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