x:=1 to 70 do
   begin
   c:=random(40);
   if c=39 then a[x,y]:=0 else a[x,y]:=1;
   end;
{Распечатка массива на экран монитора}
 for y:=1 to 20 do
   for x:=1 to 70 do
    begin
     gotoxy(x,y);write(a[x,y]);
    end;
 s:=1400;q:=0; {Начинаем с максимально возможной площади}
 repeat
  gotoxy(40,22);write('    ');
  gotoxy(40,22);write(s);
  ly:=1;
  repeat {Перебираем все возможные длины высот}
   lx:=1;
   repeat {Перебираем все возможные значения ширины}
    if lx*ly=s then {Если на данной ширине и высоте можно построить
    прямоугольник данной площади, то осуществляем дальнейшую проверку}
      begin
       y:=1; {Перебираем все возможные координаты y левой верхней вершины}
       repeat
        x:=1; {Перебираем все возможные координаты x левой верхней вершины}
        repeat
         q:=square(x,y,lx,ly);{ Обходим прямоугольник и выясняем,
         есть ли хоть один ноль.}
         x:=x+1;
         if (x+lx>70) then x:=70;
        until (q=1)or(x=70); {Завершаем, если найден прямоугольник
        или достигнут конец исходного прямоугольника по горизонтали}
```

```
        y:=y+1;
        if y+ly>20 then y:=20;
       until (q=1)or(y=20); {Завершаем, если найден прямоугольник или
       достигнут конец исходного прямоугольника по вертикали}
      end;
     lx:=lx+1;
    until (q=1)or(lx=70); {Завершаем, если найден прямоугольник
    или величина ширины достигла максимума}
    ly:=ly+1;
   until (q=1)or(ly=20); {Завершаем, если найден прямоугольник
   или величина высоты достигла максимума}
   s:=s-1;
 until (s=1)or(q=1); {Завершаем, если найден прямоугольник
 или величина площади достигла единицы}
end.
```

В заключение. Главный вывод заключается в том, что оптимизация алгоритма возможна даже в том случае, когда в структуре исходных данных нет никакой закономерности. Очень часто перебор можно построить таким образом, что часть возможных вариантов станет очевидно ненужной. Конечно, в нашей задаче возможна ситуация расположения нулей и единиц такая, что придется пересмотреть все допустимые варианты, — это одна-единственная единица в нижнем правом углу, но с точки зрения здравого смысла такая структура данных крайне маловероятна. А те, чья математическая подготовка включает знание хотя бы основ теории вероятностей, могут убедиться, что в среднем мы можем рассчитывать на игнорирование половины вариантов.

Общая стратегия оптимизации переборных задач, полученная из рассмотренного примера, выглядит так:

1. Определим порядок перебора вариантов.
2. Определим некоторые числовые характеристики, монотонно изменяющиеся в процессе перебора. Назовем их контрольными.
3. Определим значения контрольных характеристик, при которых дальнейший перебор вариантов в заданном направлении не имеет смысла.
4. Построим алгоритм, перебирающий только те варианты, при которых контрольные характеристики не достигают своих критических значений.

Глава 11

Выборка из миллиарда

Условие задачи. Из числового интервала от единицы до миллиарда выбираются случайным образом без повторений миллион чисел и записываются в файл. Необходимо за приемлемое время выяснить, какое наименьшее число отсутствует в файле. Использовать массивы или иные структуры данных, их заменяющие, запрещается.

Изюминка данной задачи заключается в том, что не сразу понятно, в чем собственно проблема. Почему эта задача сложна? Почему нельзя использовать какой-нибудь простой, лобовой метод, ведь условие выглядит достаточно просто. Так иногда бывает. Кажется, что все хорошо и просто, поэтому требуются специальные усилия для того, чтобы хотя бы понять, с чем мы столкнулись. В такой ситуации, чтобы увидеть реальное положение дел, можно придумать любое решение, какое придет в голову, и посмотреть, почему оно не будет работать.

Разрешите предложить вам следующий вариант: упорядочим числа в файле в порядке возрастания. Алгоритм упорядочения по возрастанию никак нельзя назвать логически сложным, сортировка — хорошо изученный процесс. После упорядочения будет достаточно одного прохода файла, в ходе которого легко обнаружить первое число, чей сосед справа отличается от него более чем на единицу. Тогда найденное число из файла + единица и будет искомым. При таком подходе задача действительно выглядит элементарной. Теперь подумаем, почему это не сработает. Пусть будем упорядочивать файл наибыстрейшим образом. Какой бы мы при этом не выбрали алгоритм, придется выполнить несколько миллиардов файловых операций, а если вспомнить, что файловые операции выполняются медленно, то становится понятно, в чем изюминка задачи — *нельзя читать файл слишком много раз, надо покончить с этим делом за несколько проходов.* Если вопроса, почему эта задача сложная, не осталось, приступим к ее решению.

В поиске решения задачи мы применим метод аналогии. Его суть заключается в двух действиях:

1. Вспомним задачу с похожим условием, но уже решенную, и решенную достаточно удачно.
2. Попытаемся переложить известное решение на данную задачу.

Наша задача очень похожа на задачу поиска корня заданного числа. В задаче о корне имеется числовой интервал, в котором очень много чисел, и среди них необходимо найти одно, обладающее определенными свойствами (оно является корнем). Есть, правда, очень существенное отличие. Числа интервала в задаче о корне расположены в порядке возрастания, в нашей же задаче они расположены совершенно хаотично. Но может быть, это окажется не очень существенным. Если для вас задача о корне неизвестна, тогда приведем текст ее решения в листинге 11.1.

Листинг 11.1

```
program example;
 uses crt;
 var
  a,x,l,p,e:real;
begin
 clrscr;
 read(a,e);
 l:=0;
 p:=a;
 repeat
  x:=(l+p)/2;
  if x*x>a then p:=x else l:=x;
 until (p-l<e);
 write((p+l)/2);
end.
```

Мы не будем подробно разбирать алгоритм этой программы, она достаточно прозрачна, и вы можете разобраться в ней сами. Для нас важен принцип. А принципиально здесь происходит следующее: на каждом шагу главного цикла интервал поиска разбивается на два, затем определяется новый интервал, в котором далее надо вести поиск корня. Попробуем использовать этот принцип для нашей задачи.

Поделим миллиардный интервал на два интервала по 500 миллионов. Где может находиться искомое число? Очевидно, в первом, так как 500-миллионный интервал миллионом чисел не заполнить. Теперь поделим первый 500-миллионный интервал на два и т. д. Рано или поздно заключение

о том, что интервал, содержащий искомое число, будет первым, окажется несправедливым. Поэтому давайте для любой пары интервалов разработаем более универсальный метод.

Заметим, что в нашем простом способе определения мы делали вывод на основании сравнения того, сколько чисел должен содержать интервал и сколько он их содержит на самом деле. Разовьем эту идею, она кажется перспективной. Предположим, на текущий момент есть интервал [3000, 3999]. Очевидно для того, чтобы быть полностью заполненным, он должен содержать 1000 чисел (напомним, что числа в файле не повторяются, иначе чисел в интервале может оказаться существенно больше 1000 и при этом интервал будет не заполненным). Мы знаем, что все они не меньше 3000 и не больше 3999. Это легко проверяемый факт. Достаточно один раз прочитать файл и подсчитать, сколько в нем чисел, укладывающихся в данный интервал. Если окажется, что чисел меньше, чем длина интервала, то, следовательно, в нем есть число, не записанное в файл.

Процесс разбиения интервала на два следующих необходимо завершать, когда в текущем интервале останется два числа, тогда следующее разбиение даст два интервала, в одном из которых будет одно число, а в другом ни одного. Первое число этого другого интервала и есть искомое. Запишем алгоритм. Его главные объекты — два счетчика. Первый считает числа, входящие в первый интервал, второй — числа, входящие во второй интервал.

```
Пока длина интервала больше единицы, делать
  Разбить интервал на два
    Читать файл и для каждого числа, прочитанного из файла, делать:
      Если число принадлежит первому интервалу, то первый счетчик
      увеличивается на единицу
      Иначе второй счетчик увеличивается на единицу
    Текущий интервал — это интервал, значение счетчика которого
    меньше длины интервала.
```

Записанный алгоритм неплохо выражает общую идею, но в нем много неопределенных понятий. Самый главный вопрос — так ли важно делить именно на два интервала, может быть лучше делить на 3 или 5. Выигрыш, который мы можем получить, изменив количество интервалов, заключается в количестве необходимых прочтений файла, а как вы помните, именно многократное чтение файла и создает проблему для решения задачи. Посмотрим, как быстро мы получим решение с разным количеством интервалов.

Для миллиарда количество шагов будет равно показателю степени двойки, при которой степень превысит миллиард. При делении интервала на 10 мы получим количество шагов, равное показателю степени 10, при котором степень превысит миллиард. Таким образом, действительно увеличение количества интервалов дает выигрыш во времени. Но, рассуждая подобным

образом, мы приходим к противоречию, так как наша цепочка рассуждений быстро приведет к необходимости разбить миллиард на миллиард интервалов. Это, конечно, бессмыслица, и надо подумать, откуда она появилась.

Во-первых, мы забыли про память. Чем больше интервалов, тем большее количество счетчиков мы должны помнить, а это затраты на память. Кстати, сейчас мы можем сформулировать одно общее правило.

==========

Правило компенсации

Если вам удалось улучшить какую-то характеристику вашей программы, то вы должны ожидать, что какая-то другая характеристика станет хуже.

==========

Это правило не имеет характера закона. В принципе можно так улучшить программу, что все ее характеристики станут лучше. Она станет быстрее работать, в то же время уменьшатся затраты на память, ее текст станет короче, понятнее и т. д. Это вполне возможно, но чаще всего такое всеобщее улучшение будет означать, что предыдущий вариант программы был совсем плох. А чем лучше ваша программа, чем изощреннее алгоритм, и чем эффективнее использованы средства языка, тем полученное правило действует точнее.

В нашей задаче, увеличивая количество интервалов, мы теряем в памяти, таким образом, количество интервалов ограничено объемом имеющейся оперативной памяти. Но на самом деле мы теряем и в скорости! Для того чтобы перейти на следующий шаг, необходимо найти счетчик, значение которого меньше длины текущего интервала, а если массив будет велик, то его обход также займет время и совершенно не очевидно, что эти затраты времени можно игнорировать.

Конечно, для идеального решения необходимо взвесить все выигрыши и потери, а также определиться, какое количество интервалов выгодно. Но точное решение этого вопроса вряд ли возможно. Во-первых, для этого необходимо очень точно знать скорость выполнения операций процессором, и, во-вторых, предстоит сравнивать величины, имеющие различную природу (это скорость счета и память), а корректно сравнивать можно только величины одной природы. Поэтому не будем искать идеальное решение (так как непонятно, что это такое) и ограничимся достаточно хорошим. При количестве интервалов, равном 10, нам потребуется всего 12 шагов, то есть 12 прочтений файла, что немного, для миллионного файла такая работа займет приемлемое время.

Следующая неясность алгоритма — это разбиение интервала. Как именно это сделать? Вопрос не слишком прост, так как текущий интервал не только уменьшается в 10 раз, но и достаточно произвольно скачет по исходному миллиарду (искомое число может оказаться где угодно).

Ранее было заявлено два важных утверждения относительно интервала: его длина переменна, его положение переменно. Следовательно, для задания интервала нужны две величины: длина и число, с которого он начинается, левая точка.

Проблем с расчетом длины нет. На первом шаге длина равна миллиарду, на каждом последующем длина уменьшается в десять раз. Левая точка первого шага — это ноль. Обозначим текущую левую точку как НАЧАЛЬНАЯ. Попробуем посчитать, куда она переместится после очередного шага. Пусть новым интервалом будет интервал с номером $L = 1, 2, 3, ..., 10$. Обозначим длину предыдущего интервала через ДЛИНА. Тогда Новую НАЧАЛЬНУЮ точку можно вычислить по формуле:

```
Новая НАЧАЛЬНАЯ = (L – 1)*(ДЛИНА / 10)
```

На самом деле эта формула несколько неверна. Она вычисляет новое положение НАЧАЛЬНОЙ точки относительно НАЧАЛЬНОЙ точки текущего интервала. А еще надо учесть, что все числа левее предыдущей НАЧАЛЬНОЙ точки уже исключены из рассмотрения. Это можно учесть следующим образом:

```
Новая НАЧАЛЬНАЯ = Предыдущая НАЧАЛЬНАЯ + (L – 1)*(ДЛИНА / 10)
```

Следующий неясный момент заключается в процедуре определения, к какому из интервалов принадлежит очередное прочитанное число. Рассмотрим первый шаг с исходным интервалом. В каждый из внутренних интервалов может входить 100 миллионов чисел. В первый входят числа от нуля до 99 999 999. И вот тут небольшой момент срабатывания интуиции. Заметим, что если любое число из этого интервала мы разделим нацело на 100 000 000, то результатом будет ноль. Точно так же любое число из интервала 100 000 000 до 199 999 999, деленное на 100 миллионов, даст 1 и т. д. То есть если мы число разделим на 100 миллионов и прибавим 1, то как раз и получим номер интервала. Запишем это в общем виде, для чего введем обозначения: ДЛИНА — длина интервала, ЧИСЛО — число, прочитанное из файла и ИНДЕКС — индекс вложенного интервала.

```
ИНДЕКС = Целая часть от (ЧИСЛО / ДЛИНУ) + 1
```

Эта формула полезна для понимания того, что нам требуется, но она неточна. Попробуйте проверить ее действие хотя бы для второго шага. Ошибка будет в следующем. Предположим, в файле есть число пятьсот миллионов. На первом шаге (деление на 100 миллионов) это число по нашей формуле даст индекс 5, а на втором это же число даст индекс 50, то есть несуществующий, а на третьем это же число даст индекс 500 и т. д. Отсюда следует, что перед тем, как вычислять индекс, надо выяснить, входит ли прочитанное число в интересующий нас интервал, если сказать более точно, то необходимо выяснить, находится ли это число между НАЧАЛЬНОЙ точкой и точкой НАЧАЛЬНАЯ + ДЛИНА интервала. Теперь наша формула превращается в небольшой алгоритмический фрагмент.

```
Если ЧИСЛО >= НАЧАЛЬНОГО и ЧИСЛО < НАЧАЛЬНОЕ + ДЛИНА
  То ИНДЕКС = Целая часть от (ЧИСЛО / ДЛИНУ) + 1
```

Это еще не все уточнения, но прежде чем идти дальше, сделаем небольшое замечание о методе наших рассуждений. В этой задаче мы выводим уже вторую формулу. Обратите внимание, как это делается. Во-первых, мы записываем формулу расчета из самых простых и очевидных соображений, чаще всего исходя из какого-то простого примера, например исходя из начального состояния данных. Затем мы берем более сложный пример, и если формула отказывается работать, то корректируем ее с учетом новой, полученной информации, потом ищем следующую ошибку и т. д. Конечно, может возникнуть вопрос, откуда мы знаем, что ошибки закончатся? Точного, строгого ответа на данный вопрос дать невозможно, но есть одно очень сильное соображение. Если один пример считается правильно, а другой нет, то между этими примерами есть какое-то принципиальное отличие, не сводящееся к разнице в количественном значении. В наших двух примерах, рассмотренных ранее, различие следующее: миллиард — это начальный интервал, второй интервал в 100 миллионов — это интервал, впервые полученный расчетным способом. Такова принципиальная разница. Примеров, между которыми есть принципиальная разница, не может быть сколь угодно много, а следовательно, ошибки должны закончиться. Конечно, термин "принципиальная разница" не очень ясен, но будем надеяться, что интуитивного понимания окажется достаточно, все равно дать его точное определение не получится.

А теперь вернемся к формуле. Мы рассуждаем об интервале, но интервалов у нас на каждом шаге два. Это большой интервал, который мы разбиваем на меньшие, и сами меньшие. Впрочем, раньше уже мелькало понятие вложенного интервала, но как-то четко об этих различиях ничего не было сказано. Это нормально. Практически невозможно сразу дать точные определения всем используемым понятиям, но сейчас это сделать необходимо, *мы выходим на конкретные формулы, и неточности в понятиях приведут к неточности в формулах.* И такая неточность в нашей формуле уже появилась. В фрагменте, приведенном ранее, дважды используется ДЛИНА, а что при этом имеется в виду? В команде ЕСЛИ имеется в виду длина текущего интервала, в котором осуществляется поиск, а в формуле имеется в виду длина интервала разбиения (давайте далее называть его так), который в 10 раз меньше текущего. Поэтому уточним формулу следующим образом:

```
Если ЧИСЛО >= НАЧАЛЬНОМУ и ЧИСЛО < НАЧАЛЬНОЕ + ДЛИНА
  То ИНДЕКС = Целая часть от (ЧИСЛО / ДЛИНУ РАЗБИЕНИЯ) + 1
```

А в программе ДЛИНУ РАЗБИЕНИЯ будем вычислять, как десятую часть от ДЛИНЫ. И это еще не все. Предположим, что текущий интервал включает в себя число 500 миллионов, а длина текущего интервала например 1000. Ничего невозможного в этом нет. То есть число 500 миллионов окажется вполне допустимым, а индекс, который оно породит, будет 500 миллионов деленное на 1000, что, конечно же, ошибка. И эта ошибка заключается

в том, что индекс надо считать также с учетом числового значения НАЧАЛЬНОЙ точки. В конечном итоге формула приобретает такой вид:

```
Если ЧИСЛО >= НАЧАЛЬНОМУ и ЧИСЛО < НАЧАЛЬНОЕ + ДЛИНА
  То ИНДЕКС = Целая часть от ((ЧИСЛО - НАЧАЛЬНАЯ)/
  ДЛИНУ РАЗБИЕНИЯ) + 1
```

Конечно, трудно предположить, что все эти рассуждения могли быть проведены до того, как написана программа. На самом деле разумно написать программу с первоначальным вариантом формулы, затем в процессе отладки находить ошибки и, отталкиваясь от них, проводить дальнейшие рассуждения. Так всегда легче. Редко кто может провести полный анализ всех проблем до написания программы.

Теперь все логические проблемы решены и можно сесть за написание программы, но есть еще одно серьезное *но*. В условии задачи речь идет о миллиардном интервале. Как мы будем работать с такими длинными целыми? Конечно, есть в языках программирования тип "длинное целое", но все ли арифметические операции с ним работают корректно? Еще одна проблема связана с тем, что нам нужен миллион целых, случайных и неповторяющихся. Их генерация — это отдельная математическая и алгоритмическая проблема, к нашей задаче не имеющая прямого отношения. Однако сказать, что получение файла данных не наша проблема, мы тоже не можем, так как необходимо проверить программу.

Если посмотреть на полученный алгоритм, то видно, что длина интервала для него не имеет принципиального значения. Поэтому будет правильно, если для отладки мы используем достаточно большой, но не миллиардный интервал. Можно даже ограничить себя типом "целое". В общем, ограничим себя проверкой логики. Кто хочет, может заняться решением всех технических проблем, связанных с большими числами. Итак, мы будем работать с 9000 числами, выбранными из 10 000-го интервала (листинг 11.2).

Листинг 11.2

```
program example;
 uses crt;
 var
  f:file of integer;
  long, a, x, n, i: integer;
  s:array[1..10] of integer;
begin
 clrscr;
 assign(f,'dat');
 long:=10000;
 x:=0;
 for n:=1 to 4 do
```

```
  begin
   reset(f);
   {Инициализация массива счетчиков}
   for i:=1 to 10 do s[i]:=0;
   while not eof(f) do
    begin
     read(f,a);
     {Проверка на вхождение в текущий интервал}
     if (a>=x) and (a<x+long) then
       begin
        {Расчет индекса интервала разбиения}
        i:=((a-x) div (long div 10))+1;
        s[i]:=s[i]+1;
       end;
    end;
   for i:=1 to 10 do write(s[i],' ');
   writeln;
   readkey;
   i:=1;
   {Поиск первого незаполненного интервала разбиения}
   while s[i]>=(long div 10) do i:=i+1;
   writeln('i=',i);
   {Переход к новому интервалу}
   x:=x+(i-1)*(long div 10);
   long:=long div 10;
  end;
  write(x);
end.
```

И последнее. Как заполнить нужный файл? Мы решили, что исходная структура данных не очень велика, поэтому можно воспользоваться простым методом. Будем заполнять файл так:

```
Пока не записано нужное количество чисел, делать
  Генерируем случайное число
  Проверяем, есть ли оно в файле
  Если нет то
    Записываем его в файл
    Счетчик записанных чисел увеличиваем на единицу.
```

В листинге 11.3 приведена программа заполнения файла.

Листинг 11.3

```
program example;
 uses crt;
 var
```

```
  f:file of integer;
  a,b,n:integer;
  flag:boolean;
begin
 randomize;
 clrscr;
 assign(f,'dat');rewrite(f);
 for n:=0 to 1999 do
  write(f,n);
 while n<6000 do
  begin
    a:=random(10000);
    reset(f);
    flag:=true;
    while not eof(f) do
     begin
      read(f,b);
      if b=a then flag:=false;
     end;
    if flag then
     begin
      n:=n+1;
      clrscr;
      gotoxy(40,12);write(n);
      seek(f,filesize(f));
      write(f,a);
     end;
  end;
end.
```

В заключение. Рассмотренная задача, пожалуй, единственная в книге, где так много времени и усилий тратится на вывод формул, поэтому, пользуясь моментом, обратим ваше внимание на одну важную проблему. В выводимых формулах мы считаем индексы. Обратите внимание на поправку +1. Интересно, что такие поправки при расчете индексов появляются достаточно часто. Либо +1, либо −1. Их появление всегда объясняется логикой решения, но почему-то они часто упускаются неопытными программистами. Возможно, дело в том, что эти поправки порождаются какими-то мелкими и более тонкими эффектами, чем главная идея решения, может быть здесь срабатывает какой-то психологический эффект. Но в любом случае полезно запомнить, что при *расчете индексов, независимо от того, для чего он считается, в конечной формуле может появиться поправка в единицу.*

Глава 12

Раскладывание колечек по штырькам

Условие задачи. В каждую клетку шахматной доски воткнут штырек. В нашем распоряжении есть некоторое количество колечек, которые можно нанизывать на штырьки, причем на один штырек можно нанизать несколько колечек. Требуется подсчитать, сколькими способами можно распределить все эти колечки по штырькам. Два распределения считаются разными, если на двух соответствующих штырьках находится разное количество колечек.

Прежде всего, обратим внимание на одно важное обстоятельство. Два разбиения считаются различными, если в них хотя бы один штырек содержит различное количество колечек и ничего не сказано о взаимном расположении штырьков. Из этого следует, что способ расположения штырьков роли не играет и слова о квадратной доске — это не более чем способ ввести нас в заблуждение. Вытянем доску в линию, линию представим в виде массива, а кольца пусть будут единицами. Тогда нанизывание колечка на штырек сведется к добавлению еще одной единицы к элементу массива. С учетом сказанного условие можно переформулировать так:

Второе условие. Дан одномерный массив длины N и число L. Найти все возможные различные разложения числа L по элементам массива. Массив считается разложением числа L, если сумма его элементов равна L.

А можно сформулировать это же условие вообще без применения программистской терминологии.

Третье условие. Дано число L. Найти все возможные разбиения этого числа на суммы положительных чисел, при условии, что количество слагаемых не превышает N, и перемена места слагаемого дает другое разбиение.

Это хороший пример формализации условия, то есть записи условия на специальном языке, слова которого имеют четко определенный смысл. Сформулировать задачу несколькими способами всегда полезно. Это дает дополнительную информацию к размышлению, а иногда удачная переформулировка

даже может подсказать и очень изящное решение. В нашем случае как минимум, удалось избавиться от лишнего понятия доски и привести все к понятиям, легко переводимым на язык программирования.

Третье условие удобно для понимания, но оно, пожалуй, более математическое, чем программистское, поэтому мы далее воспользуемся вторым. По типу определения требуется построить объект с установленными свойствами (массив, сумма элементов которого равна заданному числу). Такие задачи решаются двумя способами. Во-первых, можно придумать метод получения всех возможных объектов и из них выбрать те, которые удовлетворяют поставленному условию. Во-вторых, можно придумать способ получения объектов, уже удовлетворяющих поставленному условию. Обычно это делается так: строится первый объект, уже удовлетворяющий заданному условию, затем последующий строится из имеющегося. Это сложнее, и, кроме того, при таком подходе надо еще доказать, что в построенной последовательности все объекты будут удовлетворять условию.

Мы используем в решении задачи оба подхода, получим два решения и посмотрим на их плюсы и минусы.

Первое решение. Задача по формулировке похожа на следующую: найти среди всех N-значных чисел такие, что сумма их цифр равна заданному числу. Действительно, массив, по элементам которого раскладывается число L, можно представить себе как число по основанию L, а его элементы как цифры. Тогда идея решения выглядит так: требуется получить все N-значные числа по основанию L, а на экран вывести только те, сумма цифр которых равна L.

Для того чтобы получить, например, все десятичные числа, не превышающие заданное число, необходимо взять ноль и прибавлять к нему единицу, пока не получится самое большое число. При этом гарантированно будут получены все десятичные числа от единицы до большого числа. Очевидно, что эта процедура не зависит от основания системы счисления. Опишем ее для числа с произвольным основанием, записанным в виде массива.

Возьмем число по основанию L, запишем его в массив. Прибавим к первому разряду единицу. Возможны две ситуации. Первый разряд меньше величины основания, тогда все нормально и никаких дополнительных действий выполнять не надо. Но возможно, величина первого разряда превысит величину основания, тогда необходимо этот разряд обнулить и единицу перенести в следующий, но возможно, в следующем разряде также возникнет переполнение, тогда перенос нужно выполнять до тех пор, пока не будет найден разряд, такой что при добавлении к нему единицы ситуация переполнения не возникнет. Заметим, что один раз возникнет неустранимая ситуация переполнения со старшим разрядом, но это как раз и будет означать, что все числа уже получены. Запишем алгоритм прибавления единицы:

```
Прибавим единицу к первому разряду.
От первого разряда до предпоследнего делать
```

```
  Если значение текущего разряда больше основания, то
    Текущий разряд обнулить.
    К следующему разряду прибавить единицу.
Если Значение старшего больше основания, ТО его Значение равно значению основания.
```

Замечание

Наш алгоритм несколько отличается от алгоритма сложения двух чисел. При сложении столбиком двух разрядов, если возникает переполнение, то, выполняя перенос, мы не обнуляем текущий разряд числа результата. В текущем разряде может остаться число, меньшее основания. Различие это происходит от того, что в нашем алгоритме прибавляется не произвольное число, а единица.

Построенный алгоритм верен, но это наиболее очевидное решение, и оно, как все очевидные решения, обладает недостатком. Если в какой-то момент окажется, что текущий разряд имеет значение, меньшее или равное основанию, то последующие разряды также не могут иметь значения больше основания, и проверка от первого до последнего разряда теряет смысл. Это небольшой недостаток. С учетом того, что длина массива невелика, выигрыш в раннем завершении проверки несущественен. Но если вам нравится бороться за эффективность, то можно предложить другой алгоритм. Кстати, желание придумать более быстрый алгоритм всегда похвально. Итак, немного более эффективный алгоритм:

```
К первому разряду прибавить единицу.
Начиная со второго и до тех пор, пока значение разряда больше основания
на единицу, делать
  Если разряд не последний, то
    Обнулить разряд
    Следующий увеличить на единицу
  Иначе значение разряда равно значению основания.
```

Небольшая техническая проблема с увеличением текущего числа на единицу решена. Теперь можно приступить к разработке общего алгоритма. Ядро его составит цикл, внутри которого прибавлением единицы формируется новое N-значное число по основанию L. Затем алгоритм просчитает сумму цифр этого числа, и если она равна L, то массив-число распечатывается. Далее записан алгоритм:

```
Инициализировать массив нулями (исходное число)
Пока не получено последнее число, делать
  Выполнить алгоритм прибавления единицы
  Найти сумму элементов массива
  Если сумма равна L, то распечатать массив на экран.
```

Несколько последних технических замечаний (см. листинг 12.1).

- Функция `final` выясняет, достигнуто последнее число или нет. Последнее число — это число, в котором все разряды содержат значение, равное основанию, то есть L.
- В исходной формулировке задачи говорилось о двумерной доске. Поэтому вводится сторона этой доски, затем она возводится в квадрат, таким образом получается длина соответствующего одномерного массива.
- Процедура `print` используется для распечатки массива, и в ней же вычисляется сумма цифр.

Листинг 12.1

```
program example;
 uses crt;
 var
  i,n,l,k:word;
  a:array[1..100] of word;
function final:boolean;
var
 i:word;
 q:boolean;
begin
 q:=false;
 for i:=1 to n do
  if a[i]<l then q:=true;
 final:=q;
end;
procedure print;
var
 i,s:integer;
begin
 s:=0;
 for i:=1 to n do s:=s+a[i];
 if s=l then
  begin
   for i:=1 to n do
    write(a[i],' ');
   writeln;
  end;
end;
begin
 clrscr;
```

```
 read(n,l);
 n:=n*n;
for i:=1 to n do a[i]:=0;
 while final do
  begin
   a[1]:=a[1]+1;
   i:=1;
   while a[i]=l+1 do
    begin
     if i<n then
      begin
       a[i]:=0;
       a[i+1]:=a[i+1]+1;
      end
     else a[i]:=l;
     i:=i+1;
    end;
   print;
  end;
end.
```

Решение, полученное ранее, имеет одно серьезное достоинство и один серьезный недостаток. Достоинство его в простоте логики, а недостаток в том, что всех возможных чисел много больше, чем нужных. Если доска велика, а колечек немного, то даже на мощном компьютере программа будет заметно "подвисать". И этот недостаток настолько серьезен, что выигранное преимущество, пожалуй, теряется. Поэтому сейчас подумаем о **втором подходе**. Его сущность заключается в том, чтобы из одной подходящей комбинации строить следующую подходящую, игнорируя массу ненужного. Это, конечно, существенно сложнее логически, но выигрыш в скорости будет огромный.

При таком подходе задача выглядит достаточно сложной, поэтому подумаем, как ее можно упростить, или подумаем о том, что ее усложняет. Ясно, что это задача на построение различных комбинаций из значимых чисел (то есть больших нуля), нули нас не интересуют, так как они не вносят никакого вклада в общую сумму. Но нули могут в массиве располагаться различными способами. Для наших поисков более удобно вообще-то говорить не о расположении нулей, а о расположении значимых чисел. Следовательно, комбинации будут строиться в два этапа: во-первых, необходимо построить комбинацию позиций, в которые возможно ставить значимые числа, а затем для заданной комбинации позиций необходимо построить все возможные разложения числа L. То есть задача явно разбивается на две. Поговорим о первой.

Как построить комбинацию позиций? Если допустимую позицию помечать единицей, а недопустимую нулем, то комбинация позиций — это двоичное число длины N. Задача перебора всех двоичных чисел уже решалась в *главе 3*, в задаче о выборках, поэтому будем считать эту проблему решенной.

Как строить разложение? Исходные данные — это массив, содержащий числа, и массив-двоичное число, определяющий, в какие позиции можно записывать числа. Как именно строить очередное разложение? Можно попытаться придумать формулу, которая, исходя из уже полученного разложения, вычисляла бы числа для каждой позиции нового, а можно записать в допустимые позиции единицы, а затем добавлять по какому-либо правилу единицы, пока не будет получена нужная сумма. Мы пойдем вторым путем, а пока сформулируем важное правило.

==========

Правило малых изменений

Если от вас требуется построить набор чисел с заданной количественной характеристикой (например, определенная общая сумма, как в нашей задаче), то можно построить набор, в котором данная характеристика меньше, чем требуемая, и затем небольшими изменениями подгонять ее к требуемому значению.

==========

Наиболее очевидный способ построения минимальной суммы — это простое заполнение единицами массива числа. К примеру, вот так:

B	0	1	1	0	1	1
A	0	1	1	0	1	1

Здесь массив B — это массив двоичного числа, а массив A — массив, сумму элементов которого требуется довести до L. Значение этой исходной суммы может быть больше, чем L, может быть равно L, а может быть меньше L. Рассмотрим все три случая.

- Если значение суммы равно L, то мы уже получили одно из допустимых разложений, и прибавление еще одной единицы ничего не даст.
- Если значение суммы меньше L, то, значит, решение еще не получено, но добавлением некоторого количества единиц мы получим решение, и, возможно, вариантов решений будет много.
- Если значение суммы больше L, то решений нет, прибавление даже одной единицы только ухудшит ситуацию, эта комбинация допустимых позиций тупиковая, ее можно не рассматривать.

Примерную структуру алгоритма можно представить в виде цикла, работающего до тех пор, пока не получено последнее двоичное число, то есть число, состоящее из одних единиц. Для каждого двоичного числа необходимо построить минимальное разбиение, состоящее из одних единиц, и если окажется, что это минимальное разбиение уже дает нужную сумму или нужная сумма может быть из нее получена, то выполняется работа по дополнению минимальной суммы до значения L. А теперь самый сложный вопрос. Как дополнять минимальную сумму?

Предположим, очередная комбинация построена и заполнена единицами. Предположим, что количество единиц равно H. Тогда дополнительно не хватает L−H единиц. Мы можем добавить в первый разряд 1, 2, 3, ..., L−H единиц. Предположим, мы добавили в первый разряд t1 единиц, тогда нам не хватает еще L−H−t1 единиц, которые можно восполнить во втором, третьем, ... разрядах.

Наша задача — перебрать все возможные разложения. Следовательно, необходимо для первого разряда рассмотреть все возможные дополнительные суммы от 1 до L−H, а для второго — все возможные дополнительные суммы от 1 до L−H−t1, для третьего — все возможные дополнительные суммы от 1 до L−H−t1−t2.

То есть если длина стороны квадрата равна, например, 2, и соответственно длина одномерного массива, представляющего этот квадрат, равна 4, то полный перебор всех разложений для очередной допустимой комбинации позиций будет состоять из четырех вложенных циклов. Внешний из них имеет границы изменения параметра от 1 до 4, следующий от 1 до 3 и т. д. Это для длины стороны, равной 4. Длина стороны может быть различной, следовательно, количество вложенных циклов также должно быть различным. Построить переменное количество циклов можно с помощью механизма рекурсии. Например, так:

```
Процедура Много_Циклов(параметр)
Если параметр меньше длины массива, то
  Для всех значений от параметра до длины массива делать
    Вызывать процедуру Много_Циклов(параметр+1)
```

Это примерная конструкция, наверное, она не вполне отвечает нашим целям. Поэтому следующим шагом мы проанализируем ее, посмотрим, что в ней не отвечает поставленным целям и как ее усовершенствовать.

Во-первых, заметим, что данная конструкция просто показывает, как организовать произвольное количество вложенных циклов, здесь нет ни одной содержательной команды, ради которых, собственно, эти циклы и создаются. Наша же главная содержательная команда — это прибавление единицы к разряду. Добавим ее и перепишем алгоритм следующим образом:

```
Процедура Много_Циклов(параметр)
Если параметр меньше длины массива, то
```

```
Для всех значений от параметра до длины массива делать
  К текущему разряду прибавить единицу.
  Вызывать процедуру Много_Циклов(параметр+1)
```

Этот алгоритм выглядит уже довольно содержательно, но если вы не поленитесь и напишите по нему программу, то результат будет отрицательным. Поэтому попробуем найти расхождение между алгоритмом и изложенной ранее идеей.

В идее "сказано", что к каждому разряду может быть прибавлено много единиц. Но построенный алгоритм к каждому разряду прибавит не более одной. Действительно, в текущем вызове процедуры ее цикл пробегает массив один раз, следовательно, в текущем цикле к конкретному разряду будет прибавлено не более одной единицы, а следующий вызов начинает свою работу уже со следующего разряда. Можно решить эту проблему, передавая в следующий вызов тот же параметр, который данный вызов и получил, но тогда рекурсивные вызовы никогда не закончатся, чтобы в этом убедиться, посмотрите на условие завершения рекурсии. Чтобы все-таки завершить рекурсию, мы доопределим условие выхода, указав границу увеличения для разряда. Очевидно, что рекурсивную процедуру нет смысла вызывать, если сумма чисел массива достигла L. Обозначив для краткости длину массива через N, запишем новый вариант алгоритма:

```
Процедура Много_Циклов(параметр)
Если параметр меньше N и Сумма меньше L, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить единицу
    Вызывать процедуру Много_Циклов(параметр)
```

Алгоритм стал еще более содержательным, но программа, составленная по нему, все равно не будет работать, поэтому продолжим анализ. Первая ошибка бросается в глаза сразу. Если какая-то величина используется, то ее значение должно быть определено. В нашем тексте используется переменная `Сумма`, а ее значение никак и нигде не изменяется. Ее изменение учесть легко, как только к разряду (любому) прибавляется 1, сумма также должна измениться на единицу. Новый вариант алгоритма:

```
Процедура Много_Циклов(параметр)
Если параметр меньше N и Сумма меньше L, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить единицу
    К Сумме прибавить единицу
    Вызывать процедуру Много_Циклов(параметр)
```

А теперь произойдет удивительная вещь. Следующую довольно серьезную ошибку мы исправим исходя из свойств рекурсии, независимо от смысла

задачи. Выход из рекурсивного вызова предполагает, что данные, с которыми работала процедура до того, как она вызвала сама себя, сохранились. Это не всегда так, бывает в программировании всякое, но вообще-то смысл рекурсии в том, что ее механизм берет на себя заботу о сохранности промежуточных данных. В нашем алгоритме до вызова увеличивается разряд и увеличивается значение суммы. Вернем их значения обратно, то есть после вызова уменьшим значения разряда и суммы.

И последнее, сумму можно сделать глобальной переменной, а можно и локальной. Локальные величины всегда лучше, но возиться с передачей массива не очень хочется, поэтому массив у нас будет глобальным, а переменную `Сумма` сделать локальной труда не составляет. Впрочем, это уже вопрос вкуса. С учетом сказанного наш алгоритм окончательно выглядит так:

```
Процедура Много_Циклов(Сумма, параметр)
Если параметр меньше N и Сумма меньше L, то
  Для всех значений от параметра до длины массива делать
    К текущему разряду прибавить единицу
    К Сумме прибавить единицу
    Если Сумма = L, то массив Числа вывести на экран
    Вызывать процедуру Много_Циклов(параметр)
    Отнять единицу от разряда
    Отнять единицу от Суммы
```

Итак, мы решили главную логическую проблему и у нас есть уже решенный вопрос о получении всех двоичных чисел. Осталось прописать общую структуру алгоритма. Подробно разбирать его не будем, это всего лишь небольшая техническая работа.

```
Заполняем нулями массив двоичного числа
Пока не получено последнее двоичное число, делать
  Получить очередное двоичное число и сразу посчитать
  минимальную сумму
  Если минимальная сумма меньше или равна L, то
    Массив Числа заполнить единицами в соответствии с массивом
    двоичного числа
    Если полученная минимальная сумма равна L, то массив Числа
    вывести на экран, так как это искомый результат
    Вызывать рекурсивную процедуру дополнения массива Числа
```

Вот, собственно, и все. Теперь можно написать программу (листинг 12.2).

Листинг 12.2

```
program example;
 uses crt;
 var
  b,a:array[1..100] of word;
  num,i,n,l,sum:word;
function final:boolean;
var
 i:integer;
 q:boolean;
begin
 q:=false;
 for i:=1 to n do
  if b[i]=0 then q:=true;
 final:=q;
end;
procedure Init;
var
 i:integer;
begin
 for i:=1 to n do
   if (b[i]=1) then a[i]:=1  else a[i]:=0;
end;
procedure print;
var
 i:integer;
begin
 num:=num+1;
 write(num,') ');
 for i:=1 to n do
  write(a[i],' ');
 write('     ');
 for i:=1 to n do write(b[i],' ');
 writeln;
end;
procedure Plus_1(sum,x:integer);
```

```
var
 i:integer;
begin
 if (x<=n) and (sum<l) then
  begin
   for i:=x to n do
    begin
      if b[i]=1 then
      begin
       a[i]:=a[i]+1;
       sum:=sum+1;
       if sum=l then print;
       if sum<l then  Plus_1(sum,i);
       a[i]:=a[i]-1;
       sum:=sum-1;
      end;
    end;
  end;
end;
function digit:word;
var
 i,k,m:integer;
begin
 i:=1;
 while (b[i]=1) and (i<=n) do i:=i+1;
 b[i]:=1;
 for k:=1 to i-1 do b[k]:=0;
 m:=0;
 for i:=1 to n do
  if b[i]=1 then m:=m+1;
 digit:=m;
end;
begin
 clrscr;
 read(n,l);
 n:=n*n;
 num:=0;
```

```
 for i:=1 to n do b[i]:=0;
 while final do
  begin
   sum:=digit;
   if sum<=1 then
    begin
     Init;
     if sum=1 then print;
     Plus_1(sum,1);
    end;
  end;
end.
```

В заключение. Обратите внимание на уровень сложности разработанного алгоритма и написанной по нему программы. Это работает золотое правило механики. С его действием мы уже сталкивались раньше и будем говорить о нем и далее. Сущность его в том, что выигрыш в одном компенсируется проигрышем в другом. Наш выигрыш в эффективности программы потребовал существенно более сложной логики.

Глава 13

Разбиение кучи камней на две равного веса

Условие задачи. Дано множество камней разного веса. Разбросать их на две кучи максимально одинакового веса.

Задача имеет очевидное решение. У нас уже есть алгоритм построения всех выборок. Пусть первая куча — это очередная выборка, а вторая куча — это то, что осталось в исходной куче. Тогда решение задачи сводится к построению выборок и простому запоминанию, какая из них дает наименьшую разность. Это решение исключительно неэффективно. Из N предметов можно построить 2^N выборок, что уже для N=10 дает 2048 выборок, для более существенного значения N количество выборок становится просто заоблачным. В общем, необходимо придумать более эффективный алгоритм, способный отсекать большое количество выборок. Можно пойти по пути построения "разумного метода", то есть такого, который вообще ничего не перебирает, а просто сразу строит то, что нужно. То есть речь идет о поиске закономерности. И здесь мы опять сталкиваемся с проблемой "отсутствия оснований для закономерности", с которой мы уже сталкивались в *главе 8* в задаче о поиске прямоугольника. Сущность проблемы применительно к нашей задаче звучит так. Закономерность ищется, исходя из какой-то известной информации, каких-то ключевых фактов, но про изначальную кучу камней ничего не известно, камни ведь собираются в нее произвольным образом, следовательно, информации нет, а стало быть, нет и закономерности.

Если вас эти рассуждения испугали, то это неверная реакция. Выявление проблемы помогает в поиске пути решения. Если проблема препятствует поискам, значит, ее преодоление может дать путь и к решению задачи. Наше препятствие заключается в отсутствии информации о куче камней, следовательно, эту информацию надо добыть. Попробуем.

К большому сожалению, дальнейшие рассуждения заведут нас в тупик и заставят потратить много времени, но это будет поучительно и совсем не впустую, поэтому сейчас займемся поиском красивого и простого решения.

Единственно, что известно про камни, это то, что они имеют вес и он может быть разным. Единственно, что может быть известно про два камня (если они отличимы по весу) — это то, что один из них, вероятно, тяжелее другого. А возможность сравнения двух камней по весу дает нам возможность разложить кучу на ряд камней по увеличению веса. Если это сделать, то перед нами будет уже не безликое нагромождение камней неопределенного веса, а упорядоченное множество, заключающее в себе хорошую закономерность. А теперь попробуем ее использовать.

В отсутствии желания перебирать все выборки, процедура раскладки камней будет выглядеть так: берем очередной камень и принимаем решение о том, в какую кучу его положить, если удастся придумать, как только единожды принимать такое решение в отношении каждого камня, то вместо 2^N действий мы выполним N простых. Поразмыслим о процедуре принятия решения.

Мы имеем упорядоченное множество камней, в программе это будет, видимо, массив чисел. Берем первый камень, так как множество упорядочено, это либо самый тяжелый, либо самый легкий, пусть для определенности это будет самый тяжелый. Положим его в левую кучу. Есть ли смысл следующий камень класть в эту же кучу? Может быть есть, но тогда мы весовой разрыв увеличиваем, более логично положить его в правую и так поступать до тех пор, пока правая куча не станет тяжелее левой, после чего по такому же правилу класть камни опять в левую и так до тех пор, пока камни не закончатся. Сказанное можно записать следующим алгоритмом.

```
Упорядочиваем множество камней по убыванию веса
Пока камни не закончились
  Берем очередной
  Если вес куч различен, то
    Кладем его в кучу меньшего веса
    Иначе кладем его в левую
```

Алгоритм построен посредством вполне логичных рассуждений, если по нему написать программу, ее текст приведен далее, то можно для небольших тестовых примеров обнаружить, что она дает правильный результат, но означает ли это ее абсолютную правильность?

К сожалению, нет. Во-первых, примеры ничего не доказывают. Примером можно опровергнуть утверждение, но нельзя доказать, таков закон логики. Во-вторых, для пущей уверенности нам нужно просчитать достаточно большой пример, а в силу быстрого роста количества выборок, это практически невозможно. Что касается наших рассуждений, то при всей внешней логичности они носят не строгий характер, и основываться на них нельзя. Выход из положения в четком строгом доказательстве достижимости нашим алгоритмом правильного результата. Первый шаг в поиске доказательства — просчет примеров. Это, во-первых, даст какую-то информацию о процессе

и, во-вторых, чисто психологическую уверенность в том, что есть смысл искать доказательство. Возьмем следующую кучу камней: 18, 17, 13, 10, 10, 5, 4, 3, 3, 3, 1. Построим таблицу разложения (табл. 13.1).

Таблица 13.1. *Разложение камней*

Шаг	Левая куча	Сумма	Правая куча	Сумма
1	18	18		
2			17, 13	30
3	18, 10, 10	38		
4			17, 13, 5, 4	39
5	18, 10, 10, 3	41		
6			17, 13, 5, 4, 3	42
7	18, 10, 10, 3, 3	44		
8			17, 13, 5, 4, 3, 1	43

Очевидно, что полученное разложение оптимально. Таких примеров можно построить много, мы ограничимся одним. Для начала доказательства нужна хорошая идея. Заметим из нашего примера, что разница между промежуточными суммами уменьшается. Это можно даже строго доказать, опершись на тот факт, что веса камней уменьшаются. Что это нам даст? Только то, что разности уменьшаются! А уменьшение величины абсолютно не гарантирует того, что величина достигнет возможного минимума.

Еще одна хорошая идея. Пусть дано множество чисел. Определим минимальную разницу, с которой это множество можно раскидать на два. А вторым шагом покажем, что при нашей стратегии именно минимальное значение и достигается. Новая идея споткнется о случайный характер исходного множества чисел. Математика умеет давать точные оценки, когда дело касается рядов чисел или последовательностей, но никто и никогда не оценивает количественно случайные последовательности, сама постановка этого вопроса не имеет смысла.

Иная идея. Давайте немного обдумаем постановку задачи. Предположим, мы завершили раскладку камней и получили некоторое решение. Далее нас интересует, нет ли более точного решения. Предположим, оно есть. Можно ли от нашей раскладки перейти к этой другой, более точной? Конечно, можно. Для этого некоторую группу камней требуется переложить из левой кучи в правую и наоборот — другую группу надо переложить из правой кучи в левую. Итак, необходимо две группы камней поменять местами. Оценить количество камней в них и их вес опять-таки не получится, так как в самом общем случае они имеют самый общий состав, то есть случайный.

Но допустим, что каждая из этих групп может состоять только из одного камня. То есть допустим, что если раскладка не оптимальна, то ее можно улучшить перемещением только двух камней — одного камня из левой кучи в правую и одного камня из правой кучи в левую. Если эта гипотеза верна, то для доказательства исходного утверждения будет достаточно доказать, что при выбранной стратегии раскладки камней в конечной ситуации нет пары камней, изменение положения которых ведет к уменьшению разности между весами левой и правой кучи.

Убедиться в разумности гипотезы о двух камнях можно опытным путем (хотя, конечно, это не доказательство). Возьмите достаточно большую кучу камней, половину камней положите вправо, половину влево, но так, чтобы разница в весах между ними была очевидно неоптимальной, и попробуйте уменьшать разницу перекладыванием либо одного, либо как максимум двух камней. Вы быстро убедитесь, что это действительно эффективно работает, но вот доказательство будет сложным, если оно вообще возможно.

Наше описание попыток доказательства оптимальности стратегии не заняло много места, но тем не менее, чтобы провести все эти выкладки, нужна серьезная интеллектуальная работа, а ее тупиковый характер должен навести на мысль об ошибочности теоремы. Если мы потратили много времени на ее доказательство, есть смысл потратить некоторое время на попытку ее опровержения. Опровержение иногда дело более простое. Наверное, потому что ложных утверждений неизмеримо больше, чем истинных. Кроме того, уже говорилось, что ничего нельзя доказать примером, а вот опровергнуть можно. Поищем контрпример. Не будем описывать процесс поиска, приведем его сразу. Исходная куча: 36, 25, 12, 10, 8, 7, 1. Наша стратегия дает такую раскладку — табл. 13.2.

Таблица 13.2. *Разложение камней*

Шаг	Левая куча	Сумма	Правая куча	Сумма
1	36	36		
2			25, 12	37
3	36, 10	46		
4			25, 12, 8, 7	52
5	36, 10, 1	47		

Полученная разность равна 5. Перекладыванием двух камней эту разность можно уменьшить. А именно имеет место следующая раскладка (36, 12, 1), (25, 10, 8, 7), разность между ними равна 1.

Итак, красивая идея была неверна, но это не означает, что вся проделанная работа напрасна. Если бы вы искали опровергающий пример самостоятельно,

то скорее всего заметили бы, что это непросто, и существует большой класс ситуаций, для которых наш красивый алгоритм работает хорошо. Если есть желание, то было бы полезно определить этот класс, если он очень велик, то красивый алгоритм имеет большой смысл, хотя и теряет универсальность. Во-вторых, наша работа очень важна для понимания технологии поиска доказательства. Общая схема всех наших попыток состояла из нескольких шагов, который мы опишем в виде правила.

==========

Правило замены теоремы

- ❒ Заметим, что если некоторая гипотеза истинна, то исходная теорема сводится к другой более простой.
- ❒ Докажем эту гипотезу.
- ❒ Докажем теорему, к которой свелась исходная.

==========

И хотя наша задача доказательства универсальности красивой идеи не решилась, все же приведем в листинге 13.1 текст реализующей ее программы.

Листинг 13.1

```
program example;
uses crt;
var M: array[1..100] of integer; P: array[0..100] of 0..1;
    i, j, count, c1, c2, sum1, sum2, t: integer;
    f: boolean;
begin
 clrscr;
 readln(count);
 writeln;
 for i:=1 to count do
  readln(M[i]);
 writeln;

 for i:=count-1 downto 1 do
  for j:=1 to i do
   if M[j]<M[j+1] then begin
    t:=M[j]; M[j]:=M[j+1]; M[j+1]:=t;
   end;

 writeln;
 sum1:=0; sum2:=0; c1:=0; c2:=0;
 for i:=1 to count do
```

```
  if sum1>=sum2 then begin
   sum2:=sum2+M[i];
   c2:=c2+1;
  end
  else begin
   sum1:=sum1+M[i];
   c1:=c1+1;
  end;
 writeln(sum1,' ',sum2); writeln;
 readln;
end.
```

Заметим, наше утверждение о том, что упорядочение кучи камней дает необходимую для решения закономерность, оказалось надуманным. В какой-то степени это так. Но во-первых, никто и не говорил, что вполне логичные рассуждения обязательно приводят к безусловной истине. К таковой истине не всегда приводят и математически строгие умозаключения, а мы рассуждали нестрого и, следовательно, могли ошибиться. Во-вторых, мы не совсем ошиблись, ведь решение все-таки получено, оно только не универсальное, а универсальные решения нужны не всегда. Иногда нужен метод, который может немного, но делает свою работу быстро. А именно такой метод мы и получили.

А теперь все же займемся универсальным решением проблемы. Универсальное решение задачи полного перебора — это, конечно же, полный перебор. Никакой идеи, достойной серьезного обсуждения, у нас не будет, просто опишем набор технических действий.

Решение заключается в построении выборок (листинг 13.2). Выборки мы уже получали с помощью представления двоичным числом. Алгоритм построим как цикл, внутри которого вычисляется очередное двоичное число. Если это число представляет, например, левую кучу, тогда правая куча — это то, что осталось. После построения левой и правой куч вычисляется разность их весов, и если эта разность меньше уже найденной, то запоминается новая разность и массивы левой и правой куч.

Листинг 13.2

```
program example;
 uses crt;
 var
  b,a,c1,c2:array[1..20] of word;
  n,i,s1,s2,min:word;
  q:boolean;
function proverka:boolean;
```

```
var
 q:boolean;
 i:word;
begin
 q:=false;
 for i:=1 to n do
  if b[i]=0 then q:=true;
 proverka:=q;
end;
procedure add_1;
var
 k,i:word;
begin
 i:=1;
 while (b[i]=1) and (i<=n) do i:=i+1;
 b[i]:=1;
 for k:=1 to i-1 do b[k]:=0;
end;
procedure mass;
var
 k:word;
begin
  for k:=1 to n do
   if b[k]=1 then
    begin
     c1[k]:=a[k];
     c2[k]:=0;
    end
   else
    begin
     c1[k]:=0;
     c2[k]:=a[k];
    end;
end;
begin
 clrscr;
 read(n);
 for i:=1 to n do
  begin
   read(a[i]);
   b[i]:=0;
  end;
 q:=true;
```

```
 while proverka do
  begin
   add_1;
   s1:=0;
   s2:=0;
   for i:=1 to n do
     if b[i]=0 then s1:=s1+a[i]
      else s2:=s2+a[i];
   if q then
    begin
     min:=abs(s1-s2);
     q:=false;
     mass;
    end
   else
    begin
     if abs(s1-s2)<min then
      begin
       min:=abs(s1-s2);
       mass;
      end;
    end;
  end;
  writeln('min=',min);
  for i:=1 to n do
   if c1[i]>0 then write(c1[i],' ');
  writeln;
  for i:=1 to n do
   if c2[i]>0 then write(c2[i],' ');
  writeln;
end.
```

В заключение. Попытка найти красивое решение провалилась. Иначе и быть не могло. Наши надежды на построение закономерности строились на песке. Закономерность вообще нельзя строить, ее можно только обнаружить. Это может оказаться весьма сложным и хитроумным делом. В *главе 22* мы столкнемся с примером, в котором также вроде бы даны совершенно случайные данные, но там удастся построить красивое решение. А рассмотренная задача все же переборная в чистом виде.

Глава 14

Обратная польская запись. Прямая и обратная задача

Условие задачи. В этой главе мы попробуем найти решение двух взаимообратных задач. В первой дано арифметическое выражение, состоящее из натуральных чисел, знаков арифметических операций и скобок, необходимо преобразовать его в равнозначное в форме обратной польской записи. И во второй задаче, наоборот, преобразовать выражение, записанное в виде обратной польской записи, в обычную форму.

Что такое обратная польская запись? Это форма записи арифметического выражения, позволяющая записать любое выражение без скобок, при этом арифметические операции записываются в том же порядке, в каком и выполняются. В обратной польской записи аргументы записываются перед знаком своей операции через какой-либо разделительный знак. Например, 2+2 можно записать так: 2 2 +, мы в качестве разделительного знака будем использовать пробел. Пример более сложного выражения: 4*(2+3) равнозначно 4 2 3 + *.

Вычислим выражение во второй форме. Выполнять умножение нельзя, так как перед знаком умножения нет чисел. Первый знак, перед которым есть два числа, это знак сложения, следовательно, необходимо выполнить операцию сложения. Выполним ее и получим следующее выражение: 4 6 * . В этом выражении единственный знак операции — это знак умножения и перед ним два аргумента. Выполняем операцию и получаем 24. Действительно обошлись без скобок. Идея, наверное, уже понятна, но для лучшего усвоения попробуем перевести более сложное выражение: 2*((1+3+4)−8*(5+3−3)).

Метод перевода заключается в том, что на каждом шагу обработки арифметического выражения мы работаем только с двумя числами и одной операцией. Операции обрабатываем последовательно, иначе говоря, обратная польская запись — это не более чем форма обычного человеческого способа выполнения вычислений. Начнем с последней скобки. Первая операция 5+3. После перевода 5 3 +. Далее вычтем тройку, это даст 3 5 3 + −.

Следующий шаг умножение, переведя его, получим 8 3 5 3 + − *. Далее, чтобы не загромождать текст длинными рассуждениями, оформим всю процедуру перевода в виде таблицы:

5+3	5 3 +
5+3–3	3 5 3 + –
8*(5+3–3)	8 3 5 3 + – *
1+3	1 3 +
1+3+4	4 1 3 + +
(1+3+4)–8*(5+3–3)	4 1 3 + + 8 3 5 3 + – * –
2*((1+3+4)–8*(5+3–3))	2 4 1 3 + + 8 3 5 3 + – * – *

Можете вычислить оба окончательных выражения и убедиться, что результат у них одинаков. А метод счета для новой необычной формы намного проще. Наверное, такому методу проще обучиться и человеку. В современной младшей школе на уроках математики специально учат работать со скобками, и не у всех учеников это сразу получается, если бы основным способом записи в арифметике была бесскобочная запись, то обучение арифметике шло бы легче. Но как-то так исторически сложилось, что основной метод записи скобочный, а бесскобочная запись — это интересный фокус.

Если метод построения обратной польской записи понятен, можно начать поиск программистского решения. Для начала заметим, что в случае элементарного арифметического выражения, состоящего из одной операции, обратная польская запись получается простым переносом знака операции перед аргументами. Сразу возникает естественная идея использовать прием переноса знака для любого выражения. Идея достаточно разумна. Действительно, конкретный знак "ничего не знает" обо всем выражении, он видит только аргумент слева и аргумент справа, и чтобы (знаку) стать знаком обратной польской записи, он должен перепрыгнуть через правый аргумент. Аргумент справа вполне может оказаться сложным выражением, но это проблема скорее техническая, чем принципиальная. Сложность может быть в двух вещах:

- аргумент справа может оказаться выражением в форме обратной польской записи, состоящим не только из одного числа, а из нескольких чисел и знаков;
- аргумент справа может оказаться еще не обработанным скобочным выражением.

Это проблемы действительно не принципиального, а технического характера. В первом случае, если справа стоит обратная польская запись, то проблема

переноса знака сводится к проблеме различения уже обработанной части выражения и необработанной. Представьте себе ситуацию, вы берете знак, переносите его через число, стоящее справа от него, и обнаруживаете еще одно число. Вывод может быть только один. Это следующее число есть часть выражения, уже записанного в форме обратной польской записи, так как в традиционной форме не могут два числа стоять рядом без знака между ними.

Другая ситуация. Вы берете знак, переносите его через число и обнаруживаете знак. Вот такая ситуация уже не однозначна. Этот знак может быть как частью обработанного, так и частью необработанного выражения. Причем сам знак не может нести в себе никаких дополнительных признаков, позволяющих определить, что он есть такое. А если признаков нет, то их необходимо создать. Например, перенесенный знак (являющийся частью новой записи) мы можем выделять точками. Тогда проблема решается легко. Если перенесенный знак наталкивается на еще один знак, то надо поискать точки выделения. Если они есть, то мы имеем дело с частью новой записи, если их нет, то мы имеем дело с необработанной записью.

Вторая проблема — это проблема баланса скобок. Если на пути переносимого знака встречаются скобки, то необходимо пройти закрывающих скобок ровно столько, сколько и открывающих. Это можно и не доказывать, это практически очевидно.

Алгоритм можно оформить как процесс последовательного переноса знаков за правый аргумент с учетом ситуаций, описанных ранее. Каждый переброшенный знак будем выделять точками, для того чтобы его можно было отличить от еще не обработанного. По завершении процесса достаточно удалить все точки и оставшиеся скобки. В листинге 14.1 приведен текст программы.

Листинг 14.1

```
program example;
 uses crt;
 var
  strok:string;
  alfavit:array[1..2] of set of char;
  n:integer;
procedure remove(sign1,sign2:char);
var
 num,n,tek,n_skobka,st,tochka:integer;
 c:string;
function prov(c:char;num:integer):boolean;
begin
 if not (c in alfavit[num])
   then prov:=false
```

```
   else
    begin
     if (strok[tek-1]='.') and (strok[tek+1]='.')
       then prov:=false else prov:=true;
    end;
end;
begin
if sign1='+' then num:=2 else num:=1;
tek:=1;
repeat
 if (strok[tek]=sign1) or (strok[tek]=sign2) then
   if (strok[tek-1]<>'.') or (strok[tek+1]<>'.') then
   begin
     {remove}
     st:=tek;
     n_skobka:=0;
     repeat
      tek:=tek+1;
      if strok[tek]='(' then n_skobka:=n_skobka+1;
      if strok[tek]=')' then n_skobka:=n_skobka-1;
     until ((n_skobka=0) and prov(strok[tek],num)) or (n_skobka<0);
     if (strok[tek]=')') and (n_skobka=0) then tek:=tek+1;
     c:='.'+strok[st]+'.';
     insert(c,strok,tek);
     tek:=st;
     strok[st]:=' ';
   end;
  tek:=tek+1;
until tek>=length(strok);
end;
begin
 clrscr;
 alfavit[1]:=['*','/','+','-'];
 alfavit[2]:=['+','-'];
 readln(strok);
 remove('*','/');
 remove('+','-');
 {skobki}
 while pos('(',strok)>0 do delete(strok,pos('(',strok),1);
 while pos(')',strok)>0 do delete(strok,pos(')',strok),1);
 while pos('.',strok)>0 do delete(strok,pos('.',strok),1);
 writeln(strok);
 readln;
end.
```

А сейчас займемся обратной задачей. Дано выражение, записанное в форме обратной польской записи. Перевести его в традиционную форму. Поиск решения не займет много времени с учетом того, что прямая задача уже решена. Мы можем пойти обратным путем, то есть переносить знак справа налево до того места, где ему должно стоять. Но необходимо точно определить, где это место.

Во-первых, заметим, что в обратной польской записи все знаки стоят правее своих аргументов. Это означает, что алгоритм можно построить как цикл прохода строки, на каждом шагу которого выясняется, не знак ли очередной символ, а если да, то символ вырезается и переносится влево. Индекс, пробегающий строку символов, всегда уходит вправо, а знаки влево. Это означает, что у нас нет, как в предыдущей задаче, проблемы выяснения, с каким знаком мы имеем дело — обработанным или необработанным, любой знак справа от текущей позиции индекса — это необработанный знак.

Далее, предположим, что при выполнении прямого преобразования, переносся знак, мы оставляли на его месте пробел, и пробелы в строке есть только там, где ранее стояли знаки. Тогда справедливо следующее утверждение: законное место знака — это первый пробел слева от него при условии, что последовательность переноса знаков правильная, такая как описана ранее (в прямой задаче), то есть если знаки пронумерованы слева направо, то в первую очередь переносятся те, которые имеют меньшие номера. Данное утверждение позволяет очень легко найти законное место знака, это, как и сказано в утверждении, первый пробел слева.

Проблема скобок также решается достаточно легко. Если в процессе переноса была обнаружена закрывающая скобка, то процесс переноса надо продолжать до тех пор, пока не будет достигнут баланс скобок.

В листинге 14.2 приведена программа преобразования выражения, записанного в виде обратной польской записи, в обычную форму.

Листинг 14.2

```
program example;
 uses crt;
 var
  a:string;
  alf1,alf2:set of char;
  uk:integer;
procedure work;
var
 i:integer;
 c:char;
```

```
 flag:boolean;
begin
 c:=a[uk];
 delete(a,uk,1);
 uk:=uk-1;
 i:=uk;
 flag:=false;
 while a[i]<>' ' do
  begin
   if (((c='*') or (c='/')) and ((a[i]='+') or (a[i]='-'))) then
flag:=true;
   i:=i-1
  end;
 a[i]:=c;
 if flag then
  begin
   insert('(',a,i+1);
   uk:=uk+1;
   insert(')',a,uk+1);
   uk:=uk+1;
  end;
end;
begin
 clrscr;
 alf1:=['+','-','*','/'];
 alf2:=['1','2','3','4','5','6','7','8','9','0'];
 readln(a);
 uk:=1;
 repeat
  if a[uk] in alf1  then work;
  uk:=uk+1;
 until uk>length(a);
 write(a);
end.
```

В заключение. Решенная задача, и прямая и обратная — хороший пример задачи на выполнение сложного символьного преобразования. Для таких задач бывает полезно построить модель того, как бы эту задачу решал человек. Конечно, далеко не всегда это эффективно. Например, моделировать человеческое поведение в численных расчетах вряд ли бывает разумно, а что касается операций преобразования символьных множеств, то зачастую это единственно возможный путь поиска решения.

Нам было дано две задачи, прямая и обратная. Обратимость задач привела к хорошей обратимости алгоритмов. Вообще-то это большая удача. Далеко не всегда так бывает. Если прямой алгоритм эффективен, это не означает, что так же эффективен будет и обратный. То же самое можно сказать и об остальных свойствах алгоритма.

Но тем не менее на обратимости алгоритмов можно выстроить целый метод решения задач. Представьте себе, что вам встретилась задача, решение которой очень долго не находится. Вполне может оказаться, что обратная задача существенно проще. Конечно, если обратная задача проще, это может означать, что прямой и обратный алгоритм не переводимы друг в друга, но вам нужна только идея. Поэтому:

- ❒ сформулируйте обратную задачу;
- ❒ не доводя ее до уровня программы, сформулируйте основные идеи возможного решения;
- ❒ подумайте, как эти идеи соотносятся с прямой задачей.

Вероятность того, что идея обратной задачи поможет вам лучше понять прямую, очень велика.

Глава 15

Самый длинный путь рубки

Условие задачи. На стандартном поле 8×8 расставлены черные шашки и одна белая. Необходимо найти самый длинный путь рубки для белой шашки. В решении использовать указатели.

Требование использовать именно динамическую память на первый взгляд можно рассматривать как каприз составителя задачи. Но на более внимательный второй взгляд это не каприз, а хорошая косвенная подсказка. Ясно, что совокупность всех возможных путей рубки представляет собой ориентированный граф, а графы очень хорошо представляются связными списками.

Поэтому сейчас, прежде чем мы начнем обсуждение задачи, немного вспомним технику работы со связными списками. В листингах 15.1 и 15.2 без подробных объяснений приведены два примера: создание линейного связного списка и создание двоичного дерева.

Листинг 15.1. Пример простого связного списка

```
Program example1;
 Uses crt;
 Type
  shablon=^zapis;
  zapis=record
         a:integer;
         next:shablon;
        end;
 Var
  uk1,uk2:shablon;
  i:integer;
begin
```

```
 clrscr;
 {Создается указатель на первый элемент списка и запоминается его адрес
 в дополнительном указателе}
 new(uk1);uk2:=uk1;
 {Связный список заполняется числами}
 uk1^.a:=1;
 for i:=2 to 10 do
  begin
   new(uk1^.next);
   uk1:=uk1^.next;
   uk1^.a:=i;
  end;
 {Вспоминается адрес первого элемента}
 uk1:=uk2;
 {Связный список проходится еще раз, начиная с первого элемента}
 for i:=1 to 10 do
  begin
   write(uk1^.a,' ');
   uk1:=uk1^.next;
  end;
end.
```

Листинг 15.2. Пример построения двоичного дерева с использованием связного списка

```
Program example2;
 Uses crt;
 Type shablon=^zapis;
 zapis=record
        a:integer;
        left:shablon;
        right:shablon;
       end;
 Var
  tree1,tree2:shablon;
  max:integer;
procedure Create_Tree(tree:shablon;n:integer);
var
 new_tree:shablon;
begin
 if n<=max then
  begin
   {Переход на левый узел}
```

```
   new(tree^.left);
   new_tree:=tree^.left;
   new_tree^.a:=random(10);
   Create_Tree(New_tree,n+1);
   {Переход на правый узел}
   new(tree^.right);
   new_tree:=tree^.right;
   new_tree^.a:=random(10);
   Create_Tree(New_tree,n+1);
  end;
end;
procedure View_tree(tree:shablon;x,n:integer);
begin
 gotoxy(x,2*n);write(tree^.a);
 if n<=max then
  begin
   {Переход на левый узел}
   View_tree(tree^.left,x-(20 div n),n+1);
   {Переход на правый узел}
   View_tree(tree^.right,x+(20 div n),n+1);
  end;
end;
begin
 randomize;
 clrscr;
 read(max);
 {Создается и заполняется числовым значением корневой элемент дерева}
 new(tree1);tree1^.a:=random(10);
 {Запоминается адрес корневого элемента}
 tree2:=tree1;
 {Создается дерево}
 Create_Tree(tree1,1);
 {Повторный проход дерева}
 View_Tree(Tree2,40,1);
end.
```

После того как мы освежили свои представления о технике работы со связными списками, можно вернуться к исходной задаче.

Ранее было сказано, что совокупность путей рубки хорошо представляется ориентированным графом, следовательно, иная более программистская постановка задачи звучит так: дан ориентированный граф, найти его самую длинную ветвь (или все самые длинные ветви, что не намного сложнее).

Конечно, новая формулировка не совсем идентична первоначальной. Новая формулировка исходит из данности графа, а в исходной постановке задачи дана позиция на шашечной доске, а это совсем не одно и то же.

Поэтому мысль разделить задачу на две — построение графа и поиск на графе — будет вполне естественной. Получив позицию, мы преобразуем ее в граф, а затем в соответствии с новой формулировкой подумаем о том, как обойти этот граф в поисках самой длинной ветви.

Примечание

Обратите внимание, что мы как бы не торопимся с поиском решения. Наши рассуждения носят, с первого взгляда, подготовительный характер и не имеют прямого отношения к идее решения. О том, как собственно решать задачу, еще не было сказано ни одного слова. А вывод о возможности разбиения задачи на две нетерпеливому человеку может показаться даже лишним. Он может сказать: "Я хочу сразу придумать, как именно искать самый длинный путь". И скорее всего, если этот человек не без способностей, то так оно и будет. А дальше будет одно большое НО. Придумать идею можно, но ее надо еще и реализовать! И нужно сказать, что зачастую возникающие технические проблемы настолько велики, что эффект от идеи сходит на нет. Что же мы фактически сделали, переформулировав задачу? А мы как раз и озаботились хотя бы обозначением этих будущих технических проблем, приблизили постановку к тем структурам данных, которыми мы реально располагаем в языке программирования.

Внимательный читатель мог бы сказать, что ранее мы начинали именно с идеи, выраженной в общепонятных терминах, и зачастую даже не математических. Но здесь нет противоречия. Надо просто видеть разницу в задачах. Если задача заключает в себе очень большую логическую проблему, то конечно, главные усилия должны быть сосредоточены на разработке логики. Рассматриваемая задача — то, что называется технически сложная, поэтому и подход к ней иной. Конечно, что сложно технически, а что логически — это решается каждым разработчиком относительно себя. Что для одного сложно, то для другого легко! Что для одного сложная логика, то для другого технические детали!

Если вы согласны с тем, что было сказано в примечании, то вы должны быть согласны и с тем, что наш следующий шаг — это описание структуры данных, представляющих собой граф. Структуру данных нам определяет условие. Требуется использовать динамическую память, следовательно, использование связных списков — дело решенное и осталось только определить структуру записи связного списка.

Очевидно, что элемент связного списка нужен для описания текущей позиции рубящей шашки, следовательно, необходимо просто посмотреть, что известно про позицию. А известно следующее:

- координаты текущей позиции;
- адреса элементов связного списка, описывающих позиции, в которые рубящая шашка может попасть из текущей.

Достижимых позиций не более трех (позицию, из которой шашка пришла, можно исключить), а координат две. Следовательно, в структуре элемента связного списка должно быть три поля, являющихся указателями на элемент связного списка, и два числовых поля, имеющих смысл координат.

После обсуждения структуры данных можно обдумать, как эта структура данных должна быть построена. Если вы невнимательно разобрали пример с построением двоичного дерева, то посмотрите его еще раз. Его отличия от нашей задачи заключаются в следующем:

- количество указателей в нашей структуре данных также фиксировано, но они не обязаны все указывать на реально существующие адреса;
- ветви в нашей задаче строятся до тех пор, пока это возможно, различные ветви могут иметь различную длину.

Эти отличия существенны, но они не касаются общей структуры программы. Следовательно, общую конструкцию мы можем вполне взять из примера двоичного дерева. Это рекурсивная процедура, получающая на вход координаты текущей позиции и выполняющая следующие действия:

```
Для каждого из 4-х возможных направлений рубки
Если в рассматриваемом направлении рубка возможна, то
  Осуществляем рубку.
  Ищем свободный указатель и от него создаем указатель на новый
  элемент списка, который и будет содержать информацию о новой
  позиции рубящей шашки.
  Вызываем процедуру с координатами нового положения шашки.
```

Далее будем поступать так, как мы уже поступали, а именно зададим себе вопрос, что в нашем алгоритме неопределенного. Сформулировав неопределенности, мы получим информацию для дальнейшего анализа. Неопределенности есть, и достаточно существенные, например:

- не вполне понятно, как определить, возможно данное направление или нет. И что такое направление вообще;
- в чем заключается процедура рубки;
- предполагается, что возможных направлений рубки 4. Это не так. Их только 3, так как с одного из направлений шашка пришла, и его, как направление рубки, рассматривать нельзя. Но если мы желаем одно из направлений исключить, то мы должны придумать, как его отделить от других трех;
- и самое главное — из сказанного ранее совершено не ясно, как мы собираемся обходить построенный граф в поисках самого длинного продолжения, хотя может быть пока это и не важно.

Попробуйте подумать хотя бы над первым вопросом. Даже интуитивно ясно, что в нашей структуре данных нет достаточной информации для ответа

на этот вопрос. Это от того, что была совершена серьезная логическая ошибка. Описав необходимую структуру графа и начав рассуждать об алгоритме (программе), мы ничего не сказали о том, как данный граф будет построен. Повторимся еще раз. Необходимо понять не только то, что граф будет из себя представлять, но и то, как он строится из первичной информации, то есть из той информации, которая дана непосредственно в условии.

Первичная информация — это позиция на доске. Ее можно представить, например, двумерным массивом. Договоримся, что пустая клетка будет представлена нулем. Шашка, осуществляющая рубку, пусть изображается числом 2, а ее противники пусть будут 1.

Теперь можно ответить на поставленные вопросы.

- Направление — это вектор, такой, что если его прибавить к координатам (индексам) элемента двумерного массива доски, то мы получим элемент массива, представляющий клетку, в которую возможно осуществить ход. Таких направлений четыре: (−2, −2), (−2, 2), (2, −2), (2, 2).
- Для того чтобы выяснить допустимость направления, необходимо установить истинность двух утверждений: первое — ячейка массива, полученная прибавлением к координатам (индексам) текущего элемента вектора направления, содержит ноль (клетка доски пуста); второе — ячейка массива, полученная прибавлением половины вектора направления, содержит единицу (клетка доски содержит шашку противника).
- Процедура рубки заключается в двух действиях:
  - изменении координат (индексов) текущего элемента на вектор направления;
  - обнулении элемента, полученного прибавлением половины вектора направления.

Проблема отбрасывания одного из направлений решается двумя способами. Можно запоминать, откуда пришла шашка, для чего в элементе связного списка придется вводить по крайней мере одно дополнительное поле, описывающее, откуда был совершен переход. А усложнение структур данных всегда приводит к усложнению логики. Но можно ничего этого и не делать, если немного подумать о том, как проводится проверка на допустимость направления.

Только что было сказано, что одно из условий допустимости — это наличие единицы в ячейке через "половину вектора направления", а в описании процедуры рубки сказано, что эта единица обнуляется. Это означает, что направление, откуда шашка пришла, при расчете следующего хода будет недопустимым (там после выполнения хода стоит ноль). Следовательно, мы можем проблему допустимости проигнорировать и проверять все четыре направления. Одна из проверок окажется лишней, но в качестве компенсации отпадет необходимость в дополнительных операциях по проверке

допустимости, которые почти наверняка окажутся более сложными, а следовательно, мы получим и еще выигрыш в упрощении логики программы.

Проблема восстановления данных. Переборные алгоритмы, а мы имеем дело именно с перебором, требуют "отката", то есть восстановления данных в некоторой точке перебора с целью построения нового варианта. В нашей ситуации "откат" заключается в восстановлении одной-единственной шашки, срубленной данным ходом. Из смысла процедуры рубки ясно, что срубленная шашка находится ровно посредине между исходной и конечной позицией рубящей шашки (при рубке одной шашки противника). То есть координаты поля, на котором стояла шашка (индексы массива, где находилась единица), можно найти как среднее арифметическое координат исходного и конечного поля. Это в свою очередь, конечно, потребует запоминания информации об исходном поле.

Все, что обсуждалось до сих пор, касается только построения графа (связного списка), а теперь мы подходим к главному, к проблеме поиска наидлиннейшего пути рубки или в наших терминах наидлиннейшей ветви графа или, что то же самое, наидлиннейшей ветви связного списка.

Решение этой задачи сводится к необходимости помечать элементы связного списка, через которые проходит наидлиннейший путь или наидлиннейшие пути. Можно, конечно, создать дополнительную структуру, в которую наидлиннейший путь будет записываться, но, как уже было сказано, дополнительные структуры данных — это дополнительное усложнение логики и, может быть, существенное. Поэтому мы лучше создадим дополнительное поле в элементе связного списка для сохранения дополнительной информации. Это тоже дополнительная структура, но она уже локальная.

Конечно, хотелось бы, чтобы эта дополнительная информация была одним числом, (что проще) и это логично, так как существует число, характеризующее длину маршрута — сама длина. Если в каждом элементе будет записано число, являющееся длиной самого протяженного маршрута рубки, проходящего через данный элемент, то алгоритм поиска станет очень прост:

```
Числовую характеристику начального узла объявить максимальной.
Пока не достигнут тупик, делать
  Проверить все поля ссылок
    Если ссылка не тупиковая, то
      Если элемент, на который указывает ссылка, содержит
      характеристику, равную максимальной, то перейти по этой ссылке.
```

Нетрудно заметить, что данный алгоритм обеспечивает поиск не одного маршрута, а всех существующих. Трудность в понимании, наверное, вызывает первый пункт, в котором числовая характеристика начального узла объявляется максимальной. Не вполне понятно, на каком основании. Для полного понимания необходимо рассмотреть механизм получения такой

характеристики, но даже сразу можно заметить, что наидлиннейший маршрут обязан проходить через начальный элемент, а так как числовая характеристика длины только одна, то она длиннейший маршрут и определяет.

А теперь все-таки о механизме. Значение длины маршрута известно только на последнем поле (последнем элементе связного списка). Следовательно, механизм определения характеристики длины должен заключатся в прописывании этого конечного числа при возврате по маршруту. Пусть теперь шашка находится в некоторой точке А и предстоит принять решение о числовой характеристике для пункта А. Предположим, что из А выходит максимальное количество ненулевых указателей (то есть три), это означает, что характеристику для А необходимо выбирать из трех возможных значений, которые возвращены в А рекурсивными вызовами. Какое из них выбрать? Конечно же, максимальное.

И последнее, когда принимать решение о числовой характеристике для текущего элемента (поля доски)? Конечно же, тогда, когда будут отработаны все допустимые направления рубки. Из этого следует, что формировать числовую характеристику мы можем в процессе построения графа (связного списка). Все существенные технические проблемы обсуждены, можно записать алгоритм. Головной алгоритм, видимо, будет состоять из двух процедур:

- ❒ алгоритм построения связного списка — графа;
- ❒ алгоритм поиска длиннейшего маршрута.

Алгоритм построения связного списка:

```
Ввод данных в виде двумерного массива
Создание начального элемента списка с записью в него исходного положения рубящей шашки
Вызов рекурсивной процедуры построения связного списка
Запись в начальный элемент полученной длины длиннейшего маршрута
```

Рекурсивная процедура построения связного списка:

```
Входные данные: адрес на текущий элемент списка, длина уже пройденного маршрута
Запомнить адрес на текущий элемент
Обработка направления (-2, -2)
  Максимальная длина = возвращенной длине
Восстановить адрес на текущий элемент
Обработка направления (2, -2)
  Длина = возвращенной длине
  Если Длина > Максимальной То Максимальная = Длине
```

```
Восстановить адрес на текущий элемент
Обработка направления (2, 2)
  Длина = возвращенной длине
  Если Длина > Максимальной То Максимальная = Длине
Восстановить адрес на текущий элемент
Обработка направления (-2, 2)
  Длина = возвращенной длине
  Если Длина > Максимальной То Максимальная = Длине
Восстановить адрес на текущий элемент
Записать Максимальную длину, как характеристику длины маршрута, для текущего элемента связного списка.
```

Обработка направления:

```
Проверить допустимость хода в заданном направлении
Если Ход допустим, То
  Ищем свободный (то есть указывающий в никуда) указатель на адрес
  элемента списка
  Создаем новый элемент списка
  Записываем в новый элемент координаты новой позиции рубящей шашки
  Вызываем процедуру построения связного списка с новыми
  фактическими параметрами
  Восстанавливаем позицию
  Возвращаем длину, равную пройденной длине
Иначе
  Возвращаем длину, меньшую пройденной длины на 1 (так как ход
  оказался недопустимым)
```

Алгоритм поиска длиннейшего пути. Данный алгоритм мы уже построили ранее. Сейчас заметим только, что процесс обхода связного списка должен отображаться на экране. Для этого распечатаем массив доски и на каждом шагу после перехода к очередному пункту маршрута в координатах, сохраненных в полях элемента связного списка, будем распечатывать элемент доски, но другим цветом. В программе для выделения используется зеленый цвет. Текст программы приведен в листинге 15.3.

Листинг 15.3

```
program example;
 uses crt,graph;
 type
```

```
  nodes=^zapis;
  zapis=record
         x,y,num:integer;
         uk1,uk2,uk3:nodes;
        end;
 var
  desk:array[1..8,1..8] of integer;
  k,n,i,x,y,l,max:integer;
  list
  :nodes;
function hod(list:nodes;num:integer):integer;
var
 from:nodes;
 n,max:integer;
function move(x,y:integer):integer;
var
 num1:integer;
begin
 if (list^.x+x>0) and (list^.x+x<9) and (list^.y+y>0) and (list^.y+y<9)
  then
    if (desk[list^.x+x,list^.x+x]=0)
          and (desk[list^.x+(x div 2),list^.y+(y div 2)]=1) then
  begin
   desk[list^.x+(x div 2),list^.y+(y div 2)]:=0;
   if list^.uk1=nil then
    begin
     new(list^.uk1);
     list:=list^.uk1;
    end
   else
   if list^.uk2=nil then
    begin
     new(list^.uk2);
     list:=list^.uk2;
    end
   else
    begin
     new(list^.uk3);
     list:=list^.uk3;
    end;
   list^.x:=from^.x+x;list^.y:=from^.y+y;
```

```
   num1:=hod(list,num+1);
   desk[(list^.x+from^.x) div 2,(list^.y+from^.y) div 2]:=1;
   list:=from;
   move:=num1;
  end
  else move:=num-1
 else move:=num-1;
end;
begin
 from:=list;
 max:=move(-2,-2);
 n:=move(2,-2);
 if n>max then max:=n;
 n:=move(2,2);
 if n>max then max:=n;
 n:=move(-2,+2);
 if n>max then max:=n;
 list^.num:=max;
 hod:=max;
end;
procedure search(list:nodes);
var
 i,j,num:integer;
 node:nodes;
procedure print(node:nodes);
var
 mynode:nodes;
begin
 gotoxy(2*node^.x,2*node^.y);write(desk[node^.x,node^.y]);
 if node^.uk1<>nil then
   begin
    mynode:=node^.uk1;
    if mynode^.num=num then print(mynode);
   end;
 if node^.uk2<>nil then
    begin
     mynode:=node^.uk2;
     if mynode^.num=num then print(mynode);
    end;
  if node^.uk3<>nil then
     begin
```

```
      mynode:=node^.uk3;
      if mynode^.num=num then print(mynode);
     end
end;
begin
 textcolor(15);
 for i:=1 to 8 do
  for j:=1 to 8 do
   begin
    gotoxy(i*2,j*2);write(desk[i,j]);
   end;
 textcolor(1);
 num:=list^.num;
 print(list);
end;
begin
 clrscr;
 for i:=1 to 8 do
  for k:=1 to 8 do
    desk[i,k]:=0;
 read(n);
 for i:=1 to n do
  begin
   read(x,y);
   desk[x,y]:=1;
  end;
 read(x,y);
 clrscr;
 desk[x,y]:=2;
 textcolor(15);
 for i:=1 to 8 do
  for k:=1 to 8 do
   begin
    gotoxy(i*2,k*2);write(desk[i,k]);
   end;
 readkey;
 new(list);
 list^.x:=x;list^.y:=y;
 list^.num:=hod(list,1);
 search(list);
 readkey;
end.
```

В заключение. Мы решили достаточно сложную задачу на построение и обход графа. Есть смысл еще раз выделить ключевые моменты логических построений.

- Переборная задача на графе — это перебор всех возможных путей.
- Перебор всех возможных путей осуществляется двумя взаимообратными операциями: путь в глубину и возврат.
- Для того чтобы обеспечить возврат, необходимо обеспечить запоминание данных перед тем, как совершать очередной шаг в глубину.
- Если нас интересует характеристика, значение которой будет известно только в конечной точке, а использование этой характеристики предполагается в начале графа, то есть смысл ее значение при возврате переносить из конца в начало.

Глава 16

Движение в поле сил тяготения

Условие задачи. Несколько тел известной массы с известными векторами начальных скоростей находятся в пустом пространстве (если другие тела и существуют, то их влиянием можно пренебречь), необходимо построить траектории их движения.

Задачи моделирования физических процессов обладают особенной сложностью. Связано это зачастую с ошибочным пониманием того, что требуется получить. Например, очень часто человек, моделирующий что-то, задается вопросом, а что мы должны увидеть, когда модель будет работать? Вопрос совершенно неверный. Если бы мы знали, что должно получиться, то создавать модель не было бы необходимости. Вторая, часто встречающаяся ошибка повторяет первую — программист, моделируя, например, движение, добивается того, что бы была модель движения, но при этом исходит из каких-то соображений, не имеющих отношения к природе рассматриваемого движения. Если дать такому программисту задачу на моделирование движения в поле силы тяжести под воздействием первоначального толчка, он может рассуждать так: тело движется с течением времени, сначала вверх, затем, достигнув максимальной высоты, оно падает вниз. Это все так, но не совсем. Во-первых, так называемая максимальная высота неизвестна в процессе движения, а во-вторых, оно не просто движется вверх и вниз, оно движется по определенному закону, и в этом соль.

Поэтому сейчас, прежде чем идти на борьбу с трудной задачей, потренируемся на более простой. И в качестве простой возьмем модель движения тела в поле силы тяготения под углом к горизонту.

Единственное исходное данное для процесса — это вектор начальной скорости. Мы определим его через скалярную величину скорости и величину угла наклона. Далее воспользуемся красивой математической идеей разбиения движения в плоскости на два независимых движения: движение в вертикальном направлении под воздействием силы тяготения и равномерное

прямолинейное движение по горизонтали. Для того чтобы выполнить такое разбиение, сделаем две вещи: во-первых, вектор скорости разложим на две составляющие: горизонтальную и вертикальную, во-вторых, вертикальную и горизонтальную координаты будем вычислять независимо друг от друга по соответствующим им законам.

Величина, увязывающая два движения в одно целое — это время. Перед началом процесса время равно нулю, затем на каждом шаге цикла оно изменяется на малую величину. С точки зрения физики время в любом физическом процессе течет непрерывно, но мы не умеем моделировать непрерывные процессы, кроме того, если временной интервал взять достаточно маленьким, то его дискретность будет не заметна.

И последнее замечание. Арифметическое выражение `400-round(y)` необходимо для того, чтобы учесть "неправильность" компьютерных координат. Координата *Y* на экране монитора вниз не уменьшается, а растет, вследствие чего компьютерная координата *Y* — есть перевернутая физическая координата *Y*.

Текст созданной программы приведен в листинге 16.1.

Листинг 16.1

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  t,x,y,v,vx,vy,a:real;
begin
 read(a,v);
 dr:=detect;initgraph(dr,md,'');
 a:=Pi*a/180;
 vx:=v*cos(a);vy:=v*sin(a);
 t:=0;
 repeat
  x:=vx*t;
  y:=vy*t-4.9*t*t;
  putpixel(round(x),400-round(y),15);
  t:=t+0.001;
 until y<0;
end.
```

Рассмотренный пример нужен только для того, чтобы показать главную идею компьютерной модели физического процесса — основой модели является математически корректная запись физических законов, а не ожидаемый визуальный результат.

К сожалению, именно это часто представляет существенную сложность, что мы и увидим на нашей задаче о движении тел в поле силы тяготения. Необходимо сказать, что потребности математической модели быстро наталкиваются на ограниченность средств, которыми они реализуются. С этим мы столкнулись даже в рассмотренном простом примере. Физика требует, чтобы время было непрерывным, математика против этого требования физики не возражает, а мы, программисты, его удовлетворить не можем. Но этот конфликт разрешился достаточно легко. От абсолютной точности легко отказаться, а если пожертвовать точностью, то отпадает потребность и в непрерывности.

Второй конфликт, с которым мы столкнемся, — конфликт между действием и взаимодействием. Это уже и сложнее понять и сложнее победить. Закон всемирного тяготения, который лежит в основе модели, — это закон взаимодействия, он описывает, как тела взаимодействуют, то есть первое тело воздействует в некоторый момент времени на второе тело, а второе тело в этот же момент, до того как воздействие первого изменило его состояние, воздействует на первое. Иначе говоря, взаимодействие — это мгновенные одновременные воздействия каждого тела на все остальные. Но компьютер не может выполнять никакие две операции одновременно, он может только выполнять последовательно операцию за операцией.

Это в свою очередь означает, что какое-то тело нам придется выделить и его воздействия рассматривать в первую очередь. За один шаг работы модели такое выделение, наверное, не приведет к заметному искажению, но нельзя гарантировать, что искажения (а они будут) не накопятся очень быстро. Причем даже неизвестно, как быстро они станут заметными. Чтобы оценить скорость накопления погрешности, придется провести серьезную математическую работу.

Может быть, впрочем, погрешности будут и невелики или учесть их будет не слишком сложно, но проблема выглядит принципиальной. Одно из тел получает больше прав, становится более значимым, чем другие, а это уже говорит о том, что мы моделируем не закон всемирного тяготения, а что-то совсем другое. Наверное, уже понятно, чем действие отличается от взаимодействия, но все же приведем еще одно небольшое рассуждение. Пусть дано два тела `ТЕЛО1` и `ТЕЛО2`. Предположим, мы желаем рассчитать результат их взаимодействия в некоторый момент времени, который для удобства назовем нулевым. Сделаем это так:

```
Учтем воздействие ТЕЛА1 на ТЕЛО2, для чего
  Рассчитаем ускорение, приобретаемое ТЕЛОМ2 в результате действия
  силы со стороны ТЕЛА1.
  Рассчитаем перемещение ТЕЛА2 в соответствии в формулой, данной
  физикой для ускоренного движения.
Учтем воздействие ТЕЛА2 на ТЕЛО1 аналогично тому, как это сделано ранее.
```

На первый взгляд все очевидно, но здесь кроются две ошибки.

- Первый пункт расчетов действительно проводится для данного нулевого момента времени, а второй уже для другого. Конечно, можно возразить, что появляющуюся погрешность можно сколь угодно сильно сократить, уменьшая интервал времени между расчетами первого действия и второго, но здесь такое соображение не сработает. Дело в том, что чем меньше интервал, тем большее количество таких интервалов уложится в период наблюдения за движением тел. Получается, что мы уменьшаем значение погрешности, но увеличиваем количество погрешностей. Совершенно не очевидно, что выигрыш от уменьшения абсолютного значения слагаемых погрешностей перевесит проигрыш от увеличения количества слагаемых.
- Вторая ошибка заключается в серьезнейшей логической неувязке. С одной стороны, мы отказываемся от непрерывного расчета взаимодействия и переходим к учету последовательности разовых моментальных воздействий, а с другой стороны, предполагаем использование формулы расчета пути при равноускоренном движении, которая может использоваться только в случае *непрерывного силового воздействия!*

Вся описанная путаница получилась от того, что мы, отказываясь моделировать непрерывное движение, продолжали пользоваться непрерывной математикой. А нам нужно быть последовательными, все законы и все расчеты должны быть дискретными. Уточним модель следующим образом: тела не взаимодействуют непрерывно; имеет место последовательность мгновенных воздействий тел друг на друга, в результате которых возрастают их скорости, а затем в течение некоторого малого времени они движутся равномерно с полученной скоростью до следующего воздействия, в котором скорости опять изменяются.

Новая модель уже лучше, но нам придется скорректировать и ее. Проблема в скорости. В равноускоренном движении скорость вычисляется так: $v = v_0 + (t - t_0) * a$. Из этой формулы следует, что скорость в нашей модели вообще не будет изменяться, так как мгновенное воздействие происходит за нулевое время, то есть $t = t_0$.

Придется во имя принципа дискретности и под флагом борьбы с нарастанием погрешности пойти еще на одну корректировку модели, а именно разделим процесс изменения скорости и процесс изменения координат. Будем полагать теперь, что в течение некоторого малого интервала времени скорость изменяется в соответствии с законами физики, но на равномерное движение тела внутри данного интервала влияния не оказывает, а в последней точке интервала скорость тела скачком меняется на накопленную. Таким образом, внутри каждого расчетного интервала каждое тело, участвующее в движении, движется по закону равномерного и прямолинейного движения, но для каждого интервала формула этого закона своя, ее числовые параметры

определяются результатом изменения скорости на предыдущем расчетном интервале.

Думаю, вы согласитесь, что мы провели достаточно сложные рассуждения. Если бы мы были профессиональными математиками, то внешне все было бы проще. Была бы записана система дифференциальных уравнений, а затем выбран численный метод ее решения. Получение хорошего результата было бы гарантировано всей мощью математического аппарата, но тогда сущность перехода от непрерывности к дискретности оказалась бы скрыта, а это, как вы видите, совсем не тривиальная вещь.

На данный момент, пожалуй, все проблемы решены и можно записать алгоритм работы модели:

```
Пока не завершено наблюдение, делать
  Для каждого тела
    Рассчитать ускорение, получаемое им от воздействия других тел.
    Рассчитать скорость, которое оно получит за небольшой временной
    интервал от ускоренного движения.
  Для каждого тела
    Рассчитать изменения координат полагая, что тело движется
    равномерно и прямолинейно с текущей скоростью.
    Прорисовать участок траектории или прорисовать точку в новом
    положении тела.
```

Текст программы по приведенному алгоритму представлен в листинге 16.2.

Листинг 16.2

```
program example;
 uses crt,graph;
 var
  body:array[1..10] of record
                        x,y:real;
                        vx,vy,m,ax,ay:real;
                       end;
  n,i,j,dr,md:integer;
  t,l:real;
begin
 read(n);
 for i:=1 to n do
  read(body[i].x,body[i].y,body[i].vx,body[i].vy,body[i].m);
 dr:=detect;initgraph(dr,md,'');
 t:=0;
 repeat
```

```
  for i:=1 to n do
   begin
    body[i].ax:=0;
    body[i].ay:=0;
    for j:=1 to n do
     if j<>i then
      begin
       l:=sqrt(sqr(body[i].x-body[j].x)+sqr(body[i].y-body[j].y));
       body[i].ax:=body[i].ax+body[j].m/(l*l*l)*(body[j].x-body[i].x);
       body[i].ay:=body[i].ay+body[j].m/(l*l*l)*(body[j].y-body[i].y);
      end;
    body[i].vx:=body[i].vx+0.001*body[i].ax;
    body[i].vy:=body[i].vy+0.001*body[i].ay;
   end;
  for i:=1 to n do
   begin
    body[i].x:=body[i].x+body[i].vx*0.001;
    body[i].y:=body[i].y+body[i].vy*0.001;
    putpixel(round(body[i].x),round(body[i].y),15);
   end;
 until keypressed;
end.
```

В заключение. Решена достаточно сложная задача, показаны серьезные проблемы и важные подходы к их решению, но это еще не все. Все вопросы физического моделирования, конечно, в небольшой главе и не раскрыть, но совершенно необходимо упомянуть о грамотном визуальном отображении информации. Если, например, строится траектория движения, и она при этом не умещается на экране, говорить о соответствии с физическими законами нет большого смысла, если строится модель процесса, но визуально все происходит так быстро, что глаз не успевает ничего увидеть, то в этом опять же нет смысла. Конечно, не все физические модели подразумевают визуальное представление, как раз наоборот, для физиков часто важнее числа, чем прямо наблюдаемые эффекты, но если визуальное представление все же нужно, то необходимо обдумать его масштабы. А именно масштабы времени и расстояния.

Вернемся к задаче визуального построения траектории. Физика для расчета любых величин пользуется системами единиц. Расстояние измеряется в метрах, время в секундах. Но у экрана монитора своя единица измерения — пиксел. Сколько один пиксел вмещает в себя метров — это условность. Поэтому вы вполне можете для вашего процесса ввести такой масштаб, какой вам удобен. С единицей времени произвола будет еще больше. Фактически

единицей времени является шаг цикла, в котором реализуется моделируемый процесс. Но такая развитая демократия в вопросах масштаба расстояний и времени влечет за собой и серьезные проблемы.

Если вы для вертикальной оси координат и для горизонтальной установите разные масштабы, то искажения реального физического процесса могут стать слишком большими. Далее, представьте, что в вашей программе моделируется два процесса, для каждого созданы свои циклы, в теле которых выполняются некоторые вычисления, и предположим, что вычисления, моделирующие первый процесс, существенно сложнее, чем вычисления, моделирующие второй процесс. Из этого автоматически следует, что скорость течения времени первого процесса самым серьезным образом отличается от скорости течения времени второго процесса, и если при этом моделируемые процессы взаимодействуют друг с другом, например, обмениваются энергией, то модель будет сильно искажена.

Глава 17

Одинокий путник с плохой памятью

Условие задачи. На квадратном поле установлены препятствия произвольной формы и два пункта A и B. Перед путником поставлена задача — найти путь из A в B. Известно, что путник не располагает картой местности, у него очень плохое зрение и очень плохая память, настолько плохая, что он практически не может запомнить приметы пройденного маршрута. Из средств ориентации путник располагает естественным чувством ориентации в пространстве своего тела, то есть он может сказать, где у него левая рука, а где правая, также у него есть прибор, напоминающий компас. Этот прибор всегда в любой точке поля показывает направление на пункт B.

Задача может показаться простой. Действительно рассудим так. У путника есть только две возможности для движения: по пустому пространству и вдоль препятствия. Стратегия его движения в первом случае очевидна. Он идет по компасу (будем это устройство называть компасом). Когда он встретит на своем пути препятствие, его стратегия усложняется не намного. Путник просто идет вдоль препятствия до тех пор, пока не обойдет его, а обойдя, опять движется по пустому полю.

Рассуждение вроде бы верное, за исключением одного — неясно, откуда он узнает, что препятствие пройдено. Этот факт легко установить только в том случае, когда препятствие представляет собой тонкую стенку. В примере, приведенном далее, это очень даже непросто (рис. 17.1). Представьте себе, что путник попал в такой лабиринт, и подумайте, откуда, не имея карты, он может взять информацию, что лабиринт пройден.

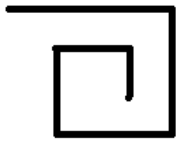

Рис. 17.1

Можно попробовать выйти из положения, используя следующую стратегию: движемся вдоль стенки препятствия, держась за стенку правой (левой) рукой, до тех пор пока не появится возможность движения в сторону пункта B. Как только появляется возможность такого движения отрываемся от стенки и движемся по компасу.

Идея неплохая. Заметьте, здесь используется способность путника распознавать, где право и где лево. Это хорошая примета. В решении корректно поставленных задач должны использоваться все условия, это знают те, кто имеет опыт решения математических или физических задач. Но, несмотря на хорошую примету, идея ошибочна. Ее ошибочность показывает рис. 17.2.

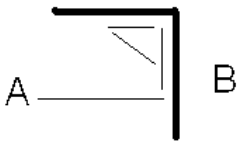

Рис. 17.2

Путник начинает движение от A к B. Встречает препятствие. Пользуясь правилом правой руки, он начинает движение вдоль вертикальной стенки и движется, пока не встретит горизонтальную стенку. Пока возможности двигаться по компасу нет, вертикальная стенка этого не позволяет. Далее он начинает движение вдоль горизонтальной стенки и, пройдя совсем немного, обнаруживает, что движение в направлении B возможно, и начинает движение по компасу. Но, пройдя совсем немного, путник опять упирается в вертикальную стенку, более того, он упирается в одну из тех точек, в которой он уже был, и далее начинает двигаться по кругу.

Два примера, рассмотренные ранее, были нужны для того, чтобы показать сложность задачи, необходимость тщательного анализа.

Еще один важный вопрос связан с проблемой существования решения. Это любимая проблема фундаментальной математики. Программисты в основном "прикладники", и если нам дана задача, то мы априори полагаем, что решение она имеет. Но это совсем неочевидно, огромное количество задач вообще алгоритмически неразрешимо, а условие исследуемой задачи выглядит весьма сложно, и было бы полезно подумать, а для какого типа препятствий можно найти решение гарантированно.

Решение задачи существования может оказаться весьма сложным, тем более, что мы пока даже не решили, как провести классификацию препятствий и что вообще такое "тип препятствия". Но мы сейчас поступим, как самые обычные "прикладники", полагая, что алгоритмическое решение есть, найдем его, а затем, глядя на полученный алгоритм, попробуем вывести условия, для которых он работает. Это вполне разумный подход. Действи-

тельно не совсем ясно, как можно искать условия работоспособности алгоритма, не имея алгоритма.

Кое-что про будущий алгоритм уже известно. Очевидно, что единственная разумная стратегия поведения путника на пустом пространстве — это движение по компасу. Других причин для выбора направления у него на пустом пространстве просто нет. Единственный способ движения при обнаружении препятствия — это, конечно, движение вдоль стенок, как уже предлагалось. Использование правила левой или правой руки также вполне целесообразно. Действительно, пользуясь одним из этих правил, можно гарантированно обойти препятствие и вернуться в ту точку, с которой движение вдоль препятствия началось, а следовательно, и та точка, в которой надо продолжить движение по компасу, также будет достигнута. Вопрос остается только в том, как определить точку отрыва от стенки?

Прежде всего, зададим себе вопрос, на основании какой информации, мы можем сделать заключение о такой точке. У этого вопроса есть две формы. Можно спросить, что необходимо знать для определения точки отрыва, а можно спросить, как использовать то, что уже известно. Ответ на любой из этих вопросов даст решение, но вопрос в первой форме выглядит как-то неопределенно. Может получиться так, что мы придумаем необходимые условия, а их в наличии не окажется. Все-таки более реально исходить из того, что есть.

Что же мы знаем о движении вдоль препятствия? Путь вдоль стены информации в себе не несет никакой. Путник не может определить положение этой стенки в пространстве. Он, наверное, может определить длину пройденного участка в шагах, но это вряд ли ему поможет. Еще одно важное событие, с которым он сталкивается, это поворот. Типов поворота может быть два. Они показаны на рис. 17.3.

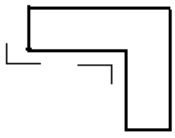

Рис. 17.3

Различие между этими двумя поворотами в типе угла. Для одного из них угол вогнутый, для другого выпуклый. Мы так их и назовем: выпуклый и вогнутый повороты. Можно назвать их еще левым поворотом и правым, но первое название нагляднее. Теперь путник может собрать содержательную информацию, а именно, двигаясь вдоль препятствия, он может считать выпуклые и вогнутые повороты. Ясно, что последовательность этих поворотов как-то характеризует форму препятствия, а знание формы как раз и требуется для определения точки отрыва.

А сейчас обратите внимание на следующий интересный факт. Ранее были приведены примеры проблемных препятствий. Их особенность в том, что они имеют вогнутые углы, для обхода которых необходимы вогнутые повороты. Рассмотрим полностью выпуклое препятствие такое, как на рис. 17.4.

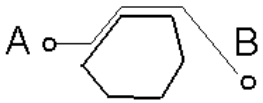

Рис. 17.4

К этому препятствию вполне применим простейший алгоритм отрыва, предложенный в самом начале анализа. Действительно, возможно идти вдоль стенок, пока не появится свободное пространство в направлении, показываемом компасом, после чего можно отойти от стенки и пойти к B. Действительно проблемы начинаются с появлением вогнутых углов.

Проблема озвучена, попробуем применить к ней новый исследовательский прием. Предположим, есть хорошая структура данных. Хорошая в том смысле, что мы умеем с ней работать. В нашей задаче такой хорошей структурой будет полностью выпуклая фигура. И есть плохие структуры, с которыми мы пока не знаем, как работать, в нашей задаче плохая структура — это фигура с вогнутыми углами. Новый прием заключается в следующих действиях:

1. Построим пример хорошей структуры.
2. Найдем минимальное изменение, такое, что оно превращает хорошую структуру в плохую.
3. Посмотрим, что изменилось и как эти изменения можно компенсировать.

Применим описанный прием к нашей задаче.

Шаг первый. Очень хорошая ситуация (рис. 17.5).

Шаг второй. Добавляем к квадрату только одну вогнутость (рис. 17.6).

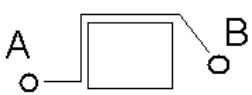

Рис. 17.5

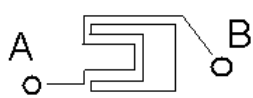

Рис. 17.6

Что изменилось в траектории движения одинокого путника? Ответ очевиден. Если его траектория, как уже говорилось, характеризуется последова-

тельностью выпуклых и вогнутых углов, то можно заметить, что в данной траектории появилось два вогнутых и два выпуклых угла. Прием дал продуктивную идею: если на препятствии появляется некоторое количество вогнутых углов, то они компенсируются таким же количеством выпуклых. Мы больше не будем приводить примеров, а вы можете попробовать проверить это утверждение на более сложных фигурах.

Теперь можно сформулировать и стратегию поведения путника. Путник движется, исходя из того, что препятствие перед ним хорошее, то есть полностью выпуклое. Из этого соображения путник после каждого выпуклого поворота пытается начать движение по компасу, если это не получается, то он движется вдоль стенки до следующего выпуклого поворота. Если же ему встречается некоторое количество вогнутых поворотов, то он не делает никаких попыток движения по компасу, пока не пройдет столько же выпуклых поворотов, сколько он прошел вогнутых.

Мы создали работающую стратегию, опираясь на специальный прием построения препятствий, и при этом утверждаем, что наша стратегия будет работать для любого типа препятствий. Это, конечно, не очевидно, для полной уверенности нужно еще доказать следующее утверждение: *любое цельное препятствие можно построить из прямоугольника последовательными вырезами из него прямоугольников*, но справедливость этого утверждения лежит на поверхности и тратить время на его доказательство не будем.

Еще одна фундаментальная проблема — необходимо выяснить, в каких ситуациях можно найти путь из A в B, а в каких нельзя. Этим надо бы заняться сейчас, но мы поступим наоборот. Интуиция подсказывает, что ситуаций, в которых можно найти путь, очень много. Поэтому проблему существования отложим на потом. А сейчас займемся техническими деталями.

Главный алгоритм оформим в виде цикла, работающего до тех пор, пока не достигнут пункт B. Тело цикла составим из двух процедур: путь по компасу и обход препятствия. Естественное условие завершения пути по компасу — это встреча препятствия, а условие завершения обхода препятствия — это отход от стенки. Оба условия можно проверять внутри самих процедур, это даже удобно, тогда текст главного алгоритма очень короток:

```
Пока не достигнут пункт назначения, делать
  Выполнить процедуру движения по компасу.
  Выполнить процедуру обхода препятствия.
```

Процедура движения по компасу заключается в прорисовке линии в направлении пункта назначения. Для этого в начале процедуры рассчитывается вектор, начальная точка которого — это текущая точка путника, а конечная — пункт назначения. Затем полученный вектор преобразовывается в единичный, и далее шаг движения путника — это смещение его на величину полученного единичного вектора до тех пор, пока не достигнут пункт назначения или не встретится препятствие.

Процедура обхода препятствия технически существенно сложнее. Ключевая сложность в моделировании пространственной ориентировки путника. Не вполне ясно, как обозначить, где у него левая рука, а где правая. В процессе обхода путник выполняет различные повороты, в результате которых меняется его ориентировка, и это надо как-то учитывать. Договоримся путника считать точкой на экране монитора с координатами x, y. Тогда его соприкосновение со стенкой препятствия возможно точками, имеющими следующие координаты: (x+1, y), (x, y+1), (x−1, y), (x, y−1). Если эти четыре точки пронумеровать произвольным образом, то номер точки, которой путник соприкоснулся с препятствием, можно считать числовым обозначением его ориентации. Расчет такой исходной ориентации в программе ведет функция `stenka`. Затем в процессе обхода, каждый раз, когда путник совершает поворот, номер его ориентации в пространстве пересчитывается в соответствии с тем, какой поворот он совершает.

Движение вдоль препятствия — это последовательность шагов, но шагов не одинаковых. Существует несколько ситуаций.

- В направлении точки соприкосновения путника со стенкой нет больше препятствия. В этом случае путник совершает выпуклый поворот и продолжает движение, соприкасаясь со стенкой уже другой точкой, номер которой перевычисляется. В программе такое движение реализуется процедурой `move1`.
- В направлении касания со стенкой препятствия двигаться по-прежнему нельзя, тогда путник продолжает движение вдоль стены в соответствии со своей ориентацией. Такой тип движения в программе осуществляется процедурой `move2`.
- Путник зашел в угол и должен совершить поворот. Поворот сводится к смене ориентации или, что то же самое, к смене точки соприкосновения с препятствием. Наш путник совершает повороты против часовой стрелки, точки касания пронумерованы также против часовой стрелки, поэтому смена номера точки сводится к его увеличению, если он меньше 4 (максимальный номер). Если же текущий номер = 4, то он становится равен 1.

В первых двух случаях соответствующим образом изменяются координаты путника, в третьем изменяется ориентация, то есть перевычисляется номер точки соприкосновения путника со стенкой препятствия.

Для определения типа новой ситуации в программе написана специальная функция `situation`, использующая как текущее положение одинокого путника, так и номер текущей ситуации.

Еще раз напомним, что ключевой момент для обработки любого движения — это ориентация путника, то есть номер точки, которой он в данный момент касается стенки препятствия. Все операции движения жестко при-

вязаны к этой точке, поэтому они организованы в виде выбора из четырех альтернатив.

Полный текст программы приведен в листинге 17.1.

Листинг 17.1

```
program example;
  uses crt,graph;
  const
    n=35;
    ax=10;ay=10;
    bx=230;by=430;
  var
   dr,md,i,x,y,dx,dy,r:integer;
procedure move;
 var
  l,dx,dy,x1,y1:real;
  q:boolean;
begin
  x1:=0;
  y1:=0;
  q:=false;
  l:=sqrt(sqr(bx-x+0.0)+sqr(by-y+0.0));
  repeat
   dx:=(bx-x)/l;dy:=(by-y)/l;
   x1:=x1+dx;y1:=y1+dy;
   if abs(x1)>=1 then
    if getpixel(x+round(x1),y)=0 then
      begin
       x:=x+round(x1);
       putpixel(x,y,15);delay(25);putpixel(x,y,0);
       x1:=0;
      end
    else q:=true;
   if abs(y1)>=1 then
    if getpixel(x,y+round(y1))=0 then
      begin
        y:=y+round(y1);
        putpixel(x,y,15);delay(25);putpixel(x,y,0);
        y1:=0;
      end
    else q:=true;
  until q
end;
```

```
procedure soround;
 var
  k,n,p:integer;
function stenka:integer;
begin
 if getpixel(x,y+1)<>0 then stenka:=1
   else if getpixel(x+1,y)<>0 then stenka:=2
     else if getpixel(x,y-1)<>0 then stenka:=3
       else stenka:=4;
end;
function situation(n:integer):integer;
begin
 case n of
  1: if getpixel(x,y+1)=0 then situation:=1
      else if getpixel(x+1,y)=0 then situation:=2
       else situation:=3;
  2: if getpixel(x+1,y)=0 then situation:=1
      else if getpixel(x,y-1)=0 then situation:=2
       else situation:=3;
  3: if getpixel(x,y-1)=0 then situation:=1
      else if getpixel(x-1,y)=0 then situation:=2
       else situation:=3;
  4: if getpixel(x-1,y)=0 then situation:=1
      else if getpixel(x,y+1)=0 then situation:=2
       else situation:=3;
 end;
end;
procedure move1(n:integer);
{to N}
begin
 putpixel(x,y,0);
 case n of
  1: y:=y+1;
  2: x:=x+1;
  3: y:=y-1;
  4: x:=x-1;
 end;
 delay(25);putpixel(x,y,15);
end;
procedure move2(n:integer);
{to n+1}
begin
 putpixel(x,y,0);
```

```
 case n of
  1: x:=x+1;
  2: y:=y-1;
  3: x:=x-1;
  4: y:=y+1;
 end;
 putpixel(x,y,15);delay(25);
end;
function turn90(n:integer):integer;
begin
 if n<4 then turn90:=n+1 else turn90:=1;
end;
begin
  n:=stenka;
  p:=1;
  repeat
   k:=situation(n);
   case k of
    1: begin
        move1(n);
        p:=p-1;
        n:=stenka;
       end;
    2: move2(n);
    3: begin
        n:=turn90(n);
        p:=p+1;
       end;
   end;
  until p=0;
  delay(25);putpixel(x,y,0);
end;
begin
 randomize;
 dr:=detect;initgraph(dr,md,'');
 for i:=1 to n do
  if i<>15 then
   begin
    setcolor(i);
    setfillstyle(1,i);
    x:=50+random(400);
    y:=50+random(300);
    dx:=2+random(100);
```

```
    dy:=2+random(100);
    r:=5+random(30);
    rectangle(x,y,x+dx,y+dy);
    floodfill(x+dx div 2,y+dy div 2,i);
{    circle(x,y,round(r));floodfill(x,y,i);}
   end;
 {setcolor(2);
 rectangle(10,80,400,85);
 setfillstyle(1,2);floodfill(12,82,2);}
 setcolor(15);
 setfillstyle(1,15);
 circle(bx,by,2);floodfill(bx,by,15);
 x:=ax;y:=ay;
 repeat
   move;
   soround;
 until (abs(x-bx)<3)and(abs(y-by)<3);
 delay(1000);
end.
```

Мы потратили достаточно много времени на разработку алгоритма и программы, полученное решение выглядит красиво, и если окажется, что оно применимо к небольшому количеству задач, работа будет напрасной. Поэтому сейчас проделаем небольшую, но важную работу по выявлению условий, при которых одинокий путник, пользуясь разработанным алгоритмом, сможет найти путь из точки A в точку B.

Критерий будем искать, анализируя алгоритм движения путника. Обратим внимание на два утверждения:

- любое цельное препятствие можно построить из большого прямоугольника, последовательно вырезая из него меньшие прямоугольники;
- путник сможет корректно обойти любое препятствие, построенное по только что сформулированному правилу, пользуясь созданным алгоритмом.

Эти два утверждения позволяют нам сформулировать очень серьезную **теорему**: если вся система препятствий, которую путнику предстоит преодолеть, находится внутри прямоугольника и путник находится за его пределами, то, пользуясь построенным алгоритмом, путник сможет преодолеть эту систему препятствий.

Теорема выглядит правдоподобной, но не очевидной. Попробуем ее хотя бы нестрого доказать.

Доказательство. Если система препятствий представляет собой цельное препятствие, то теорема очевидна из сказанного ранее. Предположим теперь, что система состоит из нескольких несвязанных между собой препятствий. Для доказательства достаточно показать, что в процессе движения путник никогда не приблизится к той стенке прямоугольника, охватывающего систему препятствий, с которой он начал свой путь. Очевидно, что если путник никогда не приблизится, то, следовательно, он будет удаляться и, следовательно, рано или поздно должен достичь дальней стенки охватывающего прямоугольника.

Весь маршрут путника внутри охватывающего прямоугольника состоит из отрезков, вдоль которых он движется по компасу, и в ходе такого движения он, очевидно, приближается к дальней стенке, и из ломаных линий обхода препятствий. Следовательно, необходимо показать, что в ходе движения по линии обхода (вдоль стенок) он также приближается к удаленной стенке охватывающего прямоугольника.

Рассмотрим текущее препятствие, с которым столкнулся путник. Для него существует охватывающий прямоугольник, ориентированный так же, как охватывающий прямоугольник для всей системы препятствий. Это означает, что путник сталкивается со стенкой, дальней по отношению к удаленной стенке охватывающего прямоугольника, и это же означает, что он пойдет в отрыв от удаленной стенки текущего охватывающего прямоугольника, а это в свою очередь и означает, что, двигаясь по ломанной линии, он также приближается к точке назначения.

Рассуждения, записанные ранее, нельзя считать строгим доказательством, скорее это более менее убедительное обоснование, но если к этому обоснованию добавить некоторое количество сложных тестовых примеров, то можно быть уверенным, что полученное решение если и не охватывает все возможные ситуации, то во всяком случае область его применения велика.

В заключение. Очень и очень часто неопытные программисты опускают обоснование своего алгоритма. Необходимость доказательства игнорируется по причинам, далеким от логики. Разработчик, придумавший красивый алгоритм, как правило, не готов согласиться с возможностью ошибки. Действительно, красивый алгоритм должен быть верен, особенно если на него потрачено много сил. В иное не хочется верить, в этом все дело, но такие рассуждения далеки от логики. Сколько бы разработчик не потратил времени и сил, его работа вполне может оказаться ошибочной. Это нужно помнить, и если алгоритм сложен, его дополнительное обоснование если не доказательство, никогда не будет лишним.

Глава 18

Метод минимакса

Минимакс — это метод принятия решений, используемый тогда, когда выбирать приходится из множества вариантов, устроенного в виде дерева. Так, например, устроено множество вариантов шахматной или шашечной партии. Ситуация выбора такого типа встречается достаточно часто, но в наиболее чистом виде она представлена все же в играх. Представьте себе любую игру, в которую умеете играть, и любую игровую ситуацию. Предположим, что продолжить эту ситуацию можно некоторым количеством вариантов. Каждый закачивается положением, которое можно оценить с точки зрения того, насколько оно хорошо для принимающего решение. Как именно будет получена такая оценка, сейчас не важно, договоримся лишь о том, что оценка — это число.

Оценки конечных позиций всех возможных вариантов — это единственная информация, которую можно использовать для определения желательного варианта (другой информации просто нет). Метод минимакса как раз и описывает, как, используя информацию об оценках конечных позиций, принимать решения о наилучшем ходе.

Для дальнейших рассуждений придумаем гипотетическую игру. Смысл этой игры не имеет значения, договоримся лишь, что из каждой позиции возможно только два варианта продолжения. Будем такую игру называть двоичной.

Метод анализа строится из тех соображений, что оба противника играют наилучшим образом. Назовем противников *Первым* и *Вторым*. Пусть оценочная функция строится для *Первого* (как именно она будет выглядеть, сейчас неважно). Тогда из нашего предположения следует, что *Первый* стремится к наилучшей оценке, но *Второй* оставляет ему только наихудшие, и следовательно, реально *Первый* может рассчитывать только на лучшую из худших оценок.

Чтобы понять, как это, построим дерево вариантов для двоичной игры. Как уже было сказано, в этой игре из каждой позиции возможно только два продолжения. Построим для нее дерево вариантов (рис. 18.1).

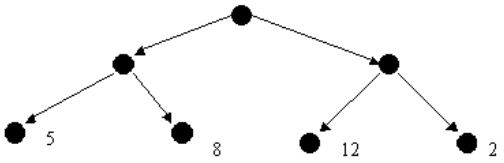

Рис. 18.1

В нижних точках проставлены значения оценочной функции. Первый игрок должен решить, выбрать ли ему продолжение влево или продолжение вправо.

Может показаться, что *Первый* должен ходить вправо, так как это направление обещает позицию с максимально большой оценкой 12. Но дело в том, что он не сразу попадает в кружок окончательной позиции. Вначале игра переходит в позицию, в которой решение принимает *Второй*, и он естественно выберет вариант с оценкой 2, в результате *Первый* вместо ожидаемых 12 очков получит только два. В варианте левой ветви, *Второй* даст *Первому* вариант с 5 очками (это наихудшая). Таким образом, на наилучшие оценки 8 и 12 *Первый* может не рассчитывать, его реальный выбор из оценок 5 и 2. 5 лучшая, и, следовательно, *Первому* предпочтительнее ход влево. Поэтому метод и называется *методом минимакса*.

Опишем процесс поиска минимаксной оценки. Для этого назовем альтернативами позиции (узлы дерева вариантов), имеющие одного и того же непосредственного предка (то есть узел уровнем выше). Будем осуществлять подъем по дереву вариантов с самого нижнего уровня. На каждом шаге подъема для каждой группы альтернатив выполним следующие операции:

- определим наибольшую и наименьшую оценку;
- если в узле предка решение принимал первый игрок (для которого рассчитывается минимаксная оценка), то оценка узла предка равна максимальной оценке, полученной из альтернатив, иначе оценка узла предка равна минимальной.

Продемонстрируем эту процедуру на более сложном примере (рис. 18.2).

На первом шаге снизу решение принимает второй игрок — и естественно, он выбирает тот вариант, при котором оценка будет минимальной. На втором шаге выбор за первым игроком, и он, естественно, выбирает из оценок, полученных на предыдущем уровне, наилучшие (это оценки 8, 3, 5, 5). На следующем шаге опять решение принимает второй игрок, и он опять оставит из полученных на предыдущем шаге наихудшие, то есть 3 и 5. И наконец, последний шаг подъема — решение принимает опять первый игрок, он выбирает наилучшую оценку 5. Таким образом, из полученной минимаксной оценки можно сделать вывод, что для первого игрока более предпочтительным является вариант с ходом вправо, с оценкой 5.

Разобравшись с постановкой задачи, можно переходить к ее решению. Решение разделим на две независимые части. Во-первых, необходимо построить двоичное дерево и завершающие узлы заполнить оценками (в качестве оце-

нок будем брать случайные числа). Во-вторых, нужно организовать собственно минимакс.

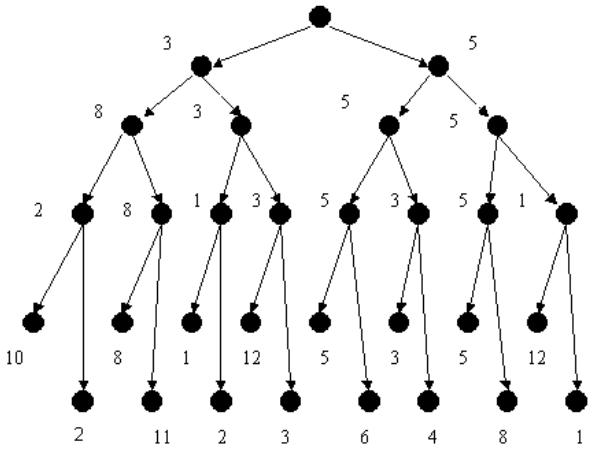

Рис. 18.2

Функция, в которой реализуется минимакс, отличается от процедуры создания дерева наличием возвращаемого значения. Механизм возврата работает различными способами, для последнего узла и для промежуточных. В последнем узле в качестве возвращаемого значения, конечно, будет значение оценки, других возможностей просто нет. Функция, обрабатывающая промежуточный узел, вызывает две функции для обработки дочерних узлов и соответственно получает в качестве вариантов два значения, из которых она должна сделать выбор. Выбор же зависит от того, какого из игроков представляет данный вызов функции. То есть какой из игроков на данном уровне дерева вариантов принимает решение. Первый игрок, для которого проводится анализ, принимает решение на нечетных уровнях, второй — на четных. Поэтому какое значение передается выше (большее или меньшее), определяется четностью номера вызова.

Текст программы приведен в листинге 18.1.

Листинг 18.1

```
program example;
 uses crt;
 type
  node=^zapis;
  zapis=record
             ocenka:integer;
             left,right:node;
            end;
```

```
 var
  uk:node;
procedure Create_tree(uk:node;n:integer);
begin
 if n<5 then
  begin
   new(uk^.left);
   Create_tree(uk^.left,n+1);
   new(uk^.right);
   Create_tree(uk^.right,n+1);
  end
 else uk^.ocenka:=random(20);
end;
function Minimax(uk:node;n,x:integer):integer;
 var
  left,right,k:integer;
begin
 if n<5 then
  begin
   left:=Minimax(uk^.left,n+1,x-(20 div n+1)+n);
   right:=Minimax(uk^.right,n+1,x+(20 div n)-n);
   if n mod 2=0 then
    if left<right then k:=left else k:=right
   else
    if left<right then k:=right else k:=left;
  end
else k:=uk^.ocenka;
gotoxy(x,n*4);write(k);
Minimax:=k;
end;
begin
 randomize;
 clrscr;
 new(uk);
 Create_tree(uk,1);
 Minimax(uk,1,40);
end.
```

В заключение. Решенная задача не очень сложна. Но относительно простой она получилась только потому, что удалось использовать уже имеющиеся результаты. А именно мы использовали процедуру обхода древовидной структуры данных. Кстати говоря, эту процедуру вполне можно принять в качестве стандартного средства построения и обхода различных деревьев. Поэтому если вам вдруг встретится задача, связанная с обработкой древовидных структур данных, вы можете считать ее уже частично решенной.

Глава 19

Экономный обход графа

Условие задачи. Дан ориентированный граф. Необходимо совершить полный его обход, с использованием минимальных дополнительных данных.

Задача полного обхода графа достаточно понятна. От уже рассмотренных задач на обход деревьев (например, предыдущая задача) она отличается тем, что в произвольном графе возможны циклы, что делает невозможным движение только вперед.

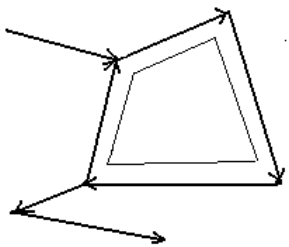

Рис. 19.1

На рис. 19.1 показан граф с одним циклом. Жирные линии изображают ребра графа, а более тонкие его цикл. Двигаясь только вдоль стрелок, легко начать бесконечное движение вдоль единственного цикла. Это, впрочем, чисто техническая проблема. Достаточно, придя в очередной узел, выяснить, выходит ли из него хотя бы один неиспользованный путь. Если да, то идти по нему, а если нет, то вернуться назад. Однако с необходимостью возврата снова возникает проблема. Посмотрим на рис. 19.2.

Пусть ребра слева — это входящие, а ребра справа — исходящие. Тогда для узла есть три пути, по которым в него можно попасть, и если вдруг случится вернуться в этот узел, то встанет вопрос — а по какому ребру возвращаться?

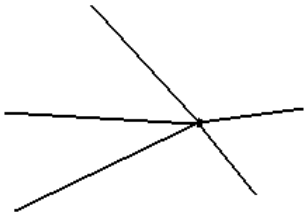

Рис. 19.2

Очевидно, необходимо выполнять возврат по тому же ребру, по которому пришли в узел последний раз. Возникшую проблему легко решить, если создать специальную структуру данных (для каждого узла), в которой каждый раз при входе в узел сохранялась бы информация об узле, из которого пришли в данный. Ясно, что это должен быть массив, ведь если в узел входит десять ребер, то вход будет осуществлен десять раз, все десять входов из разных узлов и все они должны быть запомнены. Таким образом, приходим к необходимости создать для каждого узла дополнительную структуру данных, которая будет весить почти столько же, сколько и основная информация о связях узла. Плохо ли это? И если плохо, то насколько? Чтобы оценить наши потери, рассмотрим следующую достаточно обычную задачу.

Дана сеть населенных пунктов, связанных между собой различными дорогами (сухопутными, воздушными, водными). Необходимо оценить стоимость передвижения из пункта A в пункт B. Пусть всего населенных пунктов 100 и каждый соединен с 10 другими. Это не слишком много. Для организации связей пронумеруем их в произвольном порядке, а связь между пунктом и соседями пометим однобайтными номерами соседей. Арифметика говорит, что дополнительная информация для всего графа потребует тысячи байт. Вроде бы немного, но для тысячи пунктов это около 20 Кбайт (т. к. однобайтными номерами уже не обойтись). Объем дополнительной структуры данных стремительно растет!

Можно, конечно, пока не волноваться, в абсолютных величинах объем дополнительной информации не велик. Но абсолютные величины здесь не самое главное. Главный фактор — это равнозначность объема основной информации и дополнительной. Если появится задача, в которой объем информации о связях будет исчисляться десятками мегабайт, а в этом нет ничего невероятного, то она станет почти не решаемой, так как дополнительная информация также потребует десятки мегабайт. Оперативной памяти не хватит, а использование файлов резко затормозит работу программы. Если вы не верите, что могут быть такие задачи, то вот пример.

Известно, что каждая книга, несущая знания (научная, техническая и т. д.), содержит ссылки на первоисточники информации и таких ссылок может быть сотни. Предположим, что была создана электронная библиотека, в ко-

торой количество книг исчисляется десятками миллионов наименований (в этом нет ничего невероятного, человечество накопило огромные знания), и создатели такой библиотеки решили предоставить своим пользователям возможность ссылочного поиска. Как только такая задача будет поставлена, тут же и встанет озвученная проблема.

Задач такого рода можно придумать достаточно много. Впрочем, проблема есть и для менее масштабных задач. Предположим, что та задача, которую предстоит решать именно вам, не требует десятков мегабайт. Это не означает, что вы не столкнетесь с нехваткой памяти. Возможно, придется считаться с необходимостью делиться ресурсами с другими задачами, работающими в данной вычислительной системе. В общем, сейчас, наверное, уже понятно, почему в условии поставлено требование о минимальных дополнительных структурах данных.

Заметим, что рассуждения, приведенные ранее, достаточно основательны. Необходимость дополнительного массива адресов ссылок почти логически выведена, и, наверное, сейчас трудно представить, что без этого массива можно обойтись, но все же попробуем.

Замечание

Главная решаемая в книге задача — минимизировать интуицию и максимизировать логику. Конечно, полностью логически вывести решение из условия невозможно, но сражаться творческими прорывами с каждой технической деталью вряд ли разумно. Поэтому мы все же стремимся искать логические основания, исходя из которых можно рассуждать в направлении решения. Кроме того, интуитивный прорыв всегда субъективен, он опирается на наши личные знания и способности, которым не всегда можно доверять. *Логика всегда объективна, логика не зависит от личных качеств.*

Последнее предложение специально выделено. Оно показывает место, в котором наше мышление легко попадает в ловушку. На самом деле логика тоже субъективна и тоже опирается на личные знания и убеждения в той же мере, в какой и на объективные факты. Можно смело утверждать, что любой логический вывод — это в какой-то мере вывод из фактов, а в какой-то мере вывод из наших субъективных знаний.

Именно в такую ловушку мы и попали, выведя необходимость дополнительного массива. На самом деле есть железная необходимость запоминать адрес узла источника (далее так будем называть узел, из которого пришли в текущий), а необходимость использовать для этого массив не следует ниоткуда, кроме как из привычки использовать массивы для подобных целей. Логическая ошибка в рассуждениях обнаружена, можно двигаться далее.

Наша цель — придумать, где хранить дополнительные данные. Если не придумывать новых структур для каждого узла, то можно подумать о дополнительной структуре для всего графа, так сказать дополнительный граф.

Однако сама идея дополнительного графа не обещает возможности сокращения потребности в памяти, скорее всего новый граф должен будет в какой-то степени повторять структуру уже существующего, и, весьма вероятно, мы получим не сокращение, а увеличение потребности в памяти.

Если же не создавать ничего нового, а попробовать обойтись тем, что есть, то единственной возможностью для хранения дополнительных данных остается структура, используемая для хранения основных данных, речь идет о структуре, содержащей адреса связей.

Естественно, что использовать для дополнительных данных можно только то, что уже не будет применяться для хранения основных. Это хорошая зацепка. Действительно, путь вглубь по графу не повторяется. Иначе говоря, если некоторый узел C был источником для некоторого другого узла D, то как бы ни осуществлялось дальнейшее движение, узел C уже никогда не будет источником для D. Отсюда следует, что данное, предназначенное для хранения связи C → D, используется только один раз.

После сказанного идея становится совершенно прозрачной. Для хранения адресов возврата можно использовать адреса ссылок. А именно в каждую ссылку после ее использования возможно записать адрес узла источника.

Остальное дело техники, но техники достаточно сложной, поэтому обсудим подробности, запишем алгоритм и снабдим программу необходимым количеством ремарок.

Главная техническая деталь — это, конечно, структура данных. Естественная структура для построения большого графа — это дерево, созданное с помощью связных списков, но мы воспользуемся обычными массивами, чтобы не загружать программу проблемами создания структуры данных, в программе и так достаточно сложная логика. Напомним, наш граф может иметь циклы, поэтому уже отработанные ранее процедуры создания деревьев здесь работать не будут. Если появится интерес, вы можете попытаться реализовать разработанную логику на динамической структуре данных.

Мы же договоримся о структуре данных, состоящей из следующих полей:

- числовое поле, являющееся содержательным значением узла графа (содержательные данные, привязанные к узлу, могут быть сколь угодно сложными, но для демонстрации логики достаточно самого простого поля — числа);
- числовой массив, содержащий номера узлов, с которыми связан данный.

Организовать такой граф несложно. Достаточно ввести количество узлов, затем внутри цикла по параметру для каждого узла ввести содержательное значение и номера узлов с ним связанных. Естественно, для этого номера узлы графа должны быть каким-то способом (совершенно любым) пронумерованы.

Техническая проблема, связанная с сохранением дополнительной информации в массиве ссылок, выражается двумя вопросами.

- На основании чего принимается решение о возврате в узел-источник?
- Как узнать, в каком элементе массива ссылок хранится нужный адрес возврата?

Не думайте, что ответ на первый вопрос уже есть. Да, понятно, что возврат осуществляется тогда, когда пути вперед уже нет. Но это смысловой ответ. Мы же сейчас решаем проблемы технического характера и, следовательно, должны все ответы давать в терминах используемых структур данных.

Чтобы ответить на поставленные вопросы, необходимо договориться о каких-то правилах обхода графа. Выбор этих правил — дело вкуса, но они должны быть. Пусть наше правило будет таким: придя в узел, выбираем ссылку, стоящую в массиве ссылок следующей за той, которой воспользовались в предыдущее посещение узла.

Для того чтобы учесть, какое ребро будет следующим, введем в описание узла еще одну дополнительную переменную — счетчик, роль которого заключается в подсчете количества вхождений в данный узел. Далее все очень просто. Значение счетчика в момент вхождения в узел — это количество предыдущих вхождений. Увеличивая счетчик при каждом вхождении на 1, мы обеспечиваем соответствие количества вхождений значению счетчика. И теперь значение счетчика можно использовать как номер очередной ссылки, по которой возможно идти вглубь графа.

Счетчик позволяет достаточно легко сохранять информацию и о том, где находится нужная ссылка на узел возврата. Ссылки возврата отсчитываются в обратном порядке по отношению к ссылкам движения в глубину. Если ссылки для движения в глубину отсчитываются от 1 в сторону увеличения, то ссылки возврата считаются от последней в сторону уменьшения. Еще один важный нюанс в возвратном движении. В программе после использования ссылки для возврата элемент массива, содержащий ссылку, обнуляется. Это сделано для того, чтобы выделить ребро графа, по которому уже было выполнено движение в обе стороны и, следовательно, его необходимо полностью исключить из использования.

На второй вопрос мы уже ответили. Ответ на первый вопрос будет таким — найдем неиспользованную ссылку (в программе поиск осуществляется с конца), и если номер ссылки меньше количества вхождений в данный узел, то выполняем движение в глубину, иначе выполняем возврат.

Полностью процесс движения по графу завершаем тогда, когда уже нет неиспользованных ссылок в исходном узле. Каждая ссылка была использована как для движения в глубину, так и для возврата.

Алгоритм обхода графа:

```
Индекс текущего узла = 1
Пока есть хотя бы одна неиспользованная ссылка, делать
  Найти в текущем узле последнюю неиспользованную ссылку
  Если ее номер больше количества вхождений в узел,
  То движение вперед
    Увеличить значение счетчика вхождений для текущего узла.
    Запомнить узел-источник.
    Перейти по ссылке
    Записать адрес источника вместо использованной ссылки.
  Иначе возврат
    Уменьшить значение счетчика вхождений.
    Определить ссылку возврата и запомнить в промежуточную
    Переменную.
    Обнулить ссылку.
    Вернуться в узел-источник.
```

Текст программы приведен в листинге 19.1.

Листинг 19.1

```
program example;
 uses crt;
 type
 rec=record
      count:byte;
      num:integer;
      uk:array[1..255]  of integer;
     end;
 var
  uzel:array[1..100] of rec;
  pred_index,tek_index,i,j,n,m,c:integer;
  q:boolean;
procedure print;
begin
  if uzel[tek_index].num>0 then
    begin
     write(uzel[tek_index].num,'  ');
     uzel[tek_index].num:=0;
     readkey;
    end;
end;
```

```
begin
 {создание сети}
 clrscr;
 write('Введите количество узлов сети -');read(n);
 for i:=1 to n do
   begin
    write('Узел номер -',i);
    write('   Введите значение узла -');read(uzel[i].num);
    {Инициализация массива ссылок}
    for j:=1 to 255 do uzel[i].uk[j]:=0;
    uzel[i].count:=0;
    write('Введите количество ссылок -');read(m);
    for j:=1 to m do
      begin
       write('ссылка номер ',j,'=');
       read(uzel[i].uk[j]);
      end;
   end;
 {прохождение сети}
 tek_index:=1;pred_index:=1;
 repeat
  {Поиск последней ссылки}
  m:=1;
  while (uzel[tek_index].uk[m]<>0)and(m<255) do m:=m+1;
  {Цикл проскакивает значимую ссылку, поэтому надо вернуться назад на шаг}
  if m=255 then m:=0 else m:=m-1;
 if (uzel[tek_index].count<m) then {Движение вперед}
  begin
  print;
   {Расчет индекса массива для сохранения адреса узла источника}
   m:=m-uzel[tek_index].count;
   if tek_index>1 then uzel[tek_index].count:=uzel[tek_index].count+1;
   c:=tek_index;
   tek_index:=uzel[tek_index].uk[m];
   {Сохранение адреса узла источника}
   if c>1 then uzel[c].uk[m]:=pred_index
    else uzel[c].uk[m]:=0;
   pred_index:=c;
  end
 else {отход назад}
  begin
  print;
   if uzel[tek_index].count>0
```

```
   then
    begin
     m:=uzel[tek_index].count;
     uzel[tek_index].count:=uzel[tek_index].count-1;
     c:=uzel[tek_index].uk[m];
     uzel[tek_index].uk[m]:=0;
     tek_index:=c;
    end
   else tek_index:=pred_index;
  end;
 if tek_index=1 then
 begin
 q:=true;
 {Проверка, есть ли еще неиспользованные ссылки}
  for j:=1 to 255 do
   if uzel[1].uk[j]<>0 then q:=false;
 end;
 until q; {Завершение процесса}
end.
```

В заключение. Мы получили работающую программу. Но это еще не все. Если вы тщательно ее протестируете, то, вполне возможно, сможете найти неработающий пример. Но с другой стороны, работающих примеров можно привести сколько угодно много. Что это означает? Можно ли говорить, что программа ошибочна? Или возможно сделать другой вывод?

Все зависит от установки на то, что требовалось. Можно сказать, что алгоритм и соответственно программа ошибочны, а можно сказать, что программа верна, но существуют определенные ограничения. С такой ситуацией мы уже встречались в задаче о раскладке большой кучи камней на две. Давайте и данную ситуацию воспринимать так же. Все хорошо, но есть ограничения, которые надо четко определить.

Формулировка ограничений не займет у нас много сил. Обратите внимание на ключевой пункт нашей идеи. Это запись ссылок возврата вместо ссылок на следующие узлы. Отсюда следует и ограничение. Сформулируем его для графа: алгоритм будет работоспособным только в том случае, когда для любого его узла количество входящих ребер не меньше, чем количество исходящих.

Технически задача достаточно сложная. При поиске решения использовались две различные терминологии. Мы рассуждали в терминах графов и при этом говорили о входящих и исходящих ребрах, рассуждали в терминах ссылок движения в глубину и ссылок возврата. Возможно, это не для всех оказалось удобно и, может быть, вы запутались в терминологии и неправильно

соотнесли понятия ребер и ссылок. Если это случилось, то придется тщательно поработать с программой. А в помощь попробуйте проанализировать два примера (рис. 19.3). Слева приведен работающий пример. Пример справа немного отличается от примера слева, но он не работает корректно. Анализ этих примеров поможет вам более точно понять работу программы. Заметим, что любые деревья у нашей программы проблем не вызывают.

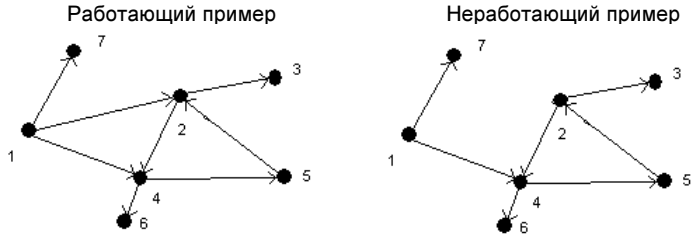

Рис. 19.3

Глава 20

Закраска односвязного контура

Условие задачи. На плоскости дан произвольный односвязный контур. Требуется его закрасить.

Контуром будем называть ломаную линию, конец которой совпадает с началом. Контур имеет внутреннюю область. Внутренняя область отличается от внешней тем, что любые две точки, одна из которых принадлежит внутренней области, а другая — внешней, разделяются линией контура и пройти из одной точки в другую, не пересекая линии контура, нельзя. Односвязный контур — это такой контур, что любые две точки его внутренней области можно соединить линией (не обязательно прямой), не пересекая линии контура.

Задача достаточно сложная. Есть в ней трудности и идейного характера и технического. Приступая к такой задаче, полезно решить похожую, но более простую. Это, во-первых, поможет в выработке идеи и уж во всяком случае даст полезный опыт в отработке необходимых технических навыков. Конечно, может получиться и так, что решение простого аналога окажется бесполезным, но опыт говорит, что похожие задачи часто имеют похожие решения.

Можно, конечно, возразить и так: решая похожую задачу, мы тратим время, которое можно было бы потратить на решение исходной, это потраченное время нам никто не вернет, а решать большую задачу все равно придется с нуля. Это правильное возражение, если у вас уже есть хорошая идея и вы четко представляете, как ее реализовать, но ведь у нас пока нет никаких идей и тем более ясного представления, что делать!

Поэтому подумаем об упрощении. В условии используется три содержательных термина: "контур", "произвольный", "односвязный". Упрощение условия должно быть как-то связано с изменением смысла этих терминов. Термин "контур", конечно, изменять нельзя, он заключает в себе главный

смысл задачи. "Односвязный" можно заменить, изменив связность контура, но это не облегчит, а усложнит задачу. Следовательно, остается только одна возможность — это сузить понятие "произвольный". И сразу запишем новое правило.

==========

Правило объема терминологии

Если упрощение задачи — это изменение области применения искомого алгоритма, то следует в условии выделить существенные термины, определить область их значений и подумать, как можно уменьшить области значений без критического искажения смысла задачи.

==========

Ограничение задачи контурами определенной формы, например прямоугольниками, будет слишком сильным сужением, попробуем поработать с произвольными выпуклыми контурами. Выпуклый контур можно определить как контур, любые две внутренние точки которого можно соединить прямой линией не пересекающейся с линией контура. Хорошим примером выпуклого контура может быть любой параллелограмм или треугольник. Более сложный пример показан на рис. 20.1.

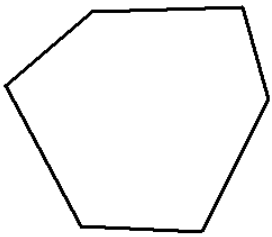

Рис. 20.1

Сейчас попробуем обсудить некоторые алгоритмические идеи.

Идея 1. Заливка. Представьте себе, как растекается вода, если ее вылить на ровную поверхность. Если поверхность будет совершенно ровной, то граница водной поверхности будет, очевидно, окружностью (так как все направления равноправны) и остановится вода, только достигнув контура (а контур представьте себе как некий бордюрчик). Для реализации такого процесса достаточно одной точки, гарантированно находящейся во внутренней области. Затем запускаем процесс, в котором для каждой уже закрашенной точки просматриваются все соседи и для каждой соседней, не закрашенной точки выполняется закраска, после чего она рассматривается в качестве следующего источника закраски.

Описанный процесс явно рекуррентный, и если у вас появилась идея решить задачу рекурсивно, то эта мысль вполне естественна. Но против идеи рекурсивного решения есть сильное возражение. Рекурсия требует серьезных ресурсов памяти, а наш процесс будет очень сильно ветвиться, и мы рискуем сорвать закраску из-за переполнения стека. Но это будет хороший пример необходимости учитывать возможности машины и системы программирования. Иначе говоря, необходимо помнить, что алгоритм не может висеть в воздухе, он выполняется на конкретном компьютере, располагающем конкретными возможностями.

Впрочем, можно поступить иначе. Если точки, являющиеся источником закраски, заносить в специальный массив записей координат, то без рекурсивных процедур можно обойтись. При очень длинном фронте разлива краски этот массив может растянуться на тысячи точек, что не слишком страшно. Если несколько тысяч одновременно работающих процедур — это много, то несколько тысяч элементов массива — это не очень много. Но тогда возникнет существенная логическая сложность. Каждая конкретная точка может являться точкой разлива краски только один шаг цикла закраски. Это означает, что массив необходимо постоянно перестраивать. Одни точки в него будут записываться, другие уходить, кроме того, длина массива не остается постоянной, и это все существенно усложнит логику алгоритма.

Может быть, упомянутые логические сложности и не так страшны, мы сейчас просто обозначили их существование. Кроме того, худа без добра не бывает, и если удастся преодолеть все преграды, мы будет существенно вознаграждены.

Продолжим. Каждая точка-источник видит вокруг себя только на одну точку, т. е. каждому источнику совершенно безразлична форма закрашиваемого контура. Это означает, что мы попали на идею, которая поможет сразу решить исходную задачу. И решать вспомогательную задачу нет никакой необходимости.

Но у этого подхода есть существенный минус. Если закрашиваемая фигура велика, то фронт разливаемой краски может быть длинным, следовательно, массив, предназначенный для хранения координат точек-источников, также может оказаться большим. Это влечет за собой два недостатка: расход памяти и потерю скорости, которая будет неизбежна при перестройке массива.

Поэтому мы попробуем решить задачу двумя способами — изложенным только что и новым методом, не требующим хранения больших массивов. Такой метод нам пока неизвестен, его мы еще поищем. И для успешности поиска нового метода все же решим упрощенную задачу (с закраской выпуклого контура).

Идея 2. Закраска параллельных линий. Идея решения несложна. Мы имеем одну точку, гарантированно находящуюся внутри закрашиваемого контура.

Пройдем по горизонтали влево от нее и вправо с целью найти соответственно левую и правую точки контура. Определив координаты искомых точек, проведем между ними линию. Затем поднимемся на один пиксел вверх и повторим описанные операции. Ясно, что при подъеме вверх внутренняя часть контура будет закрашиваться. Также поступим и для части контура лежащей ниже исходной точки. В построенном способе осталась одна неточность. Говорится о подъеме вверх и о спуске вниз и не говорится о том, до каких пор? Каково условие завершения подъема и соответственно спуска? Ответ на этот вопрос не так прост. Предположим, мы будем выполнять подъем до тех пор, пока не встретим точку контура. Тогда возникает проблема, проиллюстрированная на рис. 20.2.

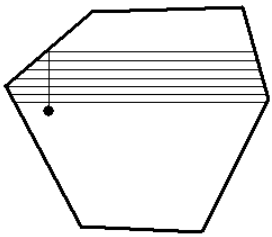

Рис. 20.2

Здесь вертикальный отрезок — это линия движения от изначально известной точки до касания с контуром, а горизонтальные линии показывают закраску. Видно, что вертикальный отрезок в своем движении вверх встретил линию контура до того, как контур был закрашен в части, расположенной выше исходной точки.

Проблема, впрочем, несложная. Ясно, что исходную точку надо просто смещать от границ контура. Делать это можно по-разному, мы используем красивый прием, позволяющий не выяснять, с какой стороны находится ближайшая линия границы.

На каждом шаге смещения вверх для определения координат закрашивающей линии необходимо вычислять координаты *X* левой и правой границы контура. Обозначим их через `rx` и `lx`. Необходимо обеспечить выполнение неравенства `lx<=x<=rx`. Среднее арифметическое величин `lx` и `rx` всегда находится между ними. И если организовать пересчет координаты *X* на каждом шагу подъема (спуска) как среднее арифметическое крайних точек, то точка всегда будет стремиться при подъеме к самой верхней вершине и при спуске к самой нижней. Отсюда сразу видно и условие завершения подъема (спуска). Подъем (спуск) возможен только до тех пор, пока `rx-lx>0`, то есть до тех пор, пока левая и правая точки не совпадут.

Для разработки программы заметим, что спуск и подъем отличаются только направлением, следовательно, эти два процесса возможно реализовать

одной процедурой, в которую в качестве параметров передаются координаты исходной точки и направление смещения. Операции поиска левой и правой точки отличаются только направлением, поэтому их также можно оформить одной функцией, получающей в качестве аргументов координаты точки, исходной для поиска и возвращающей координату *X* найденной точки.

Текст программы, записанный в листинге 20.1, выглядит совершенно прозрачно, поэтому описывать алгоритм не будем. Заметим только, что часть программы, в которой выполняется ввод, закомментирована. А данные для хорошего примера вводятся посредством группы операторов присваивания.

Листинг 20.1

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  point:array[1..20] of record
                         x,y:integer;
                        end;
  x,y,n,i:integer;
procedure paint(x,y,k:integer);
var
 lx,rx:integer;
function search(x,y,k:integer):integer;
begin
 while getpixel(x,y)<>15 do x:=x+k;
 search:=x;
end;
begin
  repeat
   y:=y+k;
   rx:=search(x,y,1);
   lx:=search(x,y,-1);
   line(lx,y,rx,y);
   delay(20000);
   x:=(lx+rx) div 2;
  until rx-lx=0;
end;
begin
 {read(n);
 for i:=1 to n do
   read(point[i].x,point[i].y);
 read(x,y);}
```

```
 n:=5;
 x:=160;y:=130;
 point[1].x:=150;point[1].y:=90;
 point[2].x:=260;point[2].y:=150;
 point[3].x:=240;point[3].y:=180;
 point[4].x:=200;point[4].y:=200;
 point[5].x:=100;point[5].y:=150;
 dr:=detect;initgraph(dr,md,'');
 for i:=1 to n-1 do
  begin
   line(point[i].x,point[i].y,point[i+1].x,point[i+1].y);
  end;
 line(point[n].x,point[n].y,point[1].x,point[1].y);
 paint(x,y,1);
 paint(x,y+1,-1);
 readkey;
end.
```

Если вы не сочтете за труд как следует протестировать программу, то попробуйте залить контуры, в которых хотя бы одна сторона сильно наклонена к горизонтали. Вполне возможно, вам удастся найти пример, в котором краска выльется за пределы контура. Если такой пример встретится, посмотрите внимательно на сильно наклоненную сторону. Хорошо видно, что фактически это не линия, а лесенка, между ступеньками которой могут быть пустые точки. Это небольшая техническая проблема, не имеющая принципиального значения, решать ее мы не будем, а вы попробуйте.

Мы же двинемся далее. Имея средство заливки краской выпуклых фигур, можно подумать о произвольном контуре, так как любой контур можно представить как объединение нескольких выпуклых. Пример такого объединения приведен на рис. 20.3.

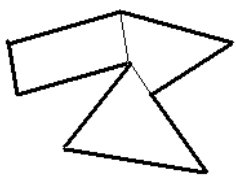

Рис. 20.3

На рисунке тонкими линиями показано деление сложной фигуры на несколько выпуклых. Таким образом, задача закраски произвольного односвязного контура свелась к задаче разбиения произвольной фигуры на несколько выпуклых. Такое разбиение, наверное, можно выполнить, если

знать координаты вершин контура. Но в этом случае метод закраски теряет универсальность, намного интереснее было бы найти метод, выполняющий свою работу и ничего не знающий о контуре, окружающем исходную точку. Кроме того, использование координат вершин, скорее всего, потребует достаточно трудоемких расчетов.

Конечно, было бы замечательно, если бы произвольный контур удалось закрасить, располагая только координатами одной внутренней точки. Однако логика и здравый смысл подсказывают, что резкое усложнение условия требует дополнительной информации. Это как золотое правило механики, выигрываем в расстоянии, проигрываем в силе и наоборот. Так же и здесь, выигрываем в условии (делаем его более сложным), проигрываем в известной информации (больше требований к известному). Однако золотое правило можно направить в другую сторону. Можно попробовать не требовать дополнительных условий, а попробовать разработать более сложный алгоритм.

Более сложный алгоритм, используя только текущую информацию, должен каким-то образом определять появление новой выпуклой фигуры. Если в качестве основы для рассуждений используем алгоритм закраски выпуклой фигуры, то текущая информация — это координаты левой и правой точек контура относительно точки спуска (подъема). Рассмотрим более внимательно процесс спуска на фигуре, разобранной ранее (рис. 20.4).

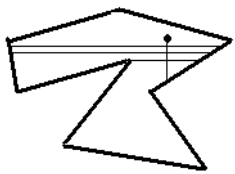

Рис. 20.4

Исходная точка обозначена черным кружочком. Вертикальный отрезок показывает направление спуска. Направление показано без пересчета координаты точки спуска через среднее арифметическое левой и правой точек контура. Мы опустили пересчет для упрощения анализа. Итак, видим, что в некоторый момент горизонтальная линия резко изменяет свою длину, в данном случае она становится существенно короче, одновременно с этим событием слева от точки спуска появляется треугольник, выпадающий из процесса заливки. Следовательно, можно резкое изменение длины связать с появлением новых фигур (точнее с выпадением части фигуры из процесса заливки).

Нас, впрочем, интересует не только факт появления новой фигуры, а и то, где она находится. Поэтому рассмотрим не просто изменение длины горизонтальной линии, а изменение координат X левой и правой точек контура.

Это действительно даст больше информации. Применительно к рассмотренному примеру можно заметить, что:

- ❒ выполнялось движение вниз;
- ❒ сильно изменилось значение левой координаты, она стала меньше.

Из этого делаем вывод, что координату *X* новой точки закраски можно определить как среднее арифметическое координаты *X* предыдущей левой точки и координаты *X* текущей левой точки. При этом новая точка закраски будет точкой спуска. Для лучшего понимания рассмотрим немного иной пример (рис. 20.5).

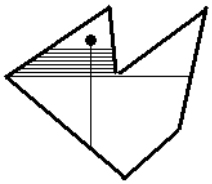

Рис. 20.5

Здесь точка закраски также спускается вниз, но изменения происходят не с левой, а с правой точкой, и ее координата не уменьшается, а увеличивается, при этом новая фигура расположена вверху, поэтому новая точка закраски будет уже точкой подъема. Двух приведенных примеров, наверное, достаточно, чтобы понять, как принципиально решается задача.

- ❒ Мы отслеживаем изменения координат левой и правой точек.
- ❒ Если координата *X* одной из них претерпела резкое изменение, а слишком резким изменением будем считать изменение на 2 точки (по причине различного наклона линий контура), то формируем новую точку закраски, определяем новое направление и запускаем новый процесс закраски относительно определенной точки.

Единственно, что осталось сделать, это определить возможные ситуации и для каждой определить координаты новой точки закраски и направление закраски. Возможно несколько ситуаций.

- ❒ Левая граница стала больше при движении вниз.
  - Координата *X* новой точки закраски есть среднее арифметическое координат *X* предыдущей и текущей левой точки. Новое направление закраски — вверх.
- ❒ Левая граница стала больше при движении вверх.
  - Координата *X* новой точки закраски есть среднее арифметическое координат *X* предыдущей и текущей левой точки. Новое направление закраски — вниз.

- ❒ Левая граница стала меньше при движении вверх.
  - Координата *X* новой точки закраски есть среднее арифметическое координат *X* предыдущей и текущей левой точки. Новое направление закраски — вверх.
- ❒ Левая граница стала меньше при движении вниз.
  - Координата *X* новой точки закраски есть среднее арифметическое координат *X* предыдущей и текущей левой точки. Новое направление закраски — вниз.

Четыре аналогичных случая определяются и для возможных изменений правой точки. Мы их рассматривать не будем, так как аналогия практически полная. Программное решение оформим в виде рекурсивной процедуры закраски. Каждый новый вызов процедуры закраски выполняется при обнаружении одной из восьми возможных ситуаций в левой и правой точках. Текст программы приведен в листинге 20.2. Он достаточно прозрачен, поэтому расписывать алгоритм не будем.

Листинг 20.2

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  point:array[1..20] of record
                         x,y:integer;
                        end;
  x,y,n,i:integer;
procedure paint(x,y,k:integer);
var
 lx,rx,lx1,rx1:integer;
function search(x,y,k:integer):integer;
begin
 while getpixel(x,y)<>15 do x:=x+k;
 search:=x;
end;
procedure ris;
begin
   y:=y+k;
   rx:=search(x,y,1);
   lx:=search(x,y,-1);
   line(lx,y,rx,y);
   delay(2000);
   x:=(lx+rx) div 2;
end;
```

```
begin
 ris;
 lx1:=lx;rx1:=rx;
 while rx-lx>=1 do
  begin
   ris;
   if (lx-lx1>2) and (k=-1) then paint((lx+lx1) div 2,y,1);
   if (lx-lx1>2) and (k=1) then paint((lx+lx1) div 2,y-1,1);
   if (lx1-lx>2) and (k=-1) then paint((lx+lx1) div 2,y-1,-1);
   if (lx1-lx>2) and (k=1) then paint((lx+lx1) div 2,y,-1);

   if (rx-rx1>2) and (k=-1) then paint((rx+rx1) div 2,y,1);
   if (rx-rx1>2) and (k=1) then paint((rx+rx1) div 2,y,-1);
   if (rx1-rx>2) and (k=-1) then paint((rx+rx1) div 2,y,-1);
   if (rx1-rx>2) and (k=1) then paint((rx+rx1) div 2,y-1,1);
   lx1:=lx;rx1:=rx;
  end;
end;
begin
 {read(n);
 for i:=1 to n do
   read(point[i].x,point[i].y);
 read(x,y);}
 n:=9;
 x:=100;y:=110;
 point[1].x:=100;point[1].y:=100;
 point[2].x:=190;point[2].y:=170;
 point[3].x:=200;point[3].y:=200;
 point[4].x:=250;point[4].y:=150;
 point[5].x:=270;point[5].y:=280;
 point[6].x:=135;point[6].y:=180;
 point[7].x:=90;point[7].y:=280;
 point[8].x:=30;point[8].y:=170;
 point[9].x:=60;point[9].y:=200;
 dr:=detect;initgraph(dr,md,'');
 for i:=1 to n-1 do
  begin
   line(point[i].x,point[i].y,point[i+1].x,point[i+1].y);
   readkey;
  end;
 line(point[n].x,point[n].y,point[1].x,point[1].y);
 {putpixel(x,y,15);}
 paint(x,y,1);
```

```
 readkey;
 paint(x,y+1,-1);
 readkey;
end.
```

Полученное решение обладает теми же недостатками, которые были указаны для программы закраски выпуклого контура. Достоинством решения является малая требовательность памяти, необходимой для хранения промежуточных данных. Достаточно трудно прорисовать контур настолько сложный, что потребуется работа большого количества рекурсивных процедур закраски. Наверное, это решение можно назвать оптимальным, если исправить имеющиеся недоработки, но тем не менее мы поищем второе решение.

Второе решение основано на идее, обсуждавшейся в самом начале главы: назовем исходную точку закраски источником. Смысл этого названия в том, что точка является источником краски для четырех соседних точек. Тогда весь процесс закраски выглядит так: мы создаем массив точек источников. На каждом шагу закраски просматриваем весь массив и для каждой точки источника выполняем проверку, какие соседи еще не закрашены. Найденный незакрашенный сосед закрашивается и становится новым источником закраски. Точка, соседи которой проверялись, перестает быть источником и убирается из массива. Самый первый источник — это известная точка. Таков принцип. Но конечно же, принцип — это далеко не все, еще придется сражаться с техническими проблемами реализации.

Процесс закраски на первый взгляд совершенно понятен. У каждой точки только четыре соседа (соседей по диагонали рассматривать не будем). Проверить их цвет не составляет труда. Интуитивно ясно, что наиболее сложное место в программе — это массив точек-источников. Источники в массив должны заноситься и удаляться, соответственно длина массива постоянно меняется, а характер этих изменений зависим от формы контура.

Конечно, мы можем построить алгоритм таким образом, чтобы точки-источники на обработку выбирались не хаотично, а в определенном порядке, например с начала массива или с конца, такой порядок даже будет наиболее естественен, но предсказать, сколько новых источников породит данный, практически невозможно. Известно только то, что как минимум их может вообще не быть, а как максимум — четыре. Правда, четыре соседа у источника может быть только на первом шаге, когда внутри контура есть только один источник — исходный.

Предположим, что точки-источники, кандидаты на обработку выбираются с начала массива. Это означает, что, взяв хотя бы один источник, мы создали в массиве одну дырку, которую можно заполнить. Если взятый источник породит только один новый источник, то мы его запишем на место исполь-

зованного, а если он породит несколько источников? Или если он не породит ни одного, и дырка в начале массива начнет разрастаться на следующих шагах? Из сказанного видно, что если мы и можем брать источники на обработку только с начала, то записывать новые только в начало уже не получится. Возникшую проблему можно обойти, если источники на обработку брать не сначала, а с конца. Тогда разрывов в массиве источников не будет. Но мы пока выберем другую стратегию поведения. Введем для каждого элемента массива дополнительное описательное поле — флаг, который сыграет роль метки. Если элемент массива использован, то флаг имеет значение "истина", иначе флаг имеет значение "ложь". Тогда для выбора из массива источников кандидата на обработку достаточно найти в массиве источник с флагом, равным "истина". Данный прием сразу дает критерий завершения процесса — процесс завершается тогда, когда не нашлось ни одного источника с истинным флагом.

В нашей программе, правда, используется другой косвенный критерий. Ясно, что если с каждым новым источником длину массива источников увеличивать, а с каждым использованным — уменьшать, то к концу процесса длина массива окажется равной нулю. Данный факт мы и используем для завершения процесса.

Это были очевидные трудности, а есть одна неочевидная. Рассмотрим рис. 20.6.

Рис. 20.6

Черные точки — это источники, уже занесенные в массив источников, а белая — нет. В этой ситуации две черные точки могут объявить белую точку следующим источником, что даст удвоение потребности в размере массива в данной локальной ситуации, а фактически размер массива начнет расти почти неограниченно, так как ситуации такого типа не редкость. В нашей стратегии записи (исключения) источников исключить источник можно только один раз, следовательно, если он окажется записанным несколько раз, балласт из лишних записей будет накапливаться быстрее, чем сбрасываться. Опытная программа, написанная без учета озвученной проблемы, показала, что удается закрасить только незначительную поверхность и существенное увеличение длины массива источников не приводит к значительному увеличению закрашенной области.

Решение проблемы заключается в ответе на вопрос, в каком порядке выполнять действия: записать точку как источник, а потом ее закрасить (когда она будет обнаружена в поиске источника) или наоборот.

Дело в том, что после занесения в массив источников точка может достаточно долго не попасть на обработку, если она была записана в конец массива, а это как раз и означает появление реальной возможности ее повторной и даже многократной записи. Чтобы этого избежать, необходимо после того, как точка попадет в массив, ее закрасить, тогда для всех прочих ее соседей она уже не сможет быть источником. Поэтому ответ на поставленный вопрос такой — точка должна быть закрашена тогда же, когда ее координаты записываются в массив источников.

Проблема с закраской/занесением делает программу не совсем тривиальной, поэтому запишем ее алгоритм:

```
Длина массива источников = 1
Записываем в массив источников исходную точку
Пока длина массива источников больше нуля, делать
  Найти первый неиспользованный источник и для него
    Объявить его использованным
    Длину массива увеличить на 1
    Провести анализ соседей следующим образом
      Если сосед незакрашенная точка, то
        Найти неиспользованный элемент массива источников.
        Сохранить в нем информацию о точке-источнике.
        Закрасить точку
        Уменьшить длину массива источников на 1
    Перейти в начало массива.
```

Обратим еще раз внимание на логику изменения длины массива источников. Уменьшение длины выполняется вполне логично. Если мы закрасили точку, то она полностью выпадает из дальнейшего анализа, ее нужно убрать из массива, в результате чего длина массива, конечно, уменьшится, но тут появляется парадокс, посмотрите внимательно, мы заносим новый источник в массив и в тот же момент уменьшаем длину массива, и наоборот, убираем точку из массива, а длину увеличиваем.

Причина уже объяснена ранее. Точка-источник закрашивается в момент занесения в массив источников. Но, конечно, такая последовательность действий в алгоритме выглядит странно, чем осложняет понимание.

Текст программы приведен в листинге 20.3.

Листинг 20.3

```
program example;
 uses crt,graph;
 var
```

```
  dr,md:integer;
  point:array[1..20] of record
                         x,y:integer;
                        end;
  front:array[1..10000] of record
                            x,y:integer;
                            flag:boolean;
                           end;
  max,x,y,n,i:integer;
procedure paint(x,y:integer);
var
 i:integer;
begin
 if getpixel(x,y)=0 then
  begin
   i:=1;
 { В цикле используется дополнительная проверка для параметра I,
   на тот случай, если фронт заливки окажется слишком велик}
   while (front[i].flag) and (i<10000) do i:=i+1;
   front[i].flag:=true;
   front[i].x:=x;
   front[i].y:=y;
   putpixel(front[i].x,front[i].y,15);
   max:=max-1;
  end;
end;
begin
 {read(n);
 for i:=1 to n do
   read(point[i].x,point[i].y);
 read(x,y);}
{ n:=5;
 x:=160;y:=130;
 point[1].x:=150;point[1].y:=90;
 point[2].x:=260;point[2].y:=150;
 point[3].x:=240;point[3].y:=180;
 point[4].x:=200;point[4].y:=200;
 point[5].x:=100;point[5].y:=150;}
 n:=9;
 x:=100;y:=110;
 point[1].x:=100;point[1].y:=100;
 point[2].x:=240;point[2].y:=120;
 point[3].x:=200;point[3].y:=200;
 point[4].x:=250;point[4].y:=150;
```

```
 point[5].x:=270;point[5].y:=280;
 point[6].x:=135;point[6].y:=180;
 point[7].x:=90;point[7].y:=280;
 point[8].x:=30;point[8].y:=170;
 point[9].x:=60;point[9].y:=200;

 dr:=detect;initgraph(dr,md,'');
 for i:=1 to 10000 do
   front[i].flag:=false;
 for i:=1 to n-1 do
   line(point[i].x,point[i].y,point[i+1].x,point[i+1].y);
 line(point[n].x,point[n].y,point[1].x,point[1].y);
 putpixel(x,y,15);
 readkey;
 max:=1;
 front[1].x:=x;front[1].y:=y;front[1].flag:=true;
 i:=0;
 repeat
  i:=i+1;
  if front[i].flag then
    begin
      front[i].flag:=false;
      delay(100);
      max:=max+1;
      paint(front[i].x,front[i].y-1);
      paint(front[i].x+1,front[i].y);
      paint(front[i].x,front[i].y+1);
      paint(front[i].x-1,front[i].y);
      i:=0;
    end;
 until max=0;
 readkey;
end.
```

Обратите внимание на последовательность, в которой программа закрашивает точки. Она действительно трудно поддается пониманию. Минусы программы — сложность логики и затраты памяти. Однако есть и компенсация. Если первый подход нуждается в доработке, напомним, что первая программа в том варианте, в котором она приведена, не может корректно закрасить контуры, содержащие линии, сильно наклоненные к горизонтали. Данная программа избавлена от такого недостатка.

А сейчас третий вариант решения. Попробуем сделать логику работы с массивом источников более естественной. Для этого будем брать на обработку источники с конца массива и выполнять запись новых также в конец.

Логика изменения массива резко упрощается. Он начинает работать как стек. Источник, последним попавший в массив, первый попадает на обработку. В массиве нет дырок и нет балласта. Логика программы, реализующая идею, настолько прозрачна, что мы не будем записывать алгоритм, а ограничимся программой (листинг 20.4).

Запустите эту программу с первым примером. Логика выбора точек для закраски видна сразу, и можно подумать, что мы нашли идеальный алгоритм, если не удастся для первого подхода найти легкое решение сильно наклоненных линий, то, пожалуй, так оно и будет.

Листинг 20.4

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  point:array[1..20] of record
                         x,y:integer;
                        end;
  front:array[1..10000] of record
                            x,y:integer;
                           end;
  max,x,y,n,i:integer;
procedure paint(x,y:integer);
begin
 if getpixel(x,y)=0 then
  begin
   putpixel(x,y,15);
   max:=max+1;
   front[max].x:=x;
   front[max].y:=y;
  end
end;
begin
 {read(n);
 for i:=1 to n do
   read(point[i].x,point[i].y);
 read(x,y);}
 {n:=5;
 x:=160;y:=130;
 point[1].x:=150;point[1].y:=90;
 point[2].x:=260;point[2].y:=150;
 point[3].x:=240;point[3].y:=180;
 point[4].x:=200;point[4].y:=200;
```

```
 point[5].x:=100;point[5].y:=150;}
 n:=9;
 x:=100;y:=110;
 point[1].x:=100;point[1].y:=100;
 point[2].x:=240;point[2].y:=120;
 point[3].x:=200;point[3].y:=200;
 point[4].x:=250;point[4].y:=150;
 point[5].x:=270;point[5].y:=280;
 point[6].x:=135;point[6].y:=180;
 point[7].x:=90;point[7].y:=280;
 point[8].x:=30;point[8].y:=170;
 point[9].x:=60;point[9].y:=200;

 dr:=detect;initgraph(dr,md,'');
 for i:=1 to n-1 do
   line(point[i].x,point[i].y,point[i+1].x,point[i+1].y);
 line(point[n].x,point[n].y,point[1].x,point[1].y);
 readkey;
 max:=1;
 front[1].x:=x;front[1].y:=y;
 repeat
  delay(100);
  x:=front[max].x;
  y:=front[max].y;
  max:=max-1;
  paint(x,y-1);
  paint(x+1,y);
  paint(x,y+1);
  paint(x-1,y);
 until max=0;
 readkey;
end.
```

В заключение. Завершенная глава самая объемная и одна из самых информативных. Мы разработали целых три решения, рассмотрели важные проблемы. Начался анализ с формулировки продуктивного правила упрощения задачи, в заключение же сформулируем еще одно техническое.

==========

Правило управления массивом

Для массива переменной длины наиболее эффективная форма организации — это стек.

Глава 21

Живая группа ГО

В игре ГО, правила которой, конечно, сейчас изучать не будем, есть понятие живой группы. В задаче требуется, имея конкретную позицию и координаты одного камня, установить, является ли этот камень представителем живой группы.

Возможно, вы не знакомы с правилами этой игры. Если так, то ничего страшного. Фактически сама игра для формулировки задачи не нужна, нужно только определение живой группы. Игра ведется на квадратной доске размером 19×19. Это классическая доска ГО. Фигуры игры называются *камнями*. Ход игрока заключается в установке камня на перекрестие линий (правда, мы в решении будем устанавливать камень в клетку). Группой камней называется множество, в котором каждый камень имеет хотя бы одного соседа, принадлежащего к этому же множеству. Соседом называется камень, стоящий на соседнем перекрестии по вертикали или горизонтали. Камни, стоящие рядом по диагонали, соседями не являются. *Живой группой* называется такая группа, в которой есть хотя бы один камень, такой что хотя бы одно соседнее с ним перекрестие пусто. На рис. 21.1 показаны примеры живой и неживой групп.

Живая группа

ч	ч	ч	ч	ч
ч	б	б	б	ч
ч	б		б	ч
ч	б		б	ч
ч	б	б	б	ч
ч	ч	ч	ч	ч

Неживая группа

ч	ч	ч	ч	ч
ч	б	б	б	ч
	ч	ч	б	ч
		ч	б	ч
			ч	

Рис. 21.1

На рисунке камни изображены буквами. Буквой "**б**" — белый камень и буквой "**ч**" — черный камень. Определение живой группы дано, примеры, демонстрирующие определение, приведены, можно приступать к собственно решению.

На примере этой задачи мы еще раз продемонстрируем различие между рекурсивным и нерекурсивным решением. Вспомним, что в "программисткой" науке есть (правда нестрогое) утверждение о том, что каждую задачу можно решить как рекурсивно, так и не рекурсивно. Вопрос только в осмысленности и эффективности. Поиск рекурсивных решений часто очень сложен, но бывает и наоборот. Например, в уже рассмотренной нами задаче об экономном обходе графа дано нерекурсивное решение и его логика достаточно запутана. Ее рекурсивное решение было бы проще. Но рекурсивное решение той же задачи потребовало бы большего количества ресурсов. Так часто бывает, рекурсия очень требовательная дама.

Задача, к которой мы сейчас приступаем, имеет хорошее рекурсивное и хорошее нерекурсивное решение. Более того, различие между этими двумя типами решений в задаче о живой группе ГО можно свести к минимуму. Минимальное различие заключается в организации памяти. Для рекурсии нужен стек. Следовательно, чтобы организовать рекурсию, необходимо создать стековую память, например, посредством массива. Вот именно это и будет сделано. Первым шагом мы получим рекурсивное решение, обсудим его и затем получим второе, нерекурсивное решение.

Первый шаг в построении рекурсивного решения — это построение рекуррентного процесса. В нашей задаче рекуррентный процесс налицо. Сущность процесса заключается в переходе от камня к камню и поиске хотя бы одного свободного поля. Если таковое поле найдено, то группа живая, иначе нет. Рекуррентный характер процесса в том, что поиск свободного поля либо соседа по группе осуществляется для каждого текущего камня, а если найден сосед, то для него поиск повторяется. Общая схема рекурсии может быть такова:

- если рассматриваемое поле пустое, то поиск прекращается и возвращается сообщение об удачном его завершении;
- если рассматриваемое поле занято камнем противника, то поиск прекращается и возвращается сообщение о неудаче рекурсивного поиска;
- если рассматриваемое поле занято своим камнем, то выполняется новый вызов рекурсивной процедуры.

Это самая общая схема, не лишенная проблем. Первая и самая грубая ошибка заключается в том, что текущий вызов рассматривает только одно поле, а мы знаем, что у каждого камня может быть четыре соседа. Технически это означает, что либо должно быть четыре вызова процедуры, либо процедура должна рассматривать четыре ситуации. Как именно построить

работу рекурсивной процедуры — вопрос вкуса и личного стиля. Мы выберем второй вариант — в процедуру передаются координаты поля, на котором находится свой камень, и процедура рассматривает четырех его соседей.

Пусть процедура получает на вход две координаты *X*, *Y*, тогда ее внешний вид может быть такой:

```
СИТУАЦИЯ = НЕУДАЧА
Рассматривается сосед слева
  Если пустое поле, то СИТУАЦИЯ = УДАЧА
  Если камень противника, то СИТУАЦИЯ = НЕУДАЧА
  Если камень свой, то вызов процедуры с новыми координатами
Если СИТУАЦИЯ = НЕУДАЧА, то рассматривается сосед сверху
  Если пустое поле, то СИТУАЦИЯ = УДАЧА
  Если камень противника, то СИТУАЦИЯ = НЕУДАЧА
  Если камень свой, то вызов процедуры с новыми координатами
```

И точно так же проработаем соседей справа и снизу. Сразу исправим небольшую ошибку (или просто недоговоренность). Ранее везде говорилось о рекурсивной процедуре и по инерции в алгоритме речь идет о процедуре. В то же время говорится о некоторой переменной под именем `СИТУАЦИЯ`. Очевидно, данная переменная должна сообщать свое значение предыдущим вызовам, а для этого либо она должна быть глобальной величиной, либо рекурсию необходимо оформить в виде функции.

Следующая более тонкая ошибка нам уже встречалась в задаче о закраске контура, имеются в виду решения, основанные на понятии точки — источнике закраски. Вспомните, там мы столкнулись с тем, что одна и та же точка может оказаться соседом нескольких точек, в результате чего она попадет в массив источников многократно, кроме того, соседом текущей точки является и та точка, которая была источником для текущей, а это уже грозит движением по кругу.

Клетки доски ГО также можно рассматривать как точки, а следовательно проблемы предыдущей задачи полностью переходят и на данную. Но это в общем-то плюс, так как описанные проблемы нами уже решены, а их полная идентичность означает, что и решать мы их можем тем же методом. То есть, чтобы вторично не попасть в клетку, занятую своим камнем, мы этот камень после проведенного анализа, независимо от результата, уберем из клетки. Технически это можно осуществить так: пусть доска — это двумерный массив и:

- ноль — клетка пуста;
- единица — клетка занята собственным камнем;
- двойка — клетка занята камнем противника.

Тогда для того чтобы убрать камень из клетки, можно единицу заменить на любое другое число, кроме нуля. В программе это число три. Свою работу рекурсивная функция начнет с указанного камня, каждый вызов породит столько вызовов, сколько у текущего камня окажется соседей. Возвращать функция будет истину, если она получит хотя бы одну истину от порожденных вызовов функций, и ложь, если все порожденные вызовы функций вернут ложь. Таким образом, сам вызов функции вернет ложное значение только в том случае, если все порождаемые вызовы всех уровней вернут значение ложь.

Примечание

После того как было указано на сходство проблем в двух разных задачах, оставшиеся проблемы имеют характер технических вопросов, а не принципиальных, но, конечно, нужно согласиться, что задача о живой группе ГО и задача о закраске контура выглядят слишком по-разному, чтобы искать в них аналогию. Однако ничего странного нет. Аналогию мы увидели совсем не в тексте условий, а в процессе поиска нужной точки. Во-первых, грубая аналогия видна сразу. В обеих задачах ищется точка с определенными свойствами. Более тонкая аналогия в том, как она ищется. И это в обеих задачах делается одинаково. А именно для каждой точки просматриваются соседи и для каждого соседа проводится анализ его свойств. Различие в задачах следующее. В задаче о закраске нужны все пустые точки, а в задаче о живой группе — хотя бы одна, но для технологии поиска это различие не имеет принципиального значения.

Обратите внимание, что в программе обрабатываются четыре направления, несмотря на то, что одно из направлений — это направление-источник (клетка, из которой мы попали в данную), его рассматривать в качестве кандидата даже нет смысла. Но такой подход позволяет не запоминать клетку-источник и избавляет нас от существенных технических сложностей.

Текст программы приведен в листинге 21.1.

Листинг 21.1

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  doska:array[1..19,1..19] of byte;
  i,x,y,n,j:integer;
function Control(x,y:integer):boolean;
var
 q:boolean;
function Return(x,y:integer):boolean;
begin
```

```
  case doska[x,y] of
   0: Return:=true;
   1: begin
       doska[x,y]:=3;
       Return:=Control(x,y);
       floodfill(20*x+10,20*y+10,1);
      end;
   2,3: Return:=false;
  end;
end;
begin
 setfillstyle(1,1);
 q:=false;
 if x-1>=1 then q:=Return(x-1,y);
 if not q then
  if y-1>=1 then q:=Return(x,y-1);
 if not q then
  if x+1<=19 then q:=Return(x+1,y);
 if not q then
  if y+1<=19 then q:=Return(x,y+1);
 Control:=q;
end;
begin
{ read(n);
 for i:=1 to 19 do
  for j:=1 to 19 do doska[i,j]:=0;
 for i:=1 to n do
  begin
   read(x,y);
   doska[x,y]:=1;
  end;
 read(n);
 for i:=1 to n do
  begin
   read(x,y);
   doska[x,y]:=2;
  end;
 read(x,y);}
 doska[12,14]:=1; doska[12,13]:=1;doska[13,13]:=1;
 doska[14,13]:=1; doska[15,13]:=1;doska[15,14]:=1;
 doska[15,15]:=1; doska[16,16]:=1;doska[16,17]:=1;
 doska[16,18]:=1; doska[16,19]:=1;doska[15,18]:=1;
```

```
 doska[14,18]:=1; doska[13,18]:=1;doska[13,17]:=1;
 doska[13,16]:=1; doska[14,16]:=1;doska[12,16]:=1;
 doska[12,15]:=1;

 doska[14,12]:=2;doska[13,15]:=2;doska[14,15]:=2;
 doska[13,14]:=2;doska[14,14]:=2;doska[11,14]:=2;
 doska[11,13]:=2;doska[11,12]:=2;doska[12,12]:=2;
 doska[13,12]:=2;doska[15,12]:=2;doska[16,12]:=2;
 doska[16,13]:=2;doska[16,14]:=2;doska[16,15]:=2;
 doska[15,16]:=2;doska[11,15]:=2;doska[11,16]:=2;
 doska[12,17]:=2;doska[14,17]:=2;doska[12,18]:=2;
 doska[12,19]:=2;doska[13,19]:=2;doska[14,19]:=2;
 doska[15,19]:=2;doska[15,17]:=2;doska[17,17]:=2;
 doska[17,16]:=2;doska[17,18]:=2;doska[17,19]:=2;
 x:=12;y:=14;
 dr:=detect;initgraph(dr,md,'');
 for i:=1 to 20 do
  begin
   line(i*20,20,i*20,400);
   line(20,i*20,400,i*20);
  end;
 for i:=1 to 19 do
  for j:=1 to 19 do
   if doska[i,j]>0 then
    begin
     setcolor(doska[i,j]);
     circle(20*i+10,20*j+10,5);
    end;
 setfillstyle(1,1);
 floodfill(20*x+10,20*y+10,1);
 doska[x,y]:=3;
 if Control(x,y) then outtextxy(400,300,'Эта группа живая')
 else outtextxy(400,400,'Эта группа не живая');
 readkey;
end.
```

Только что мы рассмотрели рекурсивный вариант решения. Его соль и самый главный пункт — координаты текущего камня, порождающего рекурсивные вызовы для анализа его соседей. В рекурсивном варианте эти координаты хранятся в стеке. Для создания нерекурсивного варианта необходимо продумать организацию стека из координат. Наиболее естественной структурой данных для этих целей, конечно, будет массив. А так как координат две, то речь пойдет о массиве записей из двух компонентов `x` и `y`.

Для того чтобы массив работал как стек, введем специальную переменную величину (назовем ее вершиной стека), всегда равную длине массива. Она (эта переменная) обеспечит нам доступ к верхушке стека. В процессе обхода позиции при обнаружении камня занесение его координат в массив может быть выполнено двумя операциями:

- вершина стека увеличивается на 1;
- координаты камня записываются в элемент массива с индексом, равным вершине стека.

Очередной камень для обработки всегда берется с вершины стека. После обработки длина массива (стека) соответственно уменьшается на 1. Анализ позиции прекращается в двух случаях.

- На каком-то шаге обработки было обнаружено пустое соседнее поле. Даже одноразового такого события достаточно для завершения анализа. В таком случае можно сделать вывод, что исследуемая группа живая.
- Длина массива (стека) стала равна нулю. Это означает, что исследованы все камни группы и ни разу не было обнаружено пустое поле. Следовательно, в этом случае вполне можно сделать вывод, что исследуемая группа не есть живая.

Нерекурсивный вариант решения — листинг 21.2.

Листинг 21.2

```
program example;
 uses crt,graph;
 var
  dr,md:integer;
  doska:array[1..19,1..19] of byte;
  mas:array[1..300] of record
                        x,y:integer;
                       end;
  i,x,y,n,j,len:integer;
  res:boolean;
function Control(x,y:integer):boolean;
var
 i:integer;
 q:boolean;
begin
 case doska[x,y] of
  0:Control:=true;
  1:begin
     q:=true;
     for i:=1 to len do
       if (mas[i].x=x) and (mas[i].y=y) then q:=false;
```

```
     if q then
      begin
       len:=len+1;
       mas[len].x:=x;
       mas[len].y:=y;
       floodfill(20*x+10,20*y+10,1);
      end;
     Control:=false;
    end;
  2,3:Control:=false;
 end;
end;
begin
{ read(n);
 for i:=1 to 19 do
  for j:=1 to 19 do doska[i,j]:=0;
 for i:=1 to n do
  begin
   read(x,y);
   doska[x,y]:=1;
  end;
 read(n);
 for i:=1 to n do
  begin
   read(x,y);
   doska[x,y]:=2;
  end;
 read(x,y);}
 doska[12,14]:=1; doska[12,13]:=1;doska[13,13]:=1;
 doska[14,13]:=1; doska[15,13]:=1;doska[15,14]:=1;
 doska[15,15]:=1; doska[16,16]:=1;doska[16,17]:=1;
 doska[16,18]:=1; doska[16,19]:=1;doska[15,18]:=1;
 doska[14,18]:=1; doska[13,18]:=1;doska[13,17]:=1;
 doska[13,16]:=1; doska[14,16]:=1;doska[12,16]:=1;
 doska[12,15]:=1;

 doska[14,12]:=2;doska[13,15]:=2;doska[14,15]:=2;
 doska[13,14]:=2;doska[14,14]:=2;doska[11,14]:=2;
 doska[11,13]:=2;doska[11,12]:=2;doska[12,12]:=2;
 doska[13,12]:=2;doska[15,12]:=2;doska[16,12]:=2;
 doska[16,13]:=2;doska[16,14]:=2;doska[16,15]:=2;
 doska[15,16]:=2;doska[11,15]:=2;doska[11,16]:=2;
 doska[12,17]:=2;doska[14,17]:=2;doska[12,18]:=2;
 doska[12,19]:=2;doska[13,19]:=2;doska[14,19]:=2;
 doska[15,19]:=2;doska[15,17]:=2;doska[17,17]:=2;
```

```
 doska[17,16]:=2;doska[17,18]:=2;doska[17,19]:=2;
 x:=12;y:=14;
 dr:=detect;initgraph(dr,md,'');
 for i:=1 to 20 do
  begin
   line(i*20,20,i*20,400);
   line(20,i*20,400,i*20);
  end;
 for i:=1 to 19 do
  for j:=1 to 19 do
   if doska[i,j]>0 then
    begin
     setcolor(doska[i,j]);
     circle(20*i+10,20*j+10,5);
    end;
 setfillstyle(1,1);
 floodfill(20*x+10,20*y+10,1);
 mas[1].x:=x;mas[1].y:=y;
 len:=1;res:=false;
 while (len>0) and (not res) do
  begin
   x:=mas[len].x;y:=mas[len].y;
   doska[x,y]:=3;
   len:=len-1;
   if x-1>=1 then res:=Control(x-1,y);
   if not res then
     if y-1>=1 then res:=Control(x,y-1);
   if not res then
     if x+1<=19 then res:=Control(x+1,y);
   if not res then
     if y+1<=19 then res:=Control(x,y+1);
  end;
 if res then outtextxy(400,400,'Эта группа живая') else out-
textxy(400,400,'Эта группа не живая');
 readkey;
end.
```

В заключение. Глава в основном посвящена сравнению рекурсивного и нерекурсивного решений, несмотря на то, что мы уже неоднократно приводили два типа решений на одну и ту же задачу. Данный пример действительно не первый, но с точки зрения автора он наиболее ярко показывает проблему номер один — разработку конструкции стека. Но в этой задаче мы столкнулись и с не менее интересной находкой. Мы увидели неожиданную аналогию с задачей о закраске контура. Этот пример показывает, что аналогия не есть похожесть условий.

Глава 22

Поиск пути с наибольшим весом

Условие задачи. Дана таблица положительных целых чисел, сформированная случайным образом. Необходимо найти путь из левого верхнего угла таблицы в правый нижний с наибольшим весом. Весом пути будем называть сумму чисел всех элементов таблицы, через которые проходит путь. Путь можно строить только двумя типами смещений: вправо на один шаг или вниз на один шаг.

Данная задача имеет решение в виде полного перебора. Достаточно придумать алгоритм построения всех путей и в процессе построений вес каждого полученного пути сравнивать с уже найденным максимальным. Это элементарный алгоритм поиска наибольшего. Если таблица невелика, то скорость работы алгоритма окажется вполне приемлема. Например, это будет вполне реально для таблицы 10×10, которую обрабатывает наша программа. Если размеры таблицы увеличить, то количество возможных путей начнет стремительно увеличиваться, и соответственно скорость работы программы стремительно падать.

В случае совершенно произвольного построения путей задача становится задачей полного перебора, а проблема быстрого роста вариантов не решается. Однако в нашей задаче есть существенное ограничение на способ построения путей. Попробуем провести анализ дополнительного условия. Может быть, это даст возможность упрощения задачи.

Для хорошего метода нужна закономерность. Но наша таблица заполнена случайными числами. Искать закономерность на множестве случайных чисел — дело неблагодарное. По определению случайного числа среди такого множества закономерности быть просто не должно.

Но можно попробовать построить дополнительную конструкцию, такую что закономерность будет видна явным образом.

Примечание

Грамотное построение дополнительных структур часто дает дополнительные возможности в организации логики. Этим подходом (или точнее приемом) мы уже пользовались. Например, в задаче о разбиении кучи камней на две было проведено упорядочивание множества камней в порядке возрастания. Введение порядка на множестве — это то же построение дополнительной конструкции. В задаче экономного обхода графа и задаче заливки контура мы применяли флаг, содержащий полезную информацию. Два упомянутых случая использовали дополнительные конструкции различного типа. Введение порядка меняет общую структуру множества, добавление флага создает возможность введения для каждого элемента характеристики, описывающей ход процесса.

Нам нужна структура данных, привязанная к элементу таблицы и в то же время описывающая путь до этого элемента. "Структура" в этом случае звучит, наверное, излишне громко, достаточно будет числовой характеристики, описывающей наибольший вес пути до данного элемента. Если удастся придумать такую характеристику, то, видимо, это будет шаг в верном направлении.

Хорошая зацепка для последующего анализа в предложении, записанном ранее — *достаточно будет числовой характеристики, описывающей наибольший вес пути до данного элемента*. В этом предложении скрывается существенная неопределенность. Что есть *"путь до данного элемента"*? Если вы думаете, что имеется в виду путь от верхнего левого элемента до текущего, то спросите себя, ПОЧЕМУ ТАК? Не кажется ли вам, что такое убеждение происходит лишь от того, что движение осуществляется сверху вниз и справа налево! Характер движения совсем не означает, что поиск пути необходимо осуществлять в том же порядке.

Примечание

Это достаточно обычная ситуация, когда привычный или более естественный ход вещей выдается за логичный. В поисках решения у вас всегда есть серьезный риск попасть в плен к своим психологическим установкам. Поэтому старайтесь к каждому выдвинутому утверждению задавать вопрос ПОЧЕМУ?

Проанализируем обе ситуации. Пусть поиск пути ведется сверху, и на некотором шаге путник, анализирующий различные маршруты, находится в некоторой заданной точке. Сейчас он должен принять решение о дальнейшем движении. В его распоряжении есть информация о весе, накопленном в движении сверху (для удобства рассуждений не будем постоянно повторять, что движение идет не только сверху вниз, но и слева направо). Информации о том, что будет у него дальше, нет, поэтому не видно логических оснований для выбора пути.

Пусть теперь поиск пути ведется снизу вверх. Путник опять находится в определенной точке. Ситуация совершенно иная. Путник предельно точно знает, какой вес он смог набрать, и теперь выбор между направлением вверх и направлением влево теряет смысл, так как ясно, что к уже достиг-

нутому весу необходимо добавить большее число, выбранное из левого или верхнего элемента. Такой у нас получился парадокс. Движение должно осуществляться сверху вниз, а поиск пути логичнее вести снизу вверх.

Продолжим анализ. Принципиально идея решения уже есть. Она заключается в следующей фразе *"путник предельно точно знает, какой вес он смог набрать"*. Следовательно, мы должны для каждого элемента выяснить, какой вес можно набрать, двигаясь снизу до данного элемента. Теперь маленький интуитивный рывок и формулируем алгоритм.

Обойдя всю матрицу снизу вверх и справа налево, определим для каждого элемента новый вес, равный сумме собственного и наибольшего из верхнего и левого элемента. Алгоритм такой обработки:

```
Текущий элемент = Правый нижний
Пока текущий элемент не есть левый верхний, делать
  Если вес левого элемента больше веса верхнего
    То Текущий элемент = Левый элемент
    Иначе Текущий элемент = Верхний элемент
```

Искомый путь на построенной вспомогательной конструкции будет найден следующим алгоритмом:

```
Текущий элемент = Верхний левый.
Пока текущий элемент не есть Нижний правый, делать
  Если вес правого элемента больше веса нижнего,
  То Текущий элемент = Правый элемент
  Иначе Текущий элемент = Нижний элемент
```

Далее приведен пример такого построения и жирным шрифтом отмечен искомый путь. Для построения примера мы взяли клетку 5 на 5. Слева исходная таблица, справа преобразованная. Путь отмечен на преобразованной таблице.

1	2	8	3	5
3	2	2	1	45
5	1	1	0	4
2	3	4	3	8
1	1	3	4	8

1480	**816**	**419**	190	70
663	395	**221**	**117**	**65**
265	172	102	51	**20**
88	69	50	31	**16**
17	16	15	12	**8**

В листинге 22.1 приведен текст программы.

Листинг 22.1

```
program example;
 uses crt;
 var
```

```
  a:array[1..10,1..10] of longint;
  i,j:integer;
begin
 clrscr;
 randomize;
 for i:=1 to 10 do
  for j:=1 to 10 do
   begin
    a[i,j]:=random(9);
    gotoxy(i*7,j*4);write(a[i,j]);
   end;
 a[2,8]:=5360; gotoxy(2*7,8*4);write(a[2,8]);
 readkey;
 for i:=10 downto 1 do
  for j:=10 downto 1 do
    begin
     if i<10 then a[i,j]:=a[i,j]+a[i+1,j];
     if j<10 then a[i,j]:=a[i,j]+a[i,j+1];
     gotoxy(i*7,j*4);write(a[i,j]);
    end;
 readkey;
 i:=1;j:=1;
 textcolor(2);
 repeat
  gotoxy(i*7,j*4);write(a[i,j]);
  if (i=10) then j:=j+1
   else if (j=10) then i:=i+1
    else  if a[i+1,j]>a[i,j+1] then i:=i+1 else j:=j+1;
  delay(64000);
 until (i=10) and (j=10);
 gotoxy(i*7,j*4);write(a[i,j]);
 readkey;
end.
```

В заключение. Решенная задача — редкий пример того, как в переборной задаче удалось полностью избавиться от полного перебора. Такая масштабная удача говорит о том, что оценка данной задачи, как задачи полного перебора, была неверной. И действительно. Необходимость полного перебора следует из совершенного отсутствия информации об исходных данных. В рассмотренной задаче это не совсем так. Числа, которыми заполняется таблица, действительно случайные, но в построении путей полного произвола нет, именно существующее ограничение на возможные пути и создало возможность построения красивой вспомогательной конструкции.

Предметный указатель

(окончание рубрики на стр. 194)