

Потопахин В. В.

ИСКУССТВО АЛГОРИТМИЗАЦИИ



Москва, 2011

УДК 004.421
ББК 32.973.26-018
П64

Потопахин В. В.

П64 Искусство алгоритмизации. – М.: ДМК Пресс, 2011. – 320 с.: ил.

ISBN 978-5-94074-608-9

Эта книга для тех, кто хорошо, владея языком программирования и устойчивыми навыками решения задач, желает наработать свой программистский инструментарий. В книге, неформально и довольно детально, разобран значительный набор алгоритмов и методов. Большая часть представленных алгоритмов доведена до реализации на языке Компонентный Паскаль. Для большей прозрачности изложения реализация выполнена пошагово с четкой формулировкой задач каждого шага и записью программного фрагмента. Изложение сопровождается заданиями для самостоятельной работы, количество и сложность которых достаточны для хорошего усвоения материала. Требования к математическим знаниям минимальны, некоторые важные математические понятия и темы кратко изложены в приложении.

К изданию прилагается CD, на котором находится бесплатная среда программирования Блэкбокс, запустив которую, вы сразу сможете начать работу, а также сборник листингов к книге.

УДК 004.421
ББК 32.973.26-018

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-94074-608-9

© Потопахин В., 2010
© Оформление, издание, ДМК Пресс, 2011

Содержание

Введение	6
Глава 1. Парадигма структурного программирования	9
Зачем нужны общие принципы?	10
Нисходящее проектирование	12
Три базовых элемента структурного программирования	14
Пример разработки	17
Глава 2. Вычислительные алгоритмы	26
Моделирование непрерывных процессов дискретными	27
Метод половинного деления. Общая задача поиска величины	31
Метод касательных	34
Метод хорд	35
Метод итераций (последовательных приближений)	36
Обобщение метода половинного деления	37
Метод наименьших квадратов	38
Задача вычисления площадей криволинейных фигур	42
Метод Симпсона	45
Метод Монте-Карло	48
Глава 3. Числовые алгоритмы	54
Алгоритм Евклида	55
Алгоритмы факторизации и поиска простых	57
Выделение полного квадрата (алгоритм Ферма)	58
Квадратичное решето	60
Алгоритм Полларда	66
Алгоритмы поиска простых чисел	69
Решето Аткина	71
Решето Сундарама	72
Тесты простоты	73
Числа Мерсенна	75
Тест Люка-Лемера	76
Числа Ферма	78
Тест Пепина	78
Псевдослучайные числа	78
Критерии правильности случайных чисел	81
Критерий, основанный на квадратичном отклонении	81
Линейный конгруэнтный метод	81
Методы перемешивания	85

Глава 4. Арифметика	89
Представление числа в позиционной системе счисления	90
Проблемы технической реализации арифметики	93
Двоичный сумматор	94
Ускорение операции сложения	95
Представление чисел в форме с фиксированной и плавающей десятичной точкой	96
Реализация арифметики на уровне алгоритмического языка	97
Сложение двух чисел	97
Вычитание из большего меньшего	99
Умножение	102
Деление	107
Некоторые другие алгоритмы	115
Алгоритм быстрого возведения в степень	115
Быстрый перевод из десятичной в двоичную систему счисления	116
Решение диофантовых уравнений	117
Двоичная арифметика	119
Сложение двоичных чисел	120
Как преобразовать в двоичное число дробную часть	122
Вычитание двоичных чисел	124
Умножение в двоичной системе счисления	125
Деление в двоичной системе счисления	126
Глава 5. Рекурсия и динамическое программирование	131
Общее определение	132
Задача о ханойской башне	135
Переход от рекурсивного к нерекурсивному решению	138
Рекурсия как метод поиска	143
Динамическое программирование	144
Задача обхода конем шахматной доски	146
Факторизация числа	153
Глава 6. Сортировки	166
Общая постановка задачи	167
Обменные сортировки. Сортировка пузырьком	168
Шейкерная сортировка	170
Анализ качеств алгоритма	171
Сортировка выбором	174
Сортировка вставками	176
Сортировка Шелла	178
Быстрая сортировка	181
Двоичная сортировка	186
Сортировка слияниями	191

Глава 7. Комбинаторные задачи	204
Общая постановка задачи	205
Оптимизация перебора	207
Связь комбинаторики с алгоритмами на графах	209
Основные комбинаторные задачи	210
Задача получения перестановок на множестве из N элементов	210
Построение сочетаний без повторений на множестве элементов	216
Сочетания с повторениями	221
Задача получения размещений	223
Глава 8. Динамические структуры данных	224
Понятие о динамической величине	225
Линейный связный список	226
Зачем рекурсивные структуры нужны?	229
Использование рекурсивных определений для создания деревьев данных	233
Глава 9. Алгоритмы принятия решений	237
Постановка задачи. Понятие эвристического алгоритма	238
Оценочная функция	240
Метод минимакса	241
Альфа-бета алгоритм	245
Глава 10. Алгоритмы на графах	250
Стратегии обхода	251
Обход графа в ширину	251
Обход графа в глубину	253
Построение остовного дерева	253
Алгоритм Прима	254
Алгоритм Краскала	258
Алгоритм поиска компонент связности	263
Волновой алгоритм	265
Алгоритм Дейкстры	269
Алгоритм Флойда	276
Нахождение максимального потока	280
Глава 11. Приложения	296
Приложение 1. Элементы комбинаторики	297
Приложение 2. Теория графов	301
Приложение 3. Элементы теории вероятности	309
Приложение 4. Синтаксис языка Компонентный Паскаль	315
Список литературы	319

Введение

Книга, которую вы держите в руках, является логическим продолжением книги «Современное программирование с нуля» того же автора. Но если упомянутая книга была посвящена выработке базовых программистских умений, то сейчас цель – наработка инструментария программиста-профессионала. Если вы в программировании совсем новичок, то придется эту книгу пока отложить и заняться приобретением базовых навыков: необходимо уверенно владеть языком Паскаль, желательно его последней версией Компонентный Паскаль, и совершенно необходим хороший навык написания хотя бы несложных программ.

Основное содержание книги – алгоритмы и некоторые интересные задачи. Исключение составляет только первая глава, посвященная принципиальному вопросу: что такое хорошо написанная программа. Это, может быть, покажется неинтересным, но постарайтесь все же первую главу прочитать максимально внимательно. Структурное программирование, которому полностью посвящена первая глава, есть форма дисциплины мышления программиста. А недисциплинированный программист обречен на неудачу независимо от того, каким набором алгоритмов, технологий и языков программирования он владеет.

Все остальные главы посвящены искусству алгоритмизации. Часть материала потребует некоторых математических знаний. Большую часть требуемой математики вы сможете найти здесь же, но без строгих доказательств и детального изложения. Поэтому книга довольно самодостаточна, а для желающих углубить свои знания по тому или иному вопросу даны ссылки на специальные источники.

Язык изложения – Компонентный Паскаль. Для примеров практически не используются какие-либо библиотечные модули, применяемые средства максимально просты, если вы имеете хороший языковой опыт, однако не знаете КП, текст не станет для вас слишком непонятным. Но все же книга будет более читаемой, если вы как следует усвоите Компонентный Паскаль.

Наверное, главная особенность стиля изложения – это детальность разработки примеров. Только некоторые совсем уж простые задачи даны кратко, большая их часть снабжена пошаговым разъяснением всех деталей реализации. Если алгоритм сложен, то его объяснение снабжается примерами использования, иллюстрациями. Выбор реализации алгоритмов сделан автором в пользу прозрачности и понятности, быть может, иногда за счет потери некоторой части эффективности. Уровень завершенности реализации учебных примеров разный. Для некоторых примеров дан только фрагмент программы, но таких мало. Для некоторых примеров написан текст процедуры, которую еще надо оформить в какой-то модуль. И есть примеры, полностью завершенные (до модуля). Впрочем, если текст примера в книге дан только в виде процедуры, то в приложении на диске он скорее всего представляет собой полностью завершенную программу. В текстах примеров активно используются идентификаторы на русском языке для большей эффективности объяснения. Поэтому если у вас нет желания переписывать иденти-

фикаторы латиницей, то воспользуйтесь сборкой BlackBox с диска, прилагаемого к книге.

В тексте книги много заданий для самостоятельной работы. Все задания логически вытекают из хода изложения. Это могут быть теоретические вопросы о свойствах исследуемых алгоритмов, это могут быть предложения по улучшению реализации или идеи несколько иной реализации того же алгоритма. Немного пройдемся по главам.

Первая глава посвящена основным идеям структурного программирования. Глава очень краткая, изложена, если можно так сказать, самая суть метода. Здесь необходимо указать, что вопросы методологии всегда были и будут самыми спорными, поэтому, возможно, кто-то не согласен с такой структурой рассказа, очевидно, что вопросы структурного программирования и нисходящего проектирования можно излагать по-разному. Но перед книгой не ставилась цель исчерпывающего анализа этой сложнейшей темы, кроме того, в тексте главы будут ссылки на авторитетных авторов и классические книги.

Вторая глава – о вычислительных алгоритмах, это о том, как поступать в ситуациях, когда нет возможности выполнить расчет подстановкой в простую формулу. Оказывается, в реальных задачах очень часто приходится прибегать к так называемым численным методам, которые и составляют содержание второй главы.

Третья глава – это рассказ о числовых алгоритмах. Объяснены некоторые часто используемые вещи, например алгоритм Евклида, решето Эратосфена, и часть времени посвящена достаточно увлекательным задачам, не имеющим на сегодня исчерпывающего решения, – это задача факторизации, задача получения больших простых, задача построения последовательности псевдопростых чисел.

Четвертая глава – это арифметика. Мы все привыкли, что электронные устройства умеют выполнять арифметические операции, но ведь это тоже проблемы алгоритмизации. В главе рассмотрены некоторые вопросы, возникающие при программировании арифметики. Приведены реализации выполнения операций столбиком, и рассказано о некоторых возможностях усиления арифметических алгоритмов.

Пятая глава – рассказ о рекурсии. Даны определение и основные свойства. Рассказано, как строится рекурсивный процесс, какие при этом возникают проблемы. Вводится представление о динамическом программировании. Решены несколько несложных задач, и завершается глава двумя достаточно серьезными задачами: задачей обхода конем шахматной доски и еще одним алгоритмом факторизации, которого нет в главе о числовых алгоритмах.

Шестая глава. Сортировки. Рассмотрены: пузырьковая сортировка, шейкерная, сортировка подсчетом, сортировка вставками, выбором, быстрая, двоичная, сортировка Шелла, сортировка слияниями и естественными слияниями. Начинается глава общей постановкой задачи, в ходе изложения кратко анализируются свойства сортировок.

Седьмая глава. Комбинаторные задачи. Дано представление о том, что такое вообще комбинаторная задача, обсуждены проблема комбинаторного взрыва и возможности построения эвристического решения. Приведены реализации не-

которых базовых, комбинаторных задач: построение перестановок, сочетаний, с повторениями и без. Даны рекурсивные и нерекурсивные решения.

Восьмая глава. Динамические структуры данных. Это небольшой рассказ о том, что такое динамические величины и, главным образом, что такое рекурсивно определяемые величины, описаны некоторые операции над связными списками и деревьями.

Девятая глава – о том, как написать программу, умеющую принимать решения. Разъяснены основные составляющие такой программы: оценочная функция, минимаксный обход дерева вариантов, некоторые возможности его сокращения. Дано определение эвристического алгоритма, указаны проблемы, возникающие при попытке сократить дерево перебора.

Последняя, десятая глава – самая технически сложная – алгоритмы на графах. Рассмотрены следующие задачи: построение остовного дерева, построение компоненты связности, поиск кратчайшего пути волновым алгоритмом, поиск наиболее дешевых путей алгоритмами Дейкстры и Флойда, построение максимального потока алгоритмом Форда-Фалкерсона.

Завершена книга несколькими приложениями, кратко излагающими основные понятия комбинаторики, теории графов и теории вероятностей. Это на тот случай, если вашей математической подготовки окажется недостаточно.

Парадигма структурного программирования

Зачем нужны общие принципы.....	10
Нисходящее проектирование.....	12
Три базовых элемента структурного программирования.....	14
Пример разработки.....	17

Зачем нужны общие принципы?

Попробуем понять, что такое хорошая программа. Очевидно, этот вопрос надо решить дважды: во-первых, с точки зрения пользователя, так как программа в конечном итоге делается для него, и во-вторых, с точки зрения программиста, так как он, наверное, не может быть безразличен к качеству своего продукта. Пользователь программного обеспечения ожидает, что результат будет получен гарантированно и за приемлемое время. Еще одно, важное требование – это стоимость ПО. Если пользователь платит за программу, то естественно, он желает, чтобы стоимость была по возможности низкой. Это почти все. Различные программистские вопросы, вроде того, на каком языке написана программа, как она устроена, сколько в её разработку вложено программистской энергии, его не интересуют.

Важное замечание. Есть один исключительно важный момент, выпадающий из логики дальнейшего изложения, но пропустить который нельзя. Это вопрос надежности ПО. Грубо говоря, нажимая на кнопку, мы ожидаем вполне определенный результат, а не какой-нибудь. Иногда это вопрос больших материальных потерь и даже человеческих жизней. И надежность обеспечивается отнюдь не талантом программиста, а скорее выбором и строгим соблюдением общих принципов и правил.

Позиция программиста в вопросе оценки качества программы куда более сложная. Например, программист желает получить заданный результат с минимальными усилиями. Отчасти из такого желания и возникает понятие технологии. В промышленности из желания делать много малыми усилиями возникли разделение труда и конвейер. В программировании результатом такого желания можно считать понятие процедуры, коллекционирование процедур в библиотеки и сборку новой программы из уже готовых процедур. С этой точки зрения задачу программирования можно сформулировать так:

Определите, какие процедуры необходимы, и соберите из них новую процедуру, реализующую поставленную задачу.

Так звучит парадигма процедурного программирования. Конечно же компоновать программу из процедур и языковых конструкций можно по-разному. Процедурная парадигма ничего не говорит о том, как организовать процесс программирования, она лишь определяет процедуру главным строительным блоком. Вопрос, что такое хорошо и что такое плохо, с этой позиции остается открытым.

Анализ с точки зрения функциональности программы мы опустим, это можно отнести к ожиданиям пользователя, разговор о которых уже закрыт. С точки же зрения программиста существуют еще два важных фактора: скорость написания программы и её читаемость. Читаемость – собственно то же фактор скорости, скорости понимания программы.

Конечно, лучше всего было бы быстро писать и легко читать. Но, к сожалению, в реальном программировании приходится искать золотую середину, отдавая предпочтение либо скорости написания, либо читаемости. Что важнее? На первый взгляд может показаться, что скорость важнее. Чем быстрее мы напишем программу, тем меньше потратим времени и энергии. Но это только на первый

взгляд. Во-первых, любой серьезный проект не пишется за одну попытку. Сначала создается его первая, рабочая версия, которая затем развивается и улучшается, быть может, до бесконечности. Для развития и улучшения программа должна читаться, и иногда не теми людьми, которые писали первую версию. Следовательно, все-таки программа должна быть читаемой.

Даже если программу пишет один программист, то и он делает её не за один присест. Скорее всего, на разработку уходит значительное время, в течение которого у программиста может возникнуть потребность вернуться к тому или иному фрагменту, что-то вспомнить, что-то уточнить. То есть и для автора первой версии программа должна быть читаемой. Сказанное можно выразить так:

Программа пишется один раз, а читается многократно. Поэтому читабельная программа сокращает время разработки значительно эффективнее, чем «быстро написанная».

Необходимо ответить на вопрос: а как писать код, чтобы он был легко читаем? Взглянем на программу немного с другой стороны. Исполнение программы можно рассматривать как процесс передачи управления, а текст – соответственно, как описание последовательности передачи управления. Отсюда следует простая и естественная идея. Читаемая программа – это программа с максимально простой последовательностью передачи управления. Наиболее простая структура управления – это линейная последовательность программных блоков. Для того чтобы можно было выстроить линейную последовательность, необходимо и достаточно, чтобы каждый логически замкнутый блок имел один вход и один выход. Программа, построенная из таких блоков, имеет, очевидно, структуру простейшую из возможных, и такая программа называется структурной.

Структурность может быть, и как правило, бывает многоуровневой. Блоки линейной структуры могут оказаться достаточно крупными. В этом случае каждый такой блок должен допускать представление в виде линейной структуры блоков меньшего размера и т. д.

Почему программа, устроенная таким образом, будет читаемой? Ответ, видимо, лежит в области психологии человеческого восприятия. Когда мы смотрим на картину, то сначала воспринимаем крупный план, затем детали. Также работает и наше мышление. Есть крупный план проблемы, есть подзадачи, которые можно рассматривать последовательно одну за другой. Также анализируется и программа. Сначала крупный план – программа как последовательность логически законченных блоков, затем каждый блок – процедура как самостоятельная задача.

Идея структурирования программы последовательностью логически замкнутых блоков выглядит естественной. Поэтому может возникнуть вопрос зачем вообще об этом говорить. Есть много естественных вещей, и им никто не учит. Ведь действительно, никто не будет строить дом с крыши. Однако в программировании ситуация несколько хитрее. Естественно не значит просто и совершенно не означает, что структурное программирование дается каждому с легкостью. Структурное программирование предполагает дисциплинированный ум, умеющий хорошо ор-

ганизовать свою работу. Однако дисциплина ума, технологичность мышления – это то, что подлежит развитию. Поэтому структурное программирование – это не просто свод правил, как правильно писать программу, это прежде всего некоторая технология мыслительной деятельности.

Нисходящее проектирование

Сейчас речь пойдет не просто о написании программы. Речь пойдет о поиске решения задачи, и написание программы в этом процессе есть только этап, пусть и наиболее ощутимый, но все же только этап. Решение же предполагает многие вещи. Например, поиск математического решения, выбор структур данных, разработку алгоритмов. Это замечание нам нужно для того, чтобы обезопасить себя от узкого понимания вопроса „что значит решить программистскую задачу?” К сожалению, очень часто это действие сводят к написанию некоторого кода, что, конечно, неправильно.

А сейчас главная идея. Большая программистская удача заключается в том, что любая достаточно серьезная задача не представляет собой логически монолитного куска гранита. Скорее, это набор камней, уложенных в определенную конфигурацию. А если не увлекаться аналогиями, можно сказать, что программистская задача допускает разбиение на подзадачи, каждая из которых формулируется независимо от Большой Задачи и от других подзадач и соответственно решается так, как будто других подзадач и Большой Задачи просто не существует. После решения всех подзадач решение Большой Задачи komponуется из полученных меньших.

Такой процесс разбиения называется **ДЕКОМПОЗИЦИЕЙ**. Декомпозиция может быть многоуровневой, каждая из полученных подзадач также может оказаться достаточно большой, и тогда к ней тоже можно применить операцию декомпозиции.

Последовательное применение операции декомпозиции к подзадачам различного уровня называется нисходящим проектированием.

Рассмотрим простой пример. Дано уравнение вида

$$\sum_{k=0}^n a_k x^k = 0.$$

Определить все его целые корни.

Решение. Алгебра дает нам нужный для решения факт: все целочисленные корни являются делителями свободного члена, то есть величины a_0 . Следовательно, решение Большой Задачи должно заключаться в переборе всех делителей числа a_0 и выяснении, какие из них являются корнями уравнения. Отсюда и очевидное разбиение:

Задача 1. Дано некоторое целое число. Найти все его делители.

Задача 2. Дан многочлен вида

$$\sum_{k=0}^n a_k x^k = 0$$

и некоторое значение x , вычислить значение многочлена.

Легко увидеть, что сформулированные задачи логически независимы. Вторая задача нужна для расчета значения многочлена от делителя, но то, что число x является делителем, ровным счетом никого ни к чему не обязывает. Если мы напишем процедуру, считающую значение многочлена от любого x , то, естественно, мы сможем вычислить и его значение от x , являющегося делителем.

Предположим, что используемая среда программирования не умеет считать степень числа, тогда есть смысл вторую подзадачу разбить еще на две:

Задача 2.1. Вычислить сумму ряда $A_1 + A_2 + \dots + A_n$.

Задача 2.2. Вычислить величину $A_k = a_k x^k$.

Но, в общем-то, мы выполнили декомпозицию второго уровня только для того, чтобы показать, что это возможно. Глубина декомпозиции должна определяться прежде всего из соображений разумности. Подзадача вычисления степени величины, даже для не слишком опытного программиста, не должна создавать проблем, и специально её проектировать нет необходимости. Задача поиска делителей числа решается так:

Листинг 1.1

```
FOR x:=1 TO B DO
  IF B MOD x=0 THEN
    StdLog.Strng('Очередной делитель =');StdLog.Int(x);
  END;
END;
```

Данный фрагмент решает поставленную задачу. Предположим теперь, что вторая задача также решена и реализована в виде функции **Расчет**. Для того чтобы готовый фрагмент объединить с функцией **Расчет**, необходимо принять решение о том, что функция получает на входе и что она дает в качестве результата. На выходе необходимо сообщение, является ли некоторое число корнем. Поэтому разумно возвращать из **Расчет** логическое значение (TRUE – корень, FALSE – не корень). Для работы функции потребуется массив коэффициентов многочлена и значение x . Величина x в нашем фрагменте определена, единственное – уточним, что проверке подлежат два числа: x и $-x$. И фрагмент можно переписать так:

Листинг 1.2

```
FOR x:=1 TO B DO
  IF B MOD x=0 THEN
    IF Расчет (a,x) THEN
```

```
StdLog.String('Очередной корень =');StdLog.Int(x);  
END;  
IF Расчет (a,-x) THEN  
  StdLog.String('Очередной корень =');StdLog.Int(-x);  
END;  
END;  
END;
```

Обратите внимание, мы использовали процедуру – функцию **Расчет**, не написав для неё ни одной строчки кода. Это означает, что две наши подзадачи действительно независимы. Первая подзадача использует вторую, но ей все равно, как вторая устроена внутри. Является ли написанный фрагмент структурным? Да, безусловно. Его верхний уровень – это один цикл. Следующий уровень – это единственный оператор IF, записанный в структуре цикла, и наконец, тело оператора IF **В MOD x=0 THEN** – две последовательно выполняемые проверки. Завершив написание фрагмента, программист вполне может приниматься за написание функции **Расчет**, не оборачиваясь на сделанное.

В наши цели входило только привести простой пример декомпозиции, поэтому доводить работу до завершённой программы не будем.

Три базовых элемента структурного программирования

Приведенный пример несет в себе еще один важный момент. Структурная программа – это последовательность блоков с одним входом и одним выходом, но представлять такой блок только как линейную последовательность операторов не содержательно. Любой язык программирования содержит в себе конструкции циклов и конструкции выбора. Этот факт отражен в структурной парадигме вводом трех видов структурных блоков. Первый (рис. 1.1) – это простой блок, второй (рис. 1.2) – это конструкция цикла, и третий (рис. 1.3) – конструкция выбора.

В своей простейшей форме простой блок – это последовательность операторов присваивания. В общем виде простой блок – это процедура или любой программный фрагмент, который можно отделить от остальной программы и определить для него только ему присущий смысл. Простой блок может иметь сколь угодно сложную внутреннюю структуру.

Существенно важно лишь то, что есть только одна точка в его тексте, в которой ему передается выполнение, и есть только одна точка, в которой его выполнение завершается.

Цикл – это сложная конструкция, описывающая процесс многократного выполнения простого блока. Обратите внимание: в блок-схеме речь идет об условии продолжения. Говоря в терминах Компонентного Паскаля, здесь описана форма цикла **WHILE**. Разумеется, это не означает недопустимости других форм. Необхо-

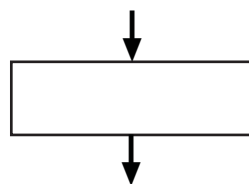


Рис. 1.1

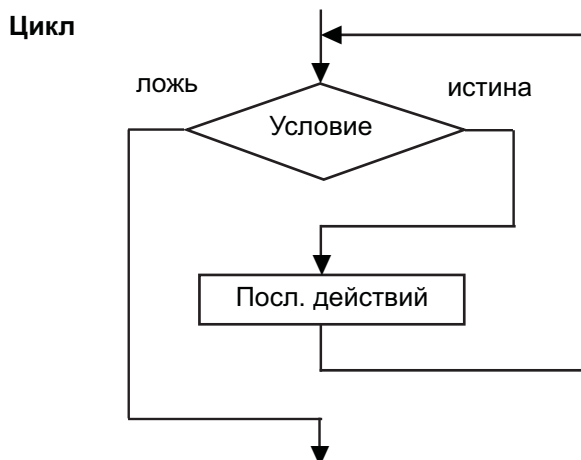


Рис. 1.2

можно понимать, что прежде всего здесь описана не определенная форма цикла, а принципиальная возможность многократного выполнения.

Для конструкции условного перехода (конструкции выбора), так же как и для цикла отметим, что любой существующий язык программирования предлагает существенно больше возможностей для организации ветвлений. Но любая форма ветвления представима в виде комбинации структур, описанных на рис. 1.3.

Условный переход

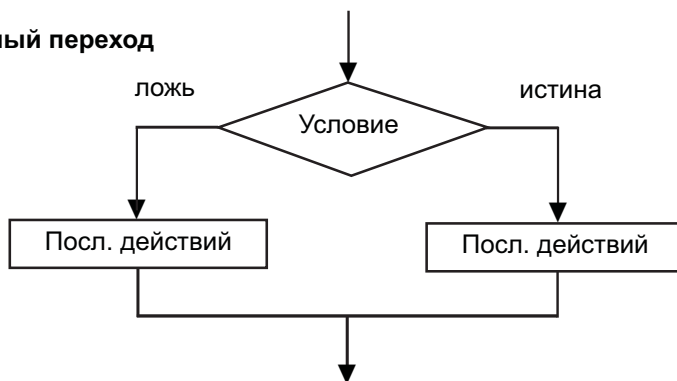


Рис. 1.3

Три приведенных своими блок-схемами конструкции и есть **единственные** блоки для построения структурных программ. Обратите внимание на выделенное слово. То, что эти блоки единственные, момент принципиальный.

Идея структурного программирования заключается в максимальной простоте при достаточной функциональности, именно поэтому структурные программы прозрачны для понимания и легко читаемы.

Конечно, можно искусственно ввести какие-то дополнительные конструкции для увеличения функциональности, но, наверное, вы согласитесь, что любая палка о двух концах. Потеря читабельности сведет на нет все достижения в области функциональности. Конечно же, сказанное не означает запрета на поиск новых, эффективных конструкций. Это лишь требование осторожности в таком поиске. Примеры изобретения более мощных структур конечно же есть. Например, так называемый цикл Дейкстры.

Листинг 1.3

```
WHILE условие цикла DO
    последовательность операторов
{
    ELSIF условие DO
        Последовательность операторов
}
END;
```

Фигурные скобки означают, возможно, многократное повторение конструкций ELSIF. Смысл цикла – в том, что сложная конструкция из вложенных циклов и проверок выстраивается на одном уровне, это если не упрощает логику алгоритма, то во всяком случае делает более прозрачной запись программного текста. В КП нет конструкции, непосредственно реализующей цикл Дейкстры, но его легко реализовать циклом LOOP:

Листинг 1.4

```
LOOP
    IF логическое выражение THEN
        Последовательность операторов
    {
        ELSIF логическое выражение THEN
            Последовательность операторов
    }
    ELSE EXIT
    END;
END;
```

Структурное программирование и нисходящее проектирование позволяют сделать процесс разработки ПО высокотехнологичной дисциплиной, что жизненно важно для программирования вообще и промышленного программирования в частности.

Как ни странно, талантливый программист, интуитивно находящий красивые решения, может оказаться фактором повышенной опасности для промышленного проекта. Дело в том, что программистская работа в массе своей является работой коллективной, и зачастую не столько важна красивость решения, сколько сроки, в которые оно получено, возможность его анализа, совместимость его с другими частями проекта. То есть работа программиста должна подчиняться строгой мыс-

лительной дисциплине, а структурное программирование вкупе с идеей нисходящего проектирования, по большому счету, и есть основа для такой дисциплины. Разумеется, сказанное ни в коем случае нельзя воспринимать как попытку принизить личный талант. Талантливый специалист всегда лучше бесталанного. Разумеется только то, что личный талант – это лишь основа, нуждающаяся в жестком дисциплинирующем каркасе.

Замечание об операторе GOTO. Альтернатива – структурное или неструктурное программирование при переходе к языку программирования отчасти переходит в альтернативу: использовать или не использовать оператор безусловного перехода GOTO. В языке Компонентный Паскаль этот оператор отсутствует, но, как правило, языки программирования проектируются с этим оператором, есть он и в диалектах языка Паскаль. GOTO – это главная возможность «грубого» прерывания хода исполнения программы и передачи управления почти в любое место её текста без проверки каких-либо условий. Этот оператор и обеспечивает возможность появления программных блоков с более чем одним входом и более чем одним выходом. Спор о том, нужен или нет оператор GOTO, не имеет большого смысла. Каждый программист должен ответить для себя на более фундаментальный вопрос: будет ли его программа структурной, если да, то потребности в GOTO просто нет. Если же программист допускает появление неструктурных элементов, то, естественно, он допускает и появление в своих программах GOTO.

Пример разработки

Для построения примера возьмем известную и очень сложную задачу разработки расписаний. Естественно, мы её ограничим и будем говорить не о любых расписаниях, а о расписании учебных занятий в учебном заведении.

Данная задача сложна не только с идейной стороны. Она очень трудоемка, и детальное её описание может потребовать слишком много места. Поэтому договоримся прежде всего об упрощении структуры данных и не будем требовать от алгоритма реального успеха в любых условиях. Мы будем использовать алгоритм достаточно разумный и достаточно интересный, но если кто решит попробовать довести его до коммерческой версии, он должен быть готов к очень большой работе.

Краткое описание идеи

Если говорить о структурах данных, на которых решается задача, то их две: во-первых, набор занятий, и, во-вторых, набор аудиторий, в которых занятия можно проводить. Занятие можно идентифицировать именем или номером. С аудиторией дело обстоит несколько сложнее. Объектом распределения является не сама аудитория, а время, в течение которого аудиторию можно занять. Договоримся отрезок времени, в течение которого аудитория может быть занята одним занятием, условно называть вакансией. Тогда вместо множества аудиторий есть смысл рассматривать множество вакансий.

Структура составляемого расписания должна удовлетворять некоторому набо-

ру требований. Эти требования можно разбить на два класса. Назовем их условно требования класса *A* и требования класса *B*. Требования класса *A* определяют принципиальное соответствие занятия и вакансии. Это, например, требование вместимости, аудитория должна иметь достаточное количество посадочных мест. Аудитория должна иметь требуемое оборудование, например для лекции нужна доска, для лабораторной работы – соответствующие приборы и т. д. Сколько таких требований описывается для расписания и как их формализовать, сейчас не важно. Заметим только, что полный набор требований составляет к каждому занятию, и набор этот определяет на множестве вакансий подмножество, которое назовем областью определения занятия. Соответственно, множество занятий, претендующих на данную вакансию, является областью определения вакансии.

Если бы требования класса *A* были единственными, то можно было бы сформулировать достаточные условия существования расписания и описать алгоритм, гарантирующий положительный результат, если таковой вообще возможен. А именно было бы достаточно для каждого занятия выполнить следующие операции:

1. Найти в области определения занятия вакансию, отсутствующую в областях определения еще не распределенных занятий.
2. Составить новую пару расписания.
3. Вычеркнуть распределенное занятие из областей определения всех свободных вакансий.

Возможна ситуация, когда первый пункт выполнить не удастся, но это не самая большая проблема. Ситуацию резко ухудшает наличие требований *B*. Это требования взаимного положения занятий в расписании. Например, у группы студентов не должно быть дыр (пропусков между занятиями). Это может быть требование чередования предметов. Нельзя в один день поставить все занятия по одному предмету, в другой – все занятия по другому и т. д.

Формализация такого рода требований – тема отдельного разговора. Мы отметим только, что требования группы *B* усложняют понятие области определения занятия. Придется ввести два понятия области определения. Первую область, сформированную требованиями *A*, назовем абсолютной. Эта область может быть рассчитана один раз, до начала процесса составления расписания, и она не меняется. Вторую область назовем текущей, это набор вакансий, доступных для занятия с точки зрения полного множества требований *B*. Ясно, что эта область определения имеет переменный размер, она может по ходу составления расписания как увеличиваться, так и уменьшаться. К сожалению, уменьшение более типично для такого процесса.

Претензии разных занятий на одни и те же вакансии и существование требований *B* приводят к тому, что при наличии большого количества свободных вакансий текущая область определения какого-то занятия может оказаться пустой. Такую ситуацию будем называть конфликтом. Введем понятия риска конфликта для занятия. Пусть D – это размер области определения для некоторого занятия. Тогда назовем величину $1/D$ риском. Ясно, что чем D меньше, тем величина риска

больше. При $D=0$ величина риска становится бесконечно большой, что и означает конфликт.

Главная идея. Пусть некоторое количество пар расписания уже построено. Рассчитаем риски для нераспределенных занятий и выполним сортировку в порядке убывания. Тогда очередным распределяемым занятием становится занятие, имеющее наибольшую величину риска. Идея очень естественная. Её разумность, наверное, очевидна, но борьба за эффективность потребует серьезной доработки.

Главную идею можно существенно дополнить. Размер области определения вакансии также является величиной, характеризующей степень риска. Если область определения вакансии велика, то, связав вакансию с занятием, мы уменьшаем области определения значительного количества занятий. Поэтому желательно привлекать к распределению вакансии с наименьшей областью определения.

Декомпозиция

Процесс составления расписания заключается в формировании пар (занятие, вакансия) и состоит из решения следующих подзадач:

- Расчет текущей области определения занятия.
- Расчет области определения вакансий.
- Сортировка множества занятий.
- Сортировка множества вакансий.
- Составление пар (занятие, вакансия).

И еще один важный пункт – выход из конфликтной ситуации. Анализируемый алгоритм позволяет уменьшить вероятность конфликтной ситуации на каждом шаге распределения, но вряд ли он дает возможность её избежать. Мы не будем сейчас заниматься разработкой алгоритма разрешения конфликтов, для наших целей достаточно понимания, что такой алгоритм необходим. Возможно, алгоритм разрешения конфликта будет сбрасывать часть расписания и как-то изменять ход последующего распределения, наверное, здесь возможны различные идеи. Но пора заняться написанием алгоритма. Ниже дан псевдокод, описывающий структуру алгоритма.

1. Рассчитать текущие области определения и величины рисков для каждого занятия. Величина риска рассчитывается, как величина обратная к величине области определения.
2. Рассчитать области определения для каждой вакансии и величины рисков. Величина риска равна размеру области определения.
3. Упорядочить множество занятий в порядке убывания рисков.
4. Упорядочить множество вакансий в порядке убывания рисков.
5. Взять для очередной пары расписания из множества занятий первое.
6. Выбрать из области определения взятого занятия вакансию, имеющую наименьший номер в списке вакансий. Построить пару расписания.

7. Пересчитать текущие области определения занятий.
8. Пересчитать текущие области определения вакансий.
9. Если в списке занятий есть занятие с пустой текущей областью определения, то вызвать процедуру разрешения конфликтов.
10. Если список занятий не пуст, то перейти на п. 3.
11. Расписание составлено. Работу завершить.

Мы выполнили декомпозицию исходной задачи. Получившийся в результате алгоритм представляет собой один большой цикл, в теле которого вызываются несколько процедур, каждая из которых реализует собой достаточно большую задачу. Процедуры вызываются последовательно, независимо друг от друга, их связь осуществляется через общий набор данных. Каждая из подзадач-процедур идентифицируется фразой, смысл которой обозначает цель подзадачи-процедуры, дальнейшую разработку каждой из подзадач могут вести разные программисты, единственное, что требуется – это точное описание структур данных.

Уточним понятие «Передача управления»

Передача управления от программного блока A к программному блоку B означает завершение работы блока A и начало работы блока B . Это во-первых. Во-вторых, передача управления означает завершение подготовки данных, необходимых для работы очередного программного блока. Работу программы можно таким образом воспринимать как передвижение набора данных между программными блоками. Тогда передачу управления можно понимать как передачу набора данных.

Замечание о процедурах и функциях

Иногда в учебниках программирования можно встретить понимание процедуры как средства, позволяющего многократно использовать код, записанный один раз. Это, конечно, правильное понимание, но, кроме того, процедура – это средство структурирования программы. В рассмотренном выше примере задачи составления расписания учебных занятий результат представляет собой набор процедур, ни одна из которых не вызывается дважды. Таким образом, имя процедуры – это некоторая метка, поясняющая, какая подзадача в данной точке текста программы должна быть решена. Это, в свою очередь, позволяет разделить процесс разработки на различные уровни абстракции с определенной последовательностью передачи управления и структур данных.

Еще один пример

Рассмотрим следующую задачу. Некий путешественник выходит из пункта A и следует в пункт B , находясь в глубоком тумане и имея компас, всегда показывающий на пункт B . Необходимо построить траекторию пути от A до B через поле, заполненное препятствиями, построенными из прямоугольников (препятствие может иметь достаточно сложную форму). Эта задача детально разобрана в [3] и [4]. Мы же сейчас используем её как ещё один пример декомпозиции. Обозначим текущие координаты путешественника через x, y , исходные координаты как A_x, A_y

и координаты пункта назначения через B_x , B_y . Тогда верхний уровень разработки дает следующее решение:

Вариант 1.

$x := A_x$; $y := A_y$;

WHILE ($x \neq B_x$) OR ($y \neq B_y$) DO

 Путешествие;

END;

В этом варианте предполагается, что процедура **Путешествие** смещает путешественника на один шаг за один свой вызов. В этом варианте текущие координаты разумно объявить как глобальные. Цикл проверки в принципе можно включить в тело процедуры и получить следующий вариант:

Вариант 2.

$x := A_x$; $y := A_y$;

Путешествие;

Но такая запись несодержательна, она не несет в себе никакой информации о решении, кроме того, что путь начинается из пункта A . Сказанное выделим еще раз, как важнейший принцип:

Каждый уровень разработки должен решать содержательную задачу, а не сводиться к констатации факта, что нечто должно быть сделано.

С этой точки зрения второй вариант полностью не содержателен, первый вариант лучше, в нем описана некоторая логика, но если учесть, что проблема задачи все же заключается в построении пути, то сведение исходной задачи к процедуре **Путешествие** мало что дает. Поэтому еще немного поработаем на верхнем уровне.

Особенностью нашего путешественника является то, что он в процессе движения попадает в две принципиально отличные ситуации:

- Путь по пустому пространству.
- Обход препятствия.

Заметим также, что если путешественник не дошел до пункта B , то в отношении его состояния справедливы следующие утверждения:

- По завершении обхода препятствия путешественник попадает на пустое пространство.
- Завершение пути по пустому пространству возможно только при столкновении с препятствием.

Сказанное позволяет описать верхний уровень существенно более содержательным:

Вариант 3.

$x := A_x$; $y := A_y$;

WHILE ($x \neq B_x$) OR ($y \neq B_y$) DO

```
ПутьВперед; (*Движение по прямой*)
Обход; (*Обход препятствия*)
END;
```

Программный фрагмент стал существенно более содержательным, но в то же время и ошибочным. Здесь движение по прямой всегда должно завершиться обходом препятствия. И только завершение обхода гарантирует проверку достижимости точки назначения. Это означает, что цикл путешествия завершится только в том случае, если пункт назначения окажется на границе некоторого препятствия. Отсюда мораль:

Структура передачи управления на каждом уровне разработки не может не зависеть от формулировки подзадач, полученных в результате декомпозиции.

В третьем же варианте этот важнейший принцип был нарушен. В варианте 1 путник двигался пошагово, на каждом шагу проверяя свои координаты. В варианте 3 мы существенно изменили характер движения. Теперь он выполняет движение определенного рода до тех пор, пока это возможно. Выход из положения может заключаться в переносе проверки координат в тело процедур, но тогда процедуры должны сообщать наверх о причине прекращения своей деятельности. Введем переменную – флаг, который будет истинным тогда и только тогда, когда пункт назначения не достигнут. И третий вариант запишется так:

Вариант 3. Исправленный

```
x:=Ax; y:=Ay; Flag:=TRUE;
WHILE Flag DO
  IF Flag THEN Flag:=ПутьВперед; END; (*Движение по прямой*)
  IF Flag THEN Flag:=Обход; END; (*Обход препятствия*)
END;
```

Верхний уровень полностью завершен. Отметим только, что для проверки координат ранее была только одна запись, сейчас их две, по одной в каждой процедуре. Проиграли мы или выиграли, продублировав текст? Согласно следующему правилу мы, выиграли, и довольно существенно:

Между краткостью текста и логической независимостью подзадач необходимо выбирать второе. Возможно, удлинение текста будет полностью компенсировано более простой структурой всего проекта.

Разбирать задачу детально не входит в наши планы, но еще немного продвинемся вглубь. Начнем с **ПутьВперед**. Работа этой процедуры заключается в смещении путника на некоторый малый вектор в направлении пункта *B* до тех пор, пока либо не будет достигнут пункт *B*, либо не встретится препятствие. Способ проверки первого условия очевиден. О втором условии необходимо поговорить. Пусть поле, по которому происходит движение, – это экран монитора, пусть его

цвет черный, препятствия обозначим линиями белого цвета. Тогда столкновение с препятствием – это обнаружение по ходу движения точки белого цвета. Отсюда следует необходимость в следующих процедурах:

- **ЦветТочки**(x, y) – функция, получающая на вход координаты точки и возвращающая её цвет.
- **ВекторСмещения**($OUT\ dx, dy:REAL$) – процедура, вычисляющая вектор смещения. Величины dx, dy – координаты вектора. Длина вектора, очевидно, мала, поэтому разумно для его координат использовать действительный тип. Вектор параллелен вектору $(Bx - x, By - y)$.
- **ШагСмещения**() – процедура, выполняющая шаг смещения на величину вектора.

Запишем вариант текста процедуры **ПутьВперед**:

```
ВекторСмещения(dx,dy);
(*Пока не достигнута точка В и не встретилось
препятствие выполнять движение*)
WHILE (x # Bx) OR (y # By) & (ЦветТочки(x, y) # White) DO
  ШагСмещения();
END;
IF (x=Bx) & (y=By) THEN Flag:=FALSE; END;
```

Детально обсуждать устройство и работы процедур **ВекторСмещения**() и **ШагСмещения**() не будем. Заметим только очевидный факт, что построение этого более низкого структурного уровня можно выполнить независимо от ранее сделанной работы. Чтобы дополнительно убедиться в сказанном, поработаем немного над процедурой обхода препятствия. Для её построения требуется ответ на два вопроса:

- Как организовать движение вдоль препятствия?
- Как определить, что препятствие пройдено?

Подробные ответы на оба вопроса можно найти в [4], сейчас существенно следующее:

Организация движения. Движение вдоль препятствия – это движение вдоль стенок, то есть движение с изменяющимся направлением.

Для ответа на второй вопрос нужен критерий, который можно было бы вычислять специальной процедурой. И конечно, нужна процедура, собственно выполняющая шаг путника вдоль стенки. Эти рассуждения приводят нас к следующему программному фрагменту:

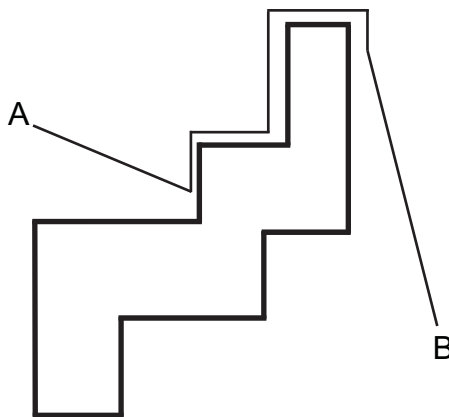


Рис. 1.4

```
WHILE Критерий() DO  
  L:=Расчет_Направления;  
  Смещение(L);  
END;
```

Прозрачный, довольно изящный, но ошибочный текст. На самом деле желания разбить задачу на подзадачи не всегда достаточно. Природа задачи может налагать естественные ограничения на возможные разбиения.

Введем понятие связанных величин. Это величины, определение значений которых выполняется в одном логически замкнутом процессе. Вычисление таких величин нельзя оформить как отдельные подзадачи.

И сейчас мы имеем дело именно с такой ситуацией. Например, величина смещения определяется после того, как определено направление смещения, эти величины нельзя назвать связанными. А вот с Критерием все несколько иначе. Собственно Критерий – это не величина, это скорее окончательный вывод, завершён обход или нет. Критерий должен быть основан на какой-то достоверной информации. В записанном тексте предполагается, что Критерий собирает необходимую информацию в той точке, в которой путешественник находится в данный момент. Это следует из того, что Критерию известны лишь глобальные значения текущих координат. Но в текущий момент времени путник видит перед собой пространство на одну точку, а для принятия решения необходимо кое-что знать о форме препятствия. Такую информацию путник может получить, только в процессе обхода, следовательно, это случай некорректно выполненной декомпозиции. Отделить процесс движения от расчета Критерия нельзя.

Если вас заинтересовала сама задача, а не иллюстрация метода декомпозиции, то выше было сказано, где о ней почитать.

В заключение. Реализуя алгоритм решения задачи, любой программист решает две фундаментальные проблемы. Во-первых, он желает получить максимально эффективный алгоритм, и, во-вторых, желает решить проблему отладки. Каждый программист тратит значительное время на борьбу с ошибками, которое желательно максимально сократить. Эти две проблемы взаимосвязаны. Грамотно спроектированная программа содержит в себе минимальное количество ошибок, поиск которых – вопрос не искусства, а технологии. Вопрос в том, как добиться грамотного построения. Тут есть два подхода. Первый – это, конечно, личный опыт и личная интуиция. Подход вполне возможный, но, к сожалению, делающий процесс разработки ПО слишком зависимым от личного фактора и переводящий программирование в область искусства или, что еще хуже, в область шаманизма. Программисты превращаются из умных инженеров в заклинателей змей, чья деятельность эффективна, но совершенно не понятна для окружающих, даже программистов (но менее опытных).

Оказывается, такой подход не обязателен. Оказывается, существует наука, может быть, не так строго развитая, как, скажем, математический анализ или теория чисел, то все же нашедшая методы, благодаря которым появился термин «прог-

рамная инженерия». Конечно, изложение этой науки – дело отдельной книги, здесь же укажем лишь тот факт, что основой ухода от танцев с бубном в программировании является правильное структурирование программы, это делает возможным и её анализ, и грамотное построение.

Вычислительные алгоритмы

Моделирование непрерывных процессов дискретными	27
Метод половинного деления.	
Общая задача поиска величины	31
Метод касательных	34
Метод хорд	35
Метод итераций (последовательных приближений)	36
Обобщение метода половинного деления	37
Метод наименьших квадратов	38
Задача вычисления площадей криволинейных фигур	42
Метод Симпсона	45
Метод Монте-Карло	48

Эта глава о том, как быстро и эффективно считать. Очень часто вопросы счета не решаются аналитически. Очень часто для неизвестной величины нельзя написать формулу, расчет по которой заключался бы в подстановке известных величин и выполнении небольшого количества арифметических операций. Например, для числа π , конечно, можно написать конечную формулу:

$$\pi = \frac{S}{2R},$$

где R – радиус некоторой окружности, а S – длина той же окружности. Но для вычисления π должны быть известны обе величины, записанные в правой части равенства, – и радиус окружности, и её длина. Если принять радиус известным, то не ясно, как вычислить длину, так как другой формулы, не использующей радиус и число π , нам неизвестно. Очень многие интегралы не выражаются аналитически (конечной формулой), например:

$$\int \sin(x^2) dx.$$

Нет гарантированного решения в радикалах для алгебраического уравнения степени выше четвертой, да и сами радикалы, за редким исключением, можно считать только приближенно. Поэтому в математике очень актуален вопрос: как организовать вычислительный процесс для расчета величины, чтобы за приемлемое время получить достаточно точное решение.

Общего ответа на этот вопрос нет. И даже более того, на некоторые частные вопросы существуют отрицательные ответы. Например, точно известно, что общая задача решения диофантова уравнения (уравнение, для которого ищутся только целые корни) алгоритмически неразрешима. Есть задачи, вполне решаемые, но найти хорошее решение пока никому не удастся. Такова, например, задача факторизации (разбиения числа на множители). Алгоритмов её решения достаточно много, но все они очень затратны по времени, и хотя никто не доказал, что нельзя придумать очень быстрый алгоритм, но пока это никому не удалось.

Поэтому вопросы счета очень актуальны, и ниже мы рассмотрим, что возможно в области вычислений. Охват проблем невелик, но для того чтобы составить хорошее представление о том, что такое вычислительный алгоритм и какие проблемы приходится решать при его разработке, материала достаточно.

Моделирование непрерывных процессов дискретными

Параграф посвящен частной, но очень важной модели вычислений в физических процессах. Многие физические задачи сводятся к уравнению, в котором в качестве неизвестной величины оказывается функция, и не просто функция сама по себе. В уравнение функция может входить вместе со своими производными. Такие уравнения называются дифференциальными.

Простой пример. Пусть физическое тело участвует в равноускоренном движении. Тогда формула пути выглядит следующим образом:

$$s = s_0 + vt + \frac{at^2}{2}.$$

Если вспомнить, что скорость – это первая производная от пути, а ускорение – вторая производная, то эта же формула будет переписана так:

$$s = s_0 + s't + s''\frac{t^2}{2}.$$

И получаем уравнение, в которое в качестве неизвестных величин входят функция s и две её производные. Вторым классический пример дифференциального уравнения – это описание процесса радиоактивного распада. Известно, что скорость распада пропорциональна количеству вещества. Эта фраза на языке формул может быть записана так:

$$y' = ky.$$

Здесь y – это функция, выражающая зависимость количества вещества от времени.

Решение дифференциального уравнения находится интегрированием, а как известно, далеко не все интегралы выражаются через функции явным образом. Мы не будем вдаваться глубоко в проблемы теории дифференциальных уравнений. Отметим только, что часто для решения уравнений и систем уравнений приходится прибегать к вычислительным методам.

А сейчас рассмотрим эту же задачу, но с несколько другой точки зрения. Это будет, во-первых, полезно для понимания источника проблем, и, во-вторых, мы получим простой и понятный метод решения задач такого типа. Анализ специально проведем на задаче не имеющей тривиального решения.

Задача движения системы тел в поле тяготения

Представим себе группу тел, движущихся в пространстве. Пусть эта группа тел единственная во Вселенной, или, по крайней мере, расстояние до других тел настолько велико, что их гравитационным воздействием можно пренебречь. Процесс начинается с некоторого момента времени, и на этот момент для каждого из тел известны пространственные координаты и векторы скорости. Пусть пока для упрощения рассуждений система состоит только из двух тел A и B .

Для расчета траектории движения тела A необходимо знать ускорение, которое тело A получает в результате гравитационного воздействия на него тела B . Ответ на этот вопрос дают закон Всемирного тяготения, позволяющий получить значение силы взаимодействия, и Второй закон Ньютона, позволяющий рассчитать ускорение из известной массы и силы. Проблема заключается в том, что в формулировку закона Всемирного тяготения входит расстояние между телами:

$$F = \gamma \frac{m_1 m_2}{R^2}.$$

Тело B движется, следовательно, в следующий момент времени расстояние между телами A и B изменится и ускорение тела A придется пересчитать. Иначе говоря, ускорение тела A есть функция от пространственных координат тела B . Но эти же рассуждения можно повторить и для тела B , то есть ускорение тела B есть функция от координат тела A . Но и это еще не все. Точно так же, как ускорение есть функция координат тел, мы можем утверждать, что координаты есть функция ускорений. И получаем систему переменных, в которой нет независимых величин. Нельзя алгебраически одну группу величин выразить через другую. Это следствие непрерывности процесса, и этот факт обуславливает появление дифференциальных уравнений и их систем.

Переход к дискретной модели

Ни одну из величин этой системы (координат и ускорений) нельзя сделать независимой. Это противоречило бы физическому смыслу процесса непрерывного взаимодействия. Но мы можем представить процесс взаимодействия дискретным. Положим, что в течение некоторого отрезка времени Δt тела взаимодействуют с постоянной силой, несмотря на изменение взаимного положения, и лишь по истечении отрезка Δt все силы пересчитываются. Получается, что мы на время Δt как бы отрываем силу от тела, и она существует самостоятельно. Так как сила численно отличается от ускорения постоянным множителем (массой), то эти же рассуждения справедливы и для ускорения. Постоянство силы на временном отрезке позволяет обойтись без дифференциальных уравнений на этом отрезке, так как сила становится независимой величиной и, более того, константой.

Будет ли полученная модель движения соответствовать действительности? Конечно же нет, так как на протяжении Δt модельное ускорение и действительное — это не одно и то же. Но также ясно, что чем меньше Δt , тем меньше и погрешность приближения, следовательно, всегда возможно подобрать такой временной интервал, что погрешности расчетов окажутся в допустимых рамках. Если потребуется более высокая точность, то всего лишь необходимо найти новый достаточно малый временной интервал. Наши потребности в точности, таким образом, ограничены только нашими же вычислительными мощностями. Наверное, для решения принципиальных теоретических проблем описанный подход не годится, но практических задач, требующих абсолютной точности, не существует. Например, нет необходимости посадить космический зонд на Марс с нулевой погрешностью. В практических задачах все требуется лишь с некоторой заданной точностью, и не более того.

В этой модели процесс разбивается на равные временные отрезки, в течение которых тела движутся с постоянным ускорением и лишь на границах отрезков значения ускорений пересчитываются.

Напишем каркас программы. Прежде всего договоримся о структурах данных. Каждое тело описывается тремя тройками величин: три пространственные координаты, три координаты вектора скорости и три координаты вектора ускорения. Это позволяет ввести общий базовый тип **БазовыйТип**.

```
TYPE БазовыйТип=RECORD
    x,y,z:REAL;
END;
```

Тогда можно описать тип **Тело** следующим образом:

```
ТипТело=RECORD
    Координаты, Ускорение, Скорость :БазовыйТип;
END;
```

И наконец, систему тел можно описать как массив объектов типа **ТипТело**.

```
VAR
    Тело:ARRAY 10 OF ТипТело;
```

Перейдем к программным конструкциям. Так как речь идет о физическом процессе, а каждый физический процесс идет во времени, то структура расчетной процедуры должна представлять собой цикл, отсчитывающий время. Время начинается с нуля (для упрощения) и завершается, когда достигает некоторого значения. В принципе, завершение цикла можно организовать по-разному, у нас время изменяется с малым шагом, до 10 000.

Процесс движения неоднороден. Есть однородные участки длиной в dt , при прохождении которых необходимо выполнять пересчет координат и возможно рисовать траектории, и по завершении каждого такого участка должен выполняться пересчет ускорений. Это означает, что кроме переменной, отвечающей за общий ход времени, должна быть еще одна временная переменная, отсчитывающая время от нуля до dt . Отсюда следует, что структура программы – это два условных цикла, внешний ведет общий отсчет времен, внутренний отсчитывает время очередного интервала dt .

Листинг 2. 1

```
t:=0;
WHILE t<=10000 DO
    t1:=0;
    WHILE t1<=dt DO
        Траектория; (*Пересчет координат и
                     возможно рисование траектории тел*)
        t1:=t1+0.001;
        t:=t+0.001;
    END;
    Пересчет; (*Пересчет ускорений для тел*)
END;
```

В тексте появляются две процедуры **Траектория** и **Пересчет**. Эти две процедуры соответствуют двум подзадачам. В полном соответствии с идеей нисходящего проектирования мы выделили три подзадачи: описание общего физического процесса, заключающегося в учете временных изменений, подзадачу расчета траектории и подзадачу пересчета ускорений. Нетрудно заметить, что подзадачи логически независимы друг от друга. Общей для них является переменная, отсчитывающая общее время. Разработку этих двух процедур мы оставим желающим.

Задание для самостоятельной работы

Разберитесь с расчетной схемой, описанной выше, и решите задачу построения траектории движения в поле взаимного тяготения N – тел. Для каждого тела известны масса и вектор начальной скорости.

Наша цель – демонстрация перехода от непрерывного процесса к дискретному – достигнута. Переходим к следующей задаче.

Метод половинного деления.

Общая задача поиска величины

Очень часто в вычислительной математике встречается задача поиска определенного значения в некотором числовом множестве. Задача решается тривиально перебором. Для примера опишем процесс поиска квадратного корня из числа C на отрезке $[a, b]$. Если известно, что корень на отрезке есть, то достаточно пройти отрезок с некоторым фиксированным шагом и найти значение x , такое что:

$$|x^2 - C| = \min,$$

где \min – минимальное из всех таких значений на $[a, b]$.

Ясно, поиск значения на большом отрезке с высокой точностью (маленьким шагом) будет слишком затратной операцией, даже для поиска квадратного корня. Рассмотрим более эффективный способ поиска значения величины, именуемый методом половинного деления, а затем попробуем его обобщить. Начнем с задачи вычисления квадратного корня. Дано число $A > 1$, необходимо найти корень. Первым действием определим отрезок, в котором корень находится гарантированно. При $A > 1$ такой отрезок определить легко: $[1, A]$. Можно взять и меньший отрезок, например $[1, A/2]$, но это не принципиально. Обозначим корень как величину B . Положим, что корень – это середина отрезка, то есть $B = (1 + A)/2$. Конечно же это предположение, скорее всего, ошибочно. Необходимо ответить на два вопроса: во-первых, насколько мы ошиблись, и, во-вторых, в какую сторону. Введем обозначения для концов отрезка, содержащего корень. Пусть левая граница обозначается переменной L , а правая – переменной R . Для первой итерации $L = 1$; $R = A$.

Какова погрешность приближения. Так как предполагаемый корень находится внутри отрезка, то очевидно, что погрешность равна половине отрезка, для каждой итерации это значение равно $(R - L)/2$.

В какую сторону ошибка. Иначе говоря, приближенное значение слишком мало

или слишком велико? Для ответа вычислим величину B^2 . Возможны три варианта: $B^2 > A$; $B^2 < A$; $B^2 = A$; третий вариант крайне маловероятен, объединим его с одним из сравнений на больше-меньше, например так:

Вариант 1. $B^2 \geq A$; наш предполагаемый корень не меньше настоящего. Случился перелет. Ситуация пояснена на рис. 2.1.

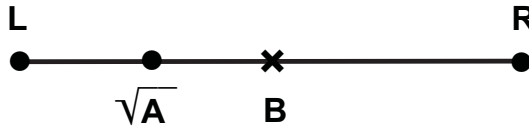


Рис. 2.1

Это означает, что в действительности корень находится на отрезке $[L, B]$, тогда для следующей итерации $R = B$. Ситуацию с равенством мы игнорируем. В конце изложения будет ясно, почему это возможно.

Вариант 2. $B^2 < A$; предполагаемый корень меньше настоящего. Это явный недостаток. Ситуация пояснена на рис. 2.2.

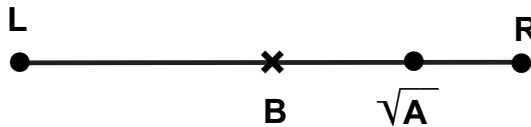


Рис. 2.2

Это означает, что в действительности корень находится на отрезке $[B, R]$, тогда для следующей итерации $L = B$.

Таким образом, вычислительный процесс представляет собой ряд итераций, в каждой из которых изменяется либо левая, либо правая граница, при этом искомый корень всегда находится в отрезке $[L, R]$ и длина отрезка после каждой итерации уменьшается в два раза. Половина длины отрезка – это погрешность расчетов, следовательно, на каком-то шаге погрешность достигнет требуемого значения, как бы мало оно ни было. После этого значение корня можно записать как $B = (L + R)/2$.

Замечание. Почему условие $B^2 = A$ можно игнорировать? Очень просто, при этом условии корень все равно находится внутри отрезка $[L, R]$, и отрезок все равно будет продолжать стягиваться к искомому значению. Конечно, это означает некоторую потерю эффективности. Но, с другой стороны, два случая обрабатывать проще, чем три, ситуация с равенством – действительно редкий случай, да и потеря эффективности невелика. Запишем процедуру расчета:

Листинг 2.2

```
PROCEDURE Расчет(A:REAL):REAL;
  VAR
    B,L,R:REAL;
  BEGIN
```



```
L:=1;R:=A;  
WHILE R-L>0.001 DO  
  B:=(L+R)/2;  
  IF B*B>A THEN  
    R:=B;  
  ELSE  
    L:=B;  
  END;  
END;  
RETURN (L+R)/2;  
END Расчет;
```

Задание для самостоятельной работы

В самом начале анализа было заявлено допущение $A > 1$. Выясните, что меняет условие $A > 0$. Надо ли в связи с этим дополнением изменить программу, и если да, то выполните необходимую модификацию.

Процедура расчета квадратного корня легко обобщается на корни любой степени, но мы сразу проведем обобщение на поиск корней любого уравнения. Пусть дано произвольное уравнение вида $F(x)=0$, где $F(x)$ – даже не обязательно алгебраическое уравнение. Заметим, что задача поиска корня n -ой степени сводится к уравнению

$$\sqrt[n]{A} - x = 0, \text{ здесь } F(x) = \sqrt[n]{A} - x.$$

Поэтому есть смысл решить общую задачу. Предположим, что в отрезке $[L, R]$ уравнение $F(x)=0$ имеет единственное решение. Источник этой информации, правда, лежит за пределами метода половинного деления. Метод позволяет найти сколь угодно точное решение, при условии что отрезок уже известен, но строить сам отрезок надо из каких-то иных соображений. Построим график функции $y=F(x)$ на отрезке $[L, R]$. Наличие корня означает, что график функции пересекает ось абсцисс в одной точке. Например, так:

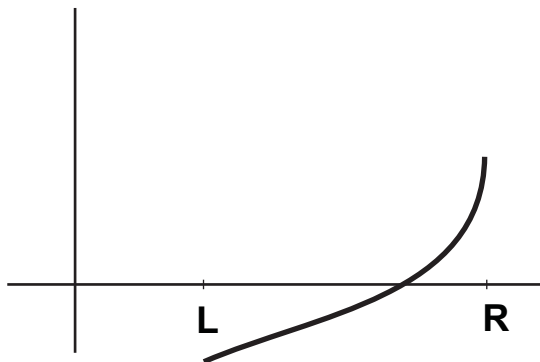


Рис. 2.3

Принципиально важна в этой картинке перемена знака. В точке L функция отрицательна, а в точке R положительна, может быть и наоборот, значимо лишь то, что на одном конце отрезка функция обязательно положительна, а на другом – отрицательна. Алгебраически это можно записать так: $F(L) \cdot F(R) < 0$. Кстати, из рисунка должно быть понятно и то, почему требуется единственность корня на заданном отрезке. Если еще не догадались, то смотрите на рис. 2.4.

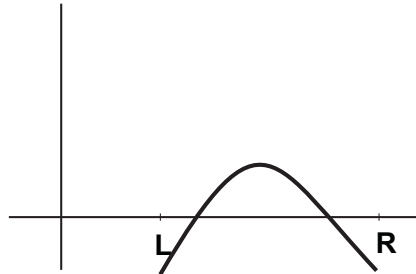


Рис. 2.4

На отрезке два корня, но ни одной перемены знака. Метод половинного деления здесь может и не сработать. Запишем фрагмент программы, вычисляющий корень уравнения на отрезке $[L, R]$.

Листинг 2.3

```
WHILE R-L>0.001 DO
  B:=(L+R)/2;
  IF F(B)*F(R)>0 THEN
    R:=B;
  ELSE
    L:=B;
  END;
END;
StdLog.Real((L+R)/2);
```

Границы задаются извне фрагмента. Функция $F()$ это некоторая функция, вычисляющая выражение вида $F(x)$. Например, для вычисления корня квадратного из числа A эта функция может выглядеть так:

Листинг 2.4

```
PROCEDURE F(x:REAL): REAL;
BEGIN
  RETURN A-x*x;
END F;
```

Метод касательных

Если говорить о поиске корней уравнений, то метод половинного деления – наиболее простой из всех методов, позволяющих пошагово уточнять искомое значение.

ние. И естественно, не единственный. Половинное деление не учитывает никаких свойств функции $F(x)$, а эта функция может нести в себе очень полезную информацию. Например, метод касательных (иногда называемый еще методом Ньютона) использует информацию о значении производной и её связи с касательной к точке графика.

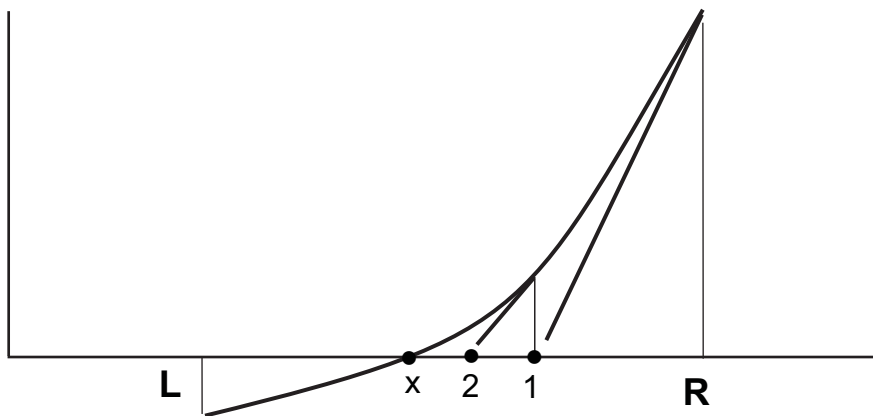


Рис. 2.5

Некоторая точка графика берется в качестве исходного приближения, естественно в качестве такой точки взять один из концов отрезка. В нашем случае исходным приближением служит правая граница отрезка. К точке графика с координатой R строится касательная. Точка пересечения касательной с осью X является точкой приближения (на рисунке точки приближения отмечены числами 1 и 2). Нетрудно увидеть, что при пологом графике точка приближения будет проходить область пологости значительно быстрее точки — половины отрезка. Этот эффект замечен и в нашем примере. Формулой итерационный шаг метода касательных выражается так:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Метод хорд

Идея та же, приближение рассчитывается, исходя из некоторого знания о функции, но если выше мы использовали для построения касательную к точке, то сейчас будет использована хорда (рис. 2.6).

Числами 1 и 2, как и в методе касательных, обозначается ряд приближений. Схема аналогична. От точек, ограничивающих кривую, строится хорда, затем определяется точка её пересечения с осью абсцисс, точка пересечения становится новой границей отрезка, после чего строится новая хорда. Итерационный процесс задается следующей формулой:

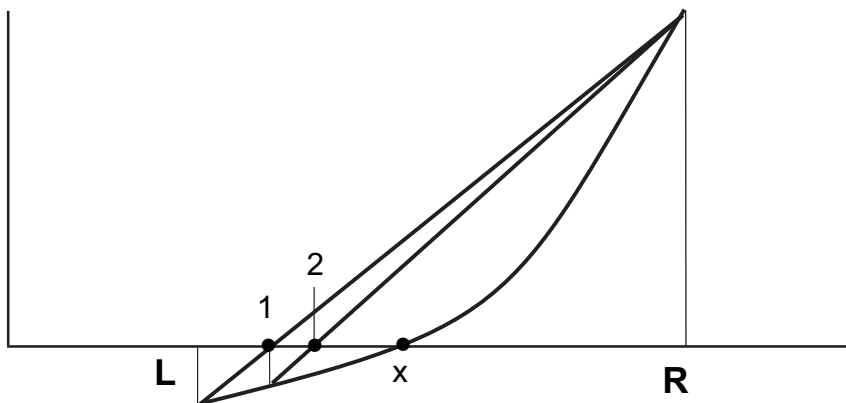


Рис. 2.6

$$x_{n+1} = x_n - \frac{(b - x_n)f(x_n)}{f(b) - f(x_n)}.$$

Оба рассмотренных метода работают быстрее, чем половинное деление. Еще большей скорости можно достигнуть, если использовать комбинированный подход, заключающийся в поочередном использовании приближения хордами и касательными. Это дает одновременный доступ и к преимуществам метода касательных, и к преимуществам метода хорд. Тратить время на детальное рассмотрение комбинированного подхода не будем, так как его детали – это детали приближений хордами и касательными.

Задание для самостоятельной работы

Постройте самостоятельно итерационный процесс для комбинированного метода, последовательно использующего метод хорд и метод касательных, и напишите программу, его реализующую.

Метод итераций (последовательных приближений)

И в основе метода хорд, и в основе метода касательных лежит одна общая идея. Она заключается в последовательном движении к корню, начиная с некоторого известного приближения. Заметим сразу, что метод половинного деления несколько отличен от этой идеи. Выполним небольшое обобщение.

Последовательность $x_1, x_2, \dots, x_n, \dots$ называется итерационной, если для любого $n \geq 2$ элемент x_n выражается через элемент x_{n-1} по рекуррентной формуле $x_n = F(x_{n-1})$. Начальное приближение x_1 – это любая точка из области определения функции F . Как видно, точки, получаемые методом касательных и методом хорд, составляют итерационную последовательность. Разумно предположить, что можно построить сколько угодно много итерационных последовательностей, и даже возможен сле-

дующий общий вопрос: даны некоторое уравнение $F(x) = 0$ и итерационная последовательность, какие условия должны выполняться, чтобы данная последовательность сходилась к корню уравнения $F(x) = 0$? Мы приведем два таких условия:

Условие 1. Пусть функция $F(x)$ непрерывна на отрезке $[a, b]$, и пусть все элементы итерационной, бесконечной последовательности $x_1, x_2, \dots, x_n, \dots$ лежат на этом отрезке. Тогда если данная последовательность сходится к некоторому числу c , то указанное число c является корнем уравнения $F(x) = 0$.

Условие 2. Пусть c – корень уравнения $F(x) = 0$, и пусть в некотором симметричном относительно точки c отрезке $[c - \epsilon, c + \epsilon]$ производная функции $F(x)$ удовлетворяет условию $|F'(x)| \leq \alpha < 1$. Тогда итерационная последовательность $x_1, x_2, \dots, x_n, \dots$, у которой в качестве x_1 взято любое число из $[c - \epsilon, c + \epsilon]$, сходится к указанному корню c .

Таковы условия, определяющие существование итерационной последовательности. Получив последовательность, пользуясь данными условиями, можно проверить, является она итерационной или нет. Сама же проблема построения итерационной последовательности остается открытой. Нет, к сожалению, никаких алгоритмов, пользуясь которыми можно было бы из условия задачи выстроить итерационную последовательность. Если судить по сложности проблемы, можно предположить, что задача построения такой последовательности алгоритмически неразрешима. По [17], [19], [23], [24] можно получить системные знания в области вычислительных методов.

Обобщение метода половинного деления

Выше мы говорили только о решении уравнений. Начали рассмотрение с наименее эффективного метода половинного деления, затем перешли к методам, дающим большую скорость вычислений. Продолжим разговор о половинном делении. Эта идея допускает существенное обобщение. В поиске корня и решении уравнений мы исходили из важного допущения, что множество, на котором ищется величина с требуемыми свойствами, обязательно линейно упорядочено и непрерывно. Это очень сильное предположение. Порядок можно построить и на дискретном множестве. А если такой порядок есть, то можно попытаться построить процедуру, определяющую, в каком из подмножеств окажется искомая величина. Приведем пример.

Задача выборки числа из миллиарда

Из числового интервала от единицы до миллиарда случайным образом выбирается миллион неповторяющихся чисел и записывается в файл. Необходимо за приемлемое время найти наименьшее, отсутствующее в файле число. Использовать массивы или иные структуры данных, могущих их заменить, запрещается.

Пояснение. Пусть, например, выбраны из интервала $1 \dots 5$ два числа 1 и 4. Тогда наименьшее невыбранное – это число 2.

Идея решения

Задача имеет простое, но, скажем сразу, негодное решение. А именно можно упорядочить файл в порядке возрастания, после чего пройти файл еще раз и найти первое число (назовем его Числом), отличающееся от своего соседа справа более чем на единицу. Тогда **Искомое значение** = **Число** + 1. Однако это плохое решение. Сортировка – очень трудоемкая операция, даже если речь идет о сортировке массива. В задаче же сказано, что числа хранятся в файле, поэтому проблема усугубляется трудоемкостью операций файлового доступа. Хорошее решение должно избавить от необходимости многократного прохода файла.

А теперь хорошая идея. Поделим миллиардный отрезок на два по 500 миллионов. Где может находиться искомое число? Очевидно, в первом отрезке, так как 500 миллионов миллионом чисел не заполнить. Далее поделим первый 500-миллионный отрезок на два и т. д. Рано или поздно заключение о том, что отрезок, содержащий искомое число, будет первым, окажется несправедливым.

Из сказанного ясно, что нужно найти отрезок, длина которого окажется больше количества чисел, в нем действительно содержащихся, и этот отрезок должен быть первым из обладающих таким свойством. Пусть, например, рассматриваемый отрезок [3000, 3999]. Чтобы быть полностью заполненным, он должен содержать 1000 чисел (напомним, что числа в файле не повторяются, иначе чисел в отрезке может оказаться ровно 1000 и отрезок при этом будет не заполнен). Известно, что все эти числа не меньше 3000 и не больше 3999. Прочитаем файл один раз и посчитаем, сколько в нем чисел от 3000 и до 3999. Если их окажется меньше 1000, то в этом отрезке есть дырка.

Очевидно, что числовой отрезок упорядочен, порядок линейный, но отрезок не непрерывен. Процедура определения подмножества, содержащего искомую величину, сильно отличается от того, как мы это делали с вычислением квадратного корня, то тем не менее по сути это тоже половинное деление. Отсюда следует вывод об общем характере метода половинного деления. Половинное деление – это даже, пожалуй, не метод, а общий принцип поиска неизвестных величин в упорядоченных множествах.

Задание для самостоятельной работы

Впрочем, в этом решении не обязательно делить отрезок на два. Подумайте о более эффективном разбиении.

Метод наименьших квадратов

Для ряда прикладных отраслей знания, например статистики, экспериментальной физики, метеорологии и т. д., важна проблема прогнозирования поведения некоторой величины. В общем виде эту задачу можно сформулировать так: известно, что величина y (будем называть её зависимой) зависит от величины x (будем называть её независимой). Из какого-то источника (например, эксперимента) известно, что в ряде определенных точек x_k зависимая величина y принимает определенные значения. Необходим прогноз значений зависимой величины y в иных точках.

Предположим, что выборка статистических данных или физический эксперимент дал в качестве результата две колонки чисел:

Таблица 2.1

х	у
....
....
....
....

Здесь x – независимая переменная, y – результат эксперимента. Значения y в заданных точках уже известны, интересно выяснить, как будет вести себя зависимая переменная в других точках. В общем-то вести она может себя как угодно, но окружающий нас мир устроен таким образом, что большинство зависимостей ведут себя гладко. Это означает, что для фиксированной точки x существует окрестность, в которой небольшое изменение независимой переменной ведет к предсказуемому, небольшому изменению зависимой, то есть существует функция $y=f(x)$. Мы не знаем вид этой функции, но из математического анализа известно, что гладкую функцию (имеющую непрерывные производные) можно разложить в ряд, иначе говоря, представить функцию в виде суммы функций одного вида, например в виде суммы степенных (ряд Тэйлора), суммы тригонометрических функций (ряд Фурье). На малом интервале любую гладкую функцию можно приблизительно представить линейной функцией, параболой. Вопрос только в точности представления. Если точек, через которые проходит гипотетическая функция, несколько, то абсолютно точного представления не получится, но это лишь вопрос допустимой погрешности.

Если поставлена задача построения приближения гипотетической функции функциями известного вида, то необходимо, во-первых, выбрать вид функции, с помощью которой должно строиться приближение, и, во-вторых, необходим метод, позволяющий выжать минимум погрешности. Простейшая функция для построения приближений – это, конечно, линейная, то есть функция вида $y = kx + b$.

Рисунок 2.7 демонстрирует неоднозначность возможных представлений. Кривой линией показана приближаемая функция (пусть она нам известна), прямыми линиями – возможные приближения.

Даже невооруженным глазом видно, что погрешность приближений очень различна (вплоть до полного отсутствия приближения). Каждая из линий определяется двумя величинами: угловым коэффициентом k и величиной смещения b . Для решения поставленной задачи необходимо найти способ вычисления этих величин в зависимости от выбранного значения погрешности. Так как приближение строится на отрезке, то, очевидно, предметом анализа должна стать суммарная погрешность вдоль всего отрезка. В качестве простого варианта можно взять сумму разностей между приближенной и приближаемой величиной. Метод наимень-

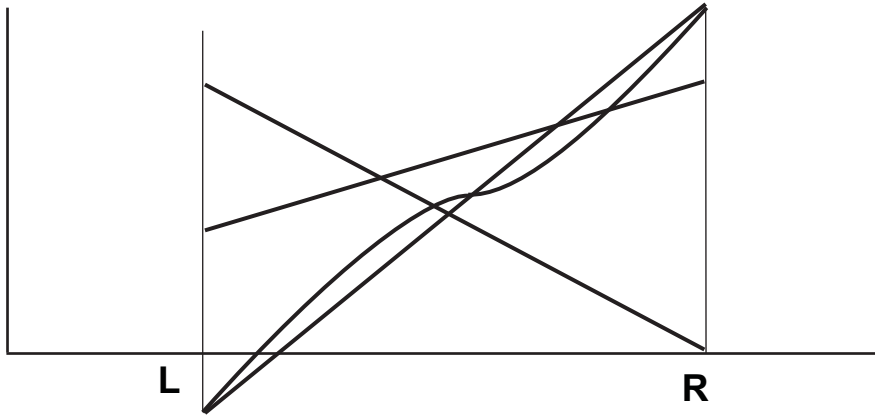


Рис. 2.7

ших квадратов предлагает в качестве оценки взять сумму квадратов разностей. Выбор обусловлен тем, что квадратичная функция растет быстрее линейной, поэтому уменьшение квадратичной погрешности даст лучшее приближение. Таким образом, основой для построения метода становится следующая формула:

$$z = \sum_i^n (y_i - kx_i - b)^2.$$

Эта сумма оценивает квадратичную погрешность. Чем она меньше, тем лучше приближение. Величину z можно рассматривать как функцию от двух величин $z = z(k, b)$. И задача поиска минимума квадратичной погрешности сводится к поиску минимума функции от двух переменных. В точке минимума, как известно из дифференциального исчисления, первые производные равны нулю. Так как функция z – функция от двух переменных, то, находя частные производные и приравняв их к нулю, получаем два уравнения:

$$\frac{\partial z}{\partial b} = 2 \sum_{i=1}^n (y_i - kx_i - b) = 0.$$

Из этих равенств получаем систему линейных уравнений для определения величин k и b :

$$\begin{cases} a \sum_{i=1}^n x_i^2 + b \sum_{i=1}^n x_i = \sum_{i=1}^n x_i y_i \\ a \sum_{i=1}^n x_i + bn = \sum_{i=1}^n y_i \end{cases}$$

Пример расчета. Пусть эксперимент дал следующие результаты:

Таблица 2.2

x	y
1	2
2	5
3	7
4	9

Мы специально взяли такой простой набор данных. Ясно, что этот набор абсолютно точно ложится на линейную функцию $y = 2x + 1$. Посмотрим, что даст попытка приближения. Рассчитаем значения рядов, дающих значения коэффициентов:

$$\sum_{i=1}^n x_i^2 = 1^2 + 2^2 + 3^2 + 4^2 = 30$$

$$\sum_{i=1}^n x_i = 1 + 2 + 3 + 4 = 10$$

$$\sum_{i=1}^n y_i = 2 + 5 + 7 + 9 = 23$$

$$\sum_{i=1}^n x_i y_i = 1 * 2 + 2 * 5 + 3 * 7 + 4 * 9 = 69.$$

В результате получаем следующую систему для расчета коэффициентов:

$$\begin{cases} 30k + 10b = 69 \\ 10k + 4b = 23 \end{cases}$$

Решив систему, получим значения коэффициентов: $b = 0$; $k = 2.3$. Приближенная функция $y = 2.3x$, определим значения в наших точках по приближенной функции:

Таблица 2.3

x	y
1	2.3
2	4.6
3	6.9
4	9.2

Как видно, погрешность достаточно высока, но и шаг эксперимента достаточно велик. Можно предположить, что с уменьшением шага независимой величины погрешность также должна уменьшаться. Это действительно так, но мы это утверждение оставим без доказательства, как гипотезу.

Программная реализация метода не представляет никаких трудностей, для небольшого набора данных все расчеты можно провести и вручную, но в реальных экспериментах речь, конечно, пойдет об огромных массивах данных, поэтому задача все же программистская, не в силу алгоритмической сложности, а по причине потребности в большом количестве вычислительных ресурсов.

Задание для самостоятельной работы

Напишите программную реализацию метода наименьших квадратов.

Задача вычисления площадей криволинейных фигур

Для некоторых фигур, например кругов, известны простые формулы расчета площади. Есть формулы и для объемов некоторых тел, например, цилиндра, шара. Но количество плоских фигур и тел неизмеримо велико, и для большинства из них никаких формул геометрия не дает. Некоторую помощь в этом вопросе оказывает математический анализ, связывающий проблему счета площадей и объемов с операцией интегрирования. Оказывается, понятие интеграла имеет несколько интерпретаций, геометрическая интерпретация связана с расчетом площадей и объемов. Рассмотрим пример на рис. 2.8:

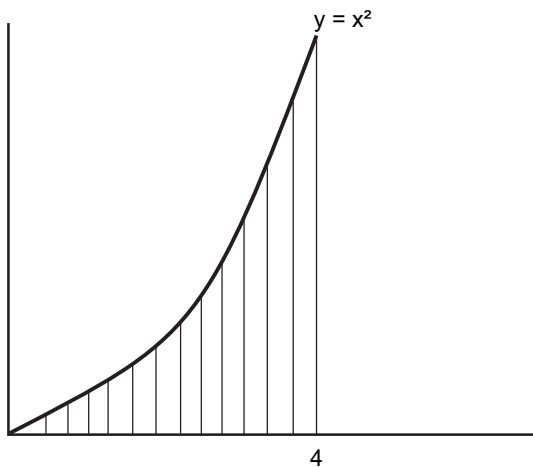


Рис. 2.8

Площадь заштрихованной области можно вычислить как следующий интеграл:

$$\int_0^4 x^2 dx = \frac{1}{3} x^3 \Big|_0^4 = \frac{64}{3} - 0 = \frac{64}{3}.$$

К сожалению, сведение расчета площади к интегралу не всегда решает проблему. Если форма кривой известна (а построить функцию на наборе точек можно всегда, это мы видели из метода наименьших квадратов), то записать интегральное выражение можно, но, скорее всего, этот интеграл окажется «неберущимся», то есть не имеющим простого аналитического выражения.

В этом случае необходимо обратиться к вычислительным методам. Попробуем ответить на вопрос, что означает применение вычислительных методов к задаче счета площади. Что вообще означает посчитать площадь?

Если сказано, что площадь некоторой фигуры составляет 10 кв. м это означает, что в фигуру можно уложить 10 квадратов, каждый площадью по одному квадратному метру. В общем можно сказать, что измерить площадь фигуры – это то же самое, что уложить в фигуру фигуры, чьи площади измеряются легко, по известным геометрическим формулам.

В примере на рис. 2.9 криволинейная фигура замощена тремя прямоугольниками.

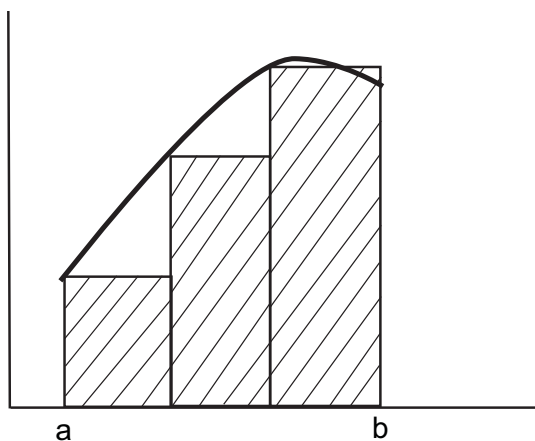


Рис. 2.9

Сумму площадей этих прямоугольников можно с некоторой погрешностью считать площадью криволинейной фигуры. Погрешность будет тем меньше, чем прямоугольников больше, или, иначе говоря, чем мельче разбиение отрезка $[a, b]$. Оценим погрешность расчетов. Для этого введем обозначение: пусть S_n – площадь разбиения, состоящего из n прямоугольников. Тогда **Погрешность** = $|S_n - S_{n-1}|$. Очень важен вопрос, насколько быстро будет убывать погрешность с ростом числа n . К сожалению, гарантировать высокую скорость убывания нельзя. Весьма вероятно, что сложная криволинейная фигура может потребовать значительных вычислительных ресурсов.

Важное замечание о погрешности. В действительности указанный метод оценки безусловно работает только в случае функций, монотонно убывающих или монотонно возрастающих на отрезке $[a, b]$. Кроме того, для погрешности метода совершенно не безразлично, как строятся прямоугольники. Наибольшая точность будет достигнута, если опорную точку для построения высоты брать в середине отрезка разбиения. В этом случае формула расчета интеграла выглядит следующим образом:

$$\int_a^b f(x)dx = h \sum_{i=1}^n f\left(x_{i-1} + \frac{h}{2}\right), \text{ где } h = \frac{b-a}{n}.$$

Идея ускорения вычислительного процесса лежит на поверхности. Если замощение прямоугольниками не дает желательной эффективности, то нужно выбрать фигуру, расчет площади которой не сложнее, чем расчет площади прямоугольника, но эта фигура ложилась бы на кривую более точно. Такой простой фигурой будет трапеция. Действительно, формула для вычисления площади трапеции не принципиально отличается от счета площади прямоугольника:

$$S = \frac{a+b}{2} h.$$

Произведение полусуммы оснований на высоту. Трапеция действительно ложится более гладко, что видно из рис. 2.10.

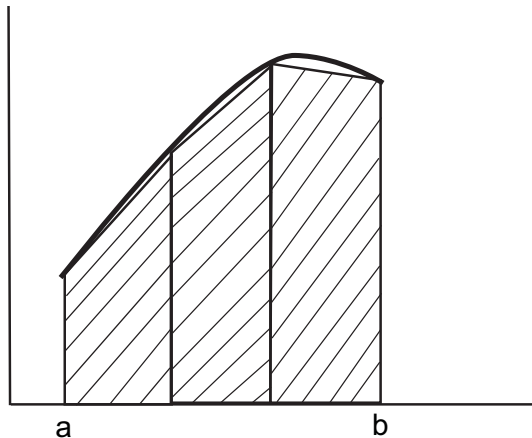


Рис. 2.10

Уже три трапеции легли на криволинейную фигуру с очень высокой степенью точности. Выигрыш в скорости не обязывает нас в этом случае на серьезные уступки. Схема оценки точности та же. Схема построения алгоритма не меняется. Расчетная формула также не намного сложнее. На отрезке из разбиения с номером k :

$$S_k \approx \frac{f(x_{k-1}) + f(x_k)}{2} (x_k - x_{k-1}).$$

Математический анализ дает и формулу для оценки погрешности:

$$|R| \leq \frac{(x_k - x_{k-1})^3}{12} M, \text{ где } M = \max |f''(x)| \text{ для } x \in [x_{k-1}, x_k].$$

Чтобы получить значение для площади на общем отрезке $[a, b]$, необходимо найти сумму всех S_k . После суммирования получается следующая формула:

$$\int_a^b f(x) dx \approx h \left(\frac{f(x_0) + f(x_n)}{2} + \sum_{k=1}^{n-1} f(x_k) \right), \text{ где } h = \frac{b-a}{n}.$$

Дальнейшее усложнение метода – опять таки связано с более точным приближением криволинейной границы. Но метод трапеций исчерпывает возможности приближения прямыми отрезками. Следующий шаг – это приближение отрезками параболы.

Метод Симпсона

Метод предполагает разбиение интервала интегрирования на небольшие отрезки, на каждом из которых строится парабола по трем точкам: двум конечным и середине отрезка. Пример на рис. 2.11.

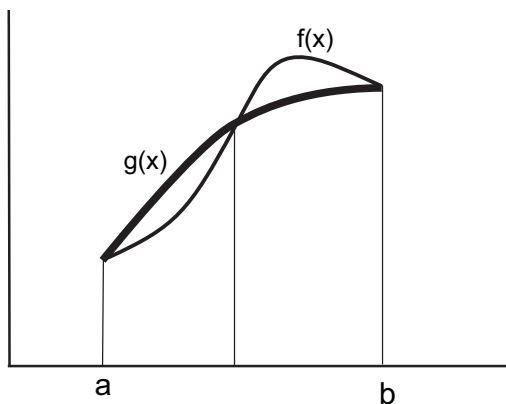


Рис. 2.11

Функция $f(x)$ – функция, ограничивающая криволинейную фигуру. Функция $g(x)$ – парабола, является приближением. В этом методе мы вынужденно отходим от простых геометрических формул, но все же использование параболы не слишком усложняет расчетную схему в силу того, что парабола легко интегрируется. Если исходную функцию $f(x)$ заменить параболой – построенной по трем точкам на $[a, b]$, то интеграл можно найти по формуле

$$\int_a^b f(x)dx \approx \int_a^b g(x)dx = \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b)).$$

Погрешность для одной криволинейной трапеции можно оценить по формуле

$$|R| \leq \frac{1}{90}(b-a)^5 M, \text{ где } M = \max(f^{(4)}(x)) \text{ для } x \in [x_{k-1}, x_k].$$

Если интервал интегрирования разбить на $2N$ равных частей, то значение для интеграла будет таково:

$$\int_a^b f(x)dx \approx \frac{b-a}{6n}(f_0 + 4(f_1 + f_3 + \dots + f_{2n-1}) + 2(f_2 + f_4 + \dots + f_{2n-2}) + f_{2n}).$$

$$\text{Здесь } f_k = f\left(a + \frac{(b-a)k}{2n}\right).$$

Еще раз об основной идее. Для подсчета площади криволинейную фигуру необходимо разбить на простые. Если это удастся сделать, то счет малых площадей сведется к счету интегралов и их суммированию. При этом погрешность определяется тем, насколько точно функция, используемая для приближения, ложится на границу области (замена одной кривой на другую называется аппроксимацией). Самая грубая аппроксимация дается методом прямоугольников, следующий уровень точности дает аппроксимация трапециями. Затем повышение точности дает аппроксимация многочленом. Простейший многочлен – это парабола, её степень 2. Увеличение степени многочлена дает уменьшение погрешности, но в то же время и усложняет расчетную схему. Уменьшение погрешности также возможно и за счет уменьшения шага разбиения интервала интегрирования.

Итак, геометрическая интерпретация интеграла – это площадь. Однако здесь есть важный нюанс, требующий уточнения. Рассмотрим следующий случай (рис. 2.12):

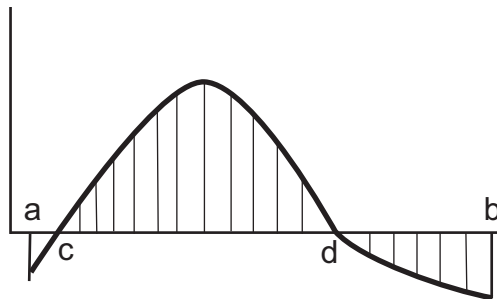


Рис. 2.12

Часть заштрихованной области лежит ниже оси OX . Это означает, что часть интеграла

$$\int_a^b f(x)dx,$$

будет иметь отрицательное значение, и как следствие – значение этого интеграла при любой точности счета не будет соответствовать площади. В этом случае:

$$S = \left| \int_a^c f(x)dx \right| + \left| \int_c^d f(x)dx \right| + \left| \int_d^b f(x)dx \right|.$$

Еще один пример (рис. 2.13):

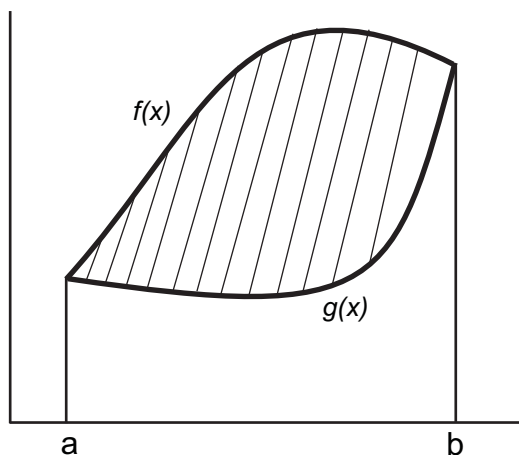


Рис. 2.13

В этом случае площадь заштрихованной области будет считаться так:

$$S = \int_a^b f(x)dx - \int_a^b g(x)dx.$$

Таким образом, к умению считать интегралы необходимо еще дополнительное умение представить измеряемую фигуру как сумму/разность фигур и соответственно интегральное выражение, как сумму/разность интегральных выражений.

Разумеется, параграф не стоит рассматривать как сколько-нибудь исчерпывающее руководство по расчету площадей криволинейных фигур. Мы дали только общую идею, как решается проблема измерения площадей. Но этой общей идеи и простых методов, изложенных выше, тем не менее будет вполне достаточно для очень значительного количества прикладных задач. Если же возникнет необходимость более глубокого изучения вопроса, то существует обширная литература

по математическому анализу и вычислительным методам, в частности из которой можно подчеркнуть любую необходимую информацию например, [17], [19].

Исследовать детально задачу счета объемов не будем. Принципиально объем считается так же, как и площадь. Но если для счета площади плоская фигура должна быть замощена простыми плоскими фигурами, то для счета объема тело должно быть заполнено простыми объемными телами. За деталями мы отправим вас к специальной литературе.

Задание для самостоятельной работы

Напишите программу, вычисляющую методами прямоугольников, трапеций и методом Симпсона площадь, ограниченную отрезком $[a, b]$, $a > 0$ и $b > 0$ и кривой

вида: $y = \sum_{k=0}^n a_k x^k$, пусть для любого k $a_k > 0$.

Оцените точность для одних и тех же разбиений.

Метод Монте-Карло

Этот параграф вам, наверное, придется прочитать дважды. Сейчас просто пробежите текст и попытайтесь ухватить основную идею. Затем на некоторое время перейдите к приложению, описывающему основные идеи теории вероятности, и после приложения изучите параграф уже внимательно. Идея излагаемого здесь метода проста, но очень неожиданна. Оказывается, вполне определенные величины, например такие, как площадь, вполне можно считать с помощью величин, не определенных совершенно. Проиллюстрируем это утверждение примером (рис. 2.14).

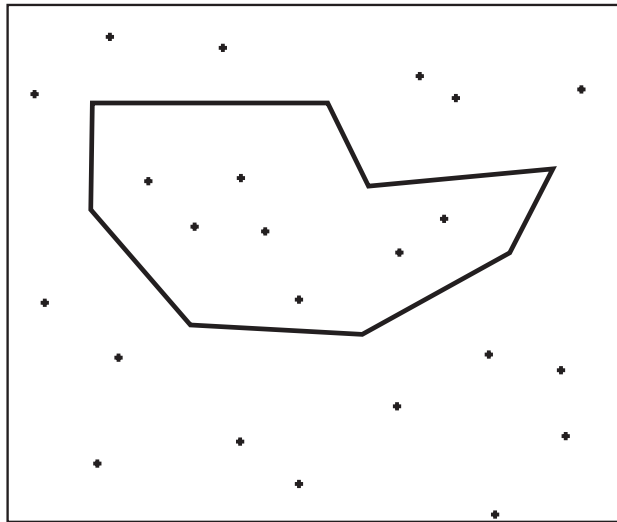


Рис. 2.14

На рис. 2.14 изображен квадрат. Обозначим его площадь через S . Внутри квадрата – некоторая фигура, площадь которой необходимо вычислить. Обозначим искомую площадь через R . Забрызгаем квадрат маленькими пятнышками краски, настолько маленькими, что их размерами в сравнении с площадью квадрата можно пренебречь. Затем подсчитаем, сколько пятнышек окажется внутри квадрата. Пусть Sp – общее количество пятнышек и Rp – количество пятнышек, оказавшихся внутри искомой фигуры. Тогда интуитивно понятно, что имеет место пропорция

$$\frac{S}{R} \approx \frac{Sp}{Rp}.$$

Откуда легко выражается площадь фигуры. В формуле стоит знак приближенного равенства, по понятной причине. Но тогда возникает вопрос о погрешности. Даже два вопроса:

1. Как оценить погрешность?
2. Как сделать так, чтобы погрешность была минимальной?

Второй вопрос особенно важен. Далеко не безразлично, каким образом мы забрызгали исследуемую фигуру. Представьте себе, что исследуемая фигура – это мишень для хорошего профессионального снайпера в ясную погоду на очень близком расстоянии. При таких условиях вполне можно ожидать, что все точки, а тогда это дырки от пуль, окажутся внутри измеряемой фигуры. В этом случае придется прийти к выводу, что площадь исследуемой фигуры равна площади всего прямоугольника, то есть это тот случай, когда ошибка окажется слишком велика.

Перейдем к более строгому языку. Координаты дырок от пуль – это случайные величины. При условии участия снайпера эти величины становятся не очень случайными. Следовательно, для успеха измерительного предприятия необходим стрелок с закрытыми глазами, абсолютно случайная величина, причем с равномерным распределением своих значений. В этой книге есть параграф, посвященный генераторам случайных величин. Можете немного отвлечься и изучить этот вопрос. Вы убедитесь, что генерация случайных значений – достаточно непростая вещь, если в вашем распоряжении только средства математики и программирования. Не зря такие генераторы (компьютерные программы и соответствующие им математические средства) называются генераторами псевдослучайных чисел. Генератор псевдослучайных чисел – это все-таки закономерный стрелок, просто его способность к снайперскому выстрелу обнаружится тогда, когда уже будет поздно. Впрочем, качество случайной величины не является предметом рассмотрения данного параграфа, далее будем предполагать, что в нашем распоряжении имеется идеальный генератор, и посмотрим, что из него можно извлечь.

Пусть требуется вычислить величину m . Для её расчета методом Монте-Карло необходимо построить случайную величину (обозначим её как x), такую что её математическое ожидание совпадет с искомой величиной m , $Mx = m$. Разумеется, Метод Монте-Карло не дает никаких рекомендаций о том, как таковую величину построить, он говорит только о том, как имеющейся случайной величиной воспользоваться.

Рассмотрим N независимых случайных величин x_k , таких что их распределения совпадают с распределением величины x . Если N достаточно велико, то согласно центральной предельной теореме теории вероятности распределение суммы

$$\sum_k^N x_k,$$

можно с некоторой погрешностью считать нормальным, с параметрами:

$$a = Nm \text{ и } \sigma = b\sqrt{N}.$$

Тогда с вероятностью, отличающейся от единицы на 0,003, значение суммарной случайной величины попадет в известный интервал:

$$P\{Nm - 3b\sqrt{N} < p_N < Nm + 3b\sqrt{N}\} \approx 0,997.$$

Если выражение, стоящее в фигурных скобках, поделить на N , это никак не отразится на вероятностной оценке, а выражение примет следующий вид:

$$P\left\{m - \frac{3b}{\sqrt{N}} < \frac{p_N}{N} < m + \frac{3b}{\sqrt{N}}\right\} \approx 0,997.$$

Выполним еще одно преобразование и получим выражение, дающее как метод вычисления неизвестной величины m , так и оценку получаемой погрешности:

$$P\left\{\left|\frac{1}{N} \sum_k^N x_k - m\right| < \frac{3b}{\sqrt{N}}\right\} \approx 0,997.$$

А именно из полученной формулы следует, что среднее арифметическое случайных значений будет примерно равно m , при этом с очень высокой долей вероятности погрешность не превышает значения $3b/\sqrt{N}$. Очевидно также, что с ростом N погрешность стремится к нулю. Рассмотрим еще одну прикладную задачу.

Задача оценки качества

Проведем анализ качества некоторого изделия, состоящего из нескольких комплектующих. Пусть качество изделия оценивается по одному выходному параметру. И этот параметр зависит от работы каждого из комплектующих. Пусть также для упрощения ситуации каждая часть изделия характеризуется ровно одним параметром. Введем обозначения. Пусть m – значение измеряемого параметра изделия и x_k – значения параметров комплектующих частей. Тогда

$$m = F(x_1, x_2, \dots, x_n).$$

Качество изделия определяется погрешностью между желаемым значением m и значением, получаемым в действительности. Следовательно, оценка качества сводится к вычислению функции $F()$. Однако функция $F()$ может оказаться достаточно сложной, более того, её аналитическая форма может быть неизвестной. Единственное, в чем можно быть уверенным, что есть метод, позволяющий получить значение m при известных значениях аргументов $F()$.

Проблемы расчета функции $F()$ к тому же не самые главные. Не вполне понятно, как оценить границы изменения величины m . Вроде бы естественная идея взять худшие значения параметров x_k не сработает, так как не ясно, что считать худшими значениями. Это именно та ситуация, в которой метод Монте-Карло даст результат.

Построим для каждого параметра случайную функцию. Затем проведем N испытаний, в которых:

- получим случайное значение для каждого параметра.
- вычислим функцию F_k .

$$\text{Тогда } Mm \approx \frac{1}{N} \sum_{k=1}^N F_k .$$

Полученное значение математического ожидания и есть наиболее вероятное значение параметра m , характеризующего качество изделия.

Расчет площади

Мы начали разговор о методе Монте-Карло с проблемы счета площади. Рассмотрим этот пример детально и доведем его до работающей процедуры. Пусть требуется рассчитать площадь круга радиуса R . Пример хорош простотой проверки. Все, что требуется для оценки погрешности, – это формула площади круга:

$$S = \pi R^2 .$$

Измеряемый круг определяется тремя величинами: двумя координатами центра окружности и радиусом: x_0, y_0, R . Случайная величина – это точка на плоскости, точнее, это две случайные величины – координаты x, y точки. Впишем исследуемую окружность в квадрат с координатами: $(x_0 - R, y_0 - R)$; $(x_0 + R, y_0 - R)$; $(x_0 + R, y_0 + R)$; $(x_0 - R, y_0 + R)$. Площадь описывающего квадрата $S_{\text{кв}} = 4R^2$.

Для измерения окружности необходимо выполнить N испытаний, результатом каждого из которых будет точка в квадрате и, возможно, в круге. Общее количество точек равно количеству испытаний. Если количество точек, попавших в круг обозначим, как m , то площадь круга можно определить из пропорции

$$\frac{S_{\text{круга}}}{4R^2} = \frac{m}{N} .$$

Для демонстрации эффективности метода напомним программу, вычисляющую относительную погрешность расчетов при разном количестве испытаний. Обозначим через N количество испытаний и проведем серию экспериментов:

```
FOR N:=10 TO 10000 BY 1000 DO
  (*Очередной эксперимент*)
END;
```

Очередной эксперимент заключается в выполнении N испытаний, для каждого испытания необходимо вычислить случайные значения координат точки. Случайные величины координат должны вычисляться таким образом, чтобы точки гарантированно попадали внутрь квадрата, описывающего круг. Это условие обеспечивается следующими формулами:

```
x:=Ran.Int(2*R)+x0-R;
y:=Ran.Int(2*R)+y0-R; (Ran – функция, дающая случайную величину)
```

Чтобы выяснить, попала точка в круг или нет, достаточно вычислить её расстояние до центра круга и сравнить это расстояние с радиусом круга, точка в круге – если это расстояние меньше либо равно радиусу. Сказанное запишется на КП так:

```
L:=Math.Sqrt(Math.Power(ABS(x-x0),2)+Math.Power(ABS(y-y0),2));
IF L<=R THEN sum:=sum+1; END;
```

Оператор `sum:=sum+1` считает точки, попавшие в круг, общее количество точек, попавших в квадрат, равно N (все точки). По завершении цикла эксперимента остается рассчитать вероятностную площадь круга и вычислить относительную погрешность. В следующем листинге полный текст:

Листинг 2.5.

```
MODULE Модуль;
IMPORT In, StdLog, Ran:=Info21sysRandom, Math;
PROCEDURE МонтеКарло*;
VAR
  x0,y0,R,N,k,x,y,sum:INTEGER;
  S1,S2,L:REAL;
BEGIN
  In.Open;
  In.Int(x0);In.Int(y0);In.Int(R);
  S1:=Math.Pi()*R*R;
  FOR N:=10 TO 10000 BY 1000 DO
    sum:=0;
    FOR k:=1 TO N DO
      x:=Ran.Int(2*R)+x0-R;
      y:=Ran.Int(2*R)+y0-R;
      L:=Math.Sqrt(Math.Power(ABS(x-x0),2)+Math.Power(ABS(y-y0),2));
```

```
IF L<=R THEN sum:=sum+1; END;  
END;  
S2:=sum/N*4*R*R;  
StdLog.Real(ABS(S2-S1)/S1*100);  
StdLog.Ln;  
END;  
END МонтеКарло;  
END Модуль.
```

При большом количестве экспериментов можно заметить, что относительная погрешность уменьшается. Однако это уменьшение не монотонное. Возможны скачки погрешности. Это следствие вероятностного характера процесса. Для вычисления случайной величины в программе использован модуль **Info21sysRandom**, вы можете использовать любое другое средство. Единственное – ваша функция случайного числа должна давать целые числа в отрезке от 0 до Аргумент – 1.

Закключение. Наверное, любой численный метод можно представить как метод пошагового уточнения вычисляемой величины. И, наверное, главная проблема любого численного метода – это скорость уточнения, или, другими словами, скорость убывания погрешности. Если погрешность расчетов убывает медленно, то это означает большую потребность в машинных ресурсах. Необходимо заметить, что современная аппаратура так быстро наращивает вычислительную мощь, что многие программисты не считают необходимым задумываться над выбором вычислительного метода, считая, что все решают ресурсы. Это принципиальная ошибка. И сейчас легко встретить задачу, для которой имеющихся ресурсов может оказаться недостаточно, так что и сейчас наш главный ресурс – это отнюдь не компьютер, а математика.

Числовые алгоритмы

Алгоритм Евклида	54
Алгоритмы факторизации	57
Тесты простоты	73
Псевдослучайные числа	78

Название «Числовые алгоритмы» очень не однозначно. В каком-то смысле, наверное, любой алгоритм можно назвать числовым. Но мы все же воспользуемся данным термином, чтобы охарактеризовать вполне определенный класс алгоритмов. От всех прочих они отличаются тем, что ставят цель – найти число определенного класса, например являющееся простым, или делителем некоторого другого числа. Мы займемся поиском псевдослучайных чисел и т.д. Этот класс алгоритмов еще можно определить как алгоритмы, обслуживающие потребности теории чисел и в какой-то степени являющиеся предметом этой теории.

Алгоритм Евклида

НОД – наибольший общий делитель, – число, делящее без остатка пару чисел и являющееся наибольшим из возможных делителей. Например: НОД(8, 6)=2; НОД(12, 8)=4. Очень важное понятие, необходимость в вычислении которого возникает достаточно часто. Простой расчетной формулы для НОД не существует. Решение полным перебором возможно, но для больших чисел требуемый объем работы делает переборное решение бессмысленным. Достаточно хорошее решение проблемы нашел еще Евклид. Есть, правда, мнение, что алгоритм Евклида придуман вовсе не Евклидом, но проблемы истории математики не являются предметом нашего анализа. Алгоритм основан на следующем арифметическом факте: поделим целое A на целое B . Результат деления можно записать так:

$$A = p_1 B + q_1,$$

где p_1 – частное, q_1 – остаток от деления.

Если числа A и B имеют наибольший общий делитель C , то, очевидно, это число C является наибольшим общим делителем и чисел B и q_1 . Таким образом, задача поиска НОД чисел A и B сводится к поиску НОД чисел B и q_1 . Так как $B < A$ и $q_1 < B$, то задача вычисления НОД сводится к задаче вычисления НОД от меньших чисел. Можно, таким образом, выстроить цепочку формул:

$$A = p_1 B + q_1$$

$$B = p_2 q_1 + q_2$$

$$q_1 = p_3 q_2 + q_3$$

.....

Получаем цепочку убывающих целых чисел, которая рано или поздно должна прерваться, цепочка не может содержать более чем B членов и заканчивается некоторым q_{n+1} , равным нулю. Тогда последний ненулевой остаток q_n , согласно алгоритму Евклида и есть наибольший общий делитель чисел A и B . В листинге 3.1 процедура получает на вход два целых числа и вычисляет их НОД.

Листинг 3.1

```
PROCEDURE Евклид(A,q0:INTEGER):INTEGER;  
VAR  
  q1,c:INTEGER;
```

```

BEGIN
q1:=A MOD q0;
WHILE q1#0 DO
  c:=q1;
  q1:=q0 MOD q1;
  q0:=c;
END;
RETURN q0;
END Евклид;

```

Примечание. Реализация имеет одно небольшое ограничение. Предполагается, что $A > q0$.

Задание для самостоятельной работы

Наша реализация алгоритма Евклида использует процедуру деления с остатком, то есть фактически используется деление, от которого вполне можно отказаться. Действительно, частное целочисленного деления – это величина, означающая, сколько раз можно из делимого вычесть делитель (до первого отрицательного или нулевого значения), остаток – это величина, содержащая остаток от делимого после того, как процедура вычитания завершена. Напишите программу, не использующую операцию нахождения остатка. Кстати, иногда можно услышать еще одно название алгоритма – «алгоритм взаимовычитания».

Дополнение. Алгоритм Евклида тесно связан с цепными дробями. Цепная дробь – это представление рационального числа последовательностью частных, что это такое, поясним примерами: пусть дано число $\frac{35}{21}$. Построим его представление в виде цепной дроби:

$$\frac{35}{21} = 1 + \frac{14}{21} = 1 + \frac{1}{\frac{21}{14}} = 1 + \frac{1}{1 + \frac{7}{14}} = 1 + \frac{1}{1 + \frac{14}{7}} = 1 + \frac{1}{3}.$$

Таким образом, число $\frac{35}{21}$ представлено числами (1, 3). Еще один пример:

$$\frac{17}{7} = 2 + \frac{3}{7} = 2 + \frac{1}{\frac{7}{3}} = 2 + \frac{1}{2 + \frac{1}{3}} \text{ то есть } \frac{17}{7} = (2, 2, 3)$$

Задание для самостоятельной работы

Из двух представленных примеров нетрудно увидеть связь между алгоритмом Евклида и процессом построения цепной дроби. Попробуйте выявить эту связь и напишите программу перевода рационального числа в цепную дробь.

Алгоритмы факторизации и поиска простых

Факторизация – это задача разбиения числа на множители. Известно, что любое число можно представить в следующем виде:

$$A = a_1^{k_1} a_2^{k_2} \dots a_n^{k_n}.$$

Здесь a_i – простые множители числа A , k_i – степень, с которой i -й делитель входит в разложение числа A . Такое разложение существует всегда и называется оно каноническим разложением числа. Для построения канонического разложения достаточно научиться находить только один делитель. Получив делитель и поделив на него число A (до тех пор, пока это возможно), мы переходим к задаче поиска делителя для меньшего числа. Поиск возможного делителя, если речь не идет о скорости – задача тривиальная. Достаточно перебрать все числа от 2 до \sqrt{A} и для каждого из них проверить делимость числа A .

Проблема такого тривиального алгоритма заключается в большом количестве операций. Количество потенциальных делителей растет существенно медленнее числа, но для чисел, состоящих из нескольких десятков разрядов, количество претендентов настолько велико, что задача поиска делителя становится практически не решаемой. Для 100-значного числа количество потенциальных делителей выражается 50-значным числом.

Еще больше усложняет задачу необходимость выполнения очень трудоемкой операции поиска остатка от деления. Несколько облегчить задачу может учет признаков делимости на 2, 3, 5, 7 и т. д. Но эффект от применения признаков делимости очень быстро сойдет на нет с ростом исходного числа. Более эффективно было бы проверять на свойство быть делителем только простые числа. Но тогда нужны алгоритмы получения простых чисел. Такие алгоритмы существуют, например решето Эратосфена, но в этом случае одна проблема заменяется на другую, и при этом не происходит никакого уменьшения сложности или трудоемкости вычислений.

Если мы упомянули решето Эратосфена (алгоритм детально будет рассмотрен в этой главе, но позже), то следует упомянуть об одной тривиальной ошибке неопытных исследователей в задаче факторизации. Если есть алгоритм поиска простых, а он есть, и не только алгоритм Эратосфена, то возникает искушение вычислить достаточно большое количество простых, сохранить их в файл и затем анализируемое число просто попытаться поделить на каждое из них. Если бы такой подход был возможен, то проблемы факторизации просто не существовало бы. Чтобы понять, что нам мешает воспользоваться таким большим файлом, попытаемся дать хотя бы грубую оценку количества простых чисел.

Предположим, поставлена задача найти делители 200-значного числа. Количество знаков его возможных делителей ограничено 100 знаками. Следовательно, необходимо оценить количество простых, содержащих не более 100 знаков. Известно, что количество простых, не больших чисел N , оценивается функцией

$$f(N) = \frac{N}{\ln(N)}.$$

Наше N – 100-значное. Точная оценка не нужна, поэтому пусть $N = 10^{100}$. Тогда согласно приведенной выше формуле:

$$f(10^{100}) = \frac{10^{100}}{\ln(10^{100})} = \frac{10^{100}}{100 \ln(10)} \approx 10^{98}.$$

К сожалению, количество простых выражается астрономическим числом. Такое количество чисел невозможно вычислить за разумное время, и их негде будет хранить. Файл окажется слишком велик. Отсюда мораль – проблема разложения числа на множители не решается кавалерийской атакой. Это именно тот случай, когда наращивание мускулов в форме вычислительных ресурсов не дает ровным счетом ничего.

Проблема трудоемкости всех известных на сегодня алгоритмов факторизации несет в себе один положительный момент. Благодаря этой высокой трудоемкости удалось построить практически непреступный алгоритм шифрования с открытым ключом (алгоритм RSA). Подробнее об этом в [6], [7], [20].

Выделение полного квадрата (алгоритм Ферма)

Алгоритм факторизации, именуемый алгоритмом выделения полного квадрата, опирается на тот факт, что для любого числа A можно найти два целых числа B и C , таких что $B^2 - C^2 = A$. Факт неочевидный, попробуем его доказать.

Так как A есть число составное, то очевидно $A = m \cdot n$, где m и n – положительные целые числа. Пусть m и n достаточно велики, то есть существует некоторое количество положительных целых, меньших m и n . Тогда можно записать, что

$$A = (x_0 + x)(y_0 + y).$$

Здесь x_0 и y_0 – небольшие числа, такие что $x = m - x_0$; $y = n - y_0$. Перемножим выражения в скобках и получим:

$$xy_0 + yx_0 + xy + x_0y_0 = A.$$

Выражение, полученное слева от знака равенства, называется квадратичной формой. Для таких выражений существует последовательность преобразований, приводящих их к нормальным формам (стандартным). Выполним несколько простых преобразований. Для начала проведем замену переменных: $x = u + v$; $y = u - v$ получим следующее:

$$(u + v)y_0 + (u - v)x_0 + (u + v)(u - v) + x_0y_0 = A;$$

$$uy_0 + vy_0 + ux_0 - vx_0 + u^2 - v^2 + x_0y_0 = A;$$

$$(u^2 + uy_0 + ux_0) + (-v^2 - vx_0 + vy_0) + x_0y_0 = A;$$

$$(u^2 + uy_0 + ux_0) - (v^2 + vx_0 - vy_0) + x_0y_0 = A;$$

$$(u^2 + u(y_0 + x_0)) - (v^2 + v(x_0 - y_0)) + x_0 y_0 = A.$$

Выражения в скобках дополним до полного квадрата

$$(u^2 + u(y_0 + x_0)) + (y_0 + x_0)^2/4 - (v^2 + v(x_0 - y_0)) + (x_0 - y_0)^2/4 - (y_0 + x_0)^2/4 + (x_0 - y_0)^2/4 + x_0 y_0 = A;$$

$$(u + (y_0 + x_0)/2)^2 - (v + (x_0 - y_0)/2)^2 - (y_0 + x_0)^2/4 + (x_0 - y_0)^2/4 + x_0 y_0 = A.$$

Преобразования над свободным членом выполним отдельно

$$-(y_0 + x_0)^2/4 + (x_0 - y_0)^2/4 + x_0 y_0;$$

$$-(y_0 + x_0)^2 + (x_0 - y_0)^2 + 4x_0 y_0.$$

Раскрыв скобки и приведя подобные, убедимся, что свободный член равен нулю. В оставшемся выражении выполним подстановку:

$$B = u + (y_0 + x_0)/2;$$

$$C = (v + (x_0 - y_0)/2).$$

И получим то, что и требовалось доказать: $B^2 - C^2 = A$.

Очевидно, что числа $(B - C)$ и $(B + C)$ являются делителями числа A . Также очевидно, что число $B^2 - A$ является полным квадратом. Таким образом, задача поиска пары делителей сводится к задаче перебора величин B . Поиск начинают с величины

$\sqrt{A} + 1$ – это наименьшее значение, при котором величина $B^2 - A$ будет положительной. Далее B увеличивается на 1 до тех пор, пока выражение $B^2 - A$ не окажется полным квадратом. Если полный квадрат найден, то A представляется как

$$(B - \sqrt{B^2 - A})(B + \sqrt{B^2 - A}).$$

Такое разложение может оказаться тривиальным (A и 1). Если найдено тривиальное разложение, то поиск должен быть продолжен.

Пример.

Факторизуем число 313591, $\sqrt{313591} \approx 560$, $\sqrt{560^2 - 313591} = 3$, тогда

$$(560 - \sqrt{560^2 - 313591}) = 557 \text{ и } (560 + \sqrt{560^2 - 313591}) = 563.$$

Разложение на множители выполнено за один шаг. Можно сказать, повезло. Везение, это конечно, не случайное. Множители числа 313591 расположены очень близко от квадратного корня этого числа, что и стало причиной везения.

Может показаться, что такая схема расчетов похожа на стрельбу с закрытыми глазами, но это не совсем так. Вероятность везения достаточно велика. Подробный математический анализ проблемы везения можно сделать на базе теории квадратичных вычетов. Это не является нашей целью, отметим только, что условие нетривиальности для найденных X и y выражается следующим неравенством:

$$1 < \text{НОД}(A, X \pm y) < A.$$

Иначе говоря, $\text{НОД}(A, X + y)$ есть делитель A , или $\text{НОД}(A, X - y)$ – делитель A . Это условие выполняется с вероятностью 0.5, если A представимо в виде произведения двух простых, и с еще большей вероятностью, если A представимо в виде произведения большего количества простых.

Задание для самостоятельной работы

Напишите программу поиска делителей алгоритмом Ферма для чисел в пределах типа INTEGER.

Квадратичное решето

Метод Ферма эффективен, если делители числа A располагаются близко от \sqrt{A} , если это не так, то метод может оказаться не более эффективным, чем простой перебор всех возможных делителей. Ключ к решению проблемы был дан выше. Повторим и выделим это утверждение:

«Это условие выполняется с вероятностью 0.5, если A представимо в виде произведения двух простых, и с еще большей вероятностью, если A представимо в виде произведения большего количества простых».

Отсюда следует красивая идея. Если полный квадрат нельзя найти перебором, то его следует построить. Соберем несколько значений $B^2 - A$, каждое из которых не является полным квадратом, и перемножим их. В результате умножения может получиться полный квадрат. Обозначим его через y^2 .

Пусть $X = \prod x_i$, так что $\prod (x_i^2 - A) = y^2$, тогда $X^2 - y^2$ кратно A . Следовательно, имеем, что $(X - y)(X + y)$ есть факторизация величины, кратной A . И, следовательно, либо $(X - y)$, либо $(X + y)$ делится на какой-либо делитель A . Выявить этот делитель можно, вычисляя $\text{НОД}(X - y, A)$ либо $\text{НОД}(X + y, A)$. Делитель вновь может оказаться тривиальным, то есть равным 1 или A . Если это так и будет, то процедуру можно повторить, если найден нетривиальный делитель, то нам повезло.

Собственно, это только базовая идея. Мы уже знаем, что вероятность попадания на полный квадрат ненулевая. А если факторизируемое число представимо произведением большего количества простых, чем два, то вероятность возрастает. Дальнейшее совершенствование метода лежит в несколько иной плоскости. Мы не будем увеличивать количество множителей, а попробуем подобрать сразу нужные.

Поиск необходимого множества делителей

Наверное, ясно, что не все равно, какой набор чисел $B^2 - A$, не являющихся полным квадратом, брать для пробного перемножения. Поэтому следующая задача звучит так: найти такие $B^2 - A$ (величины B различны), что их произведение является полным квадратом. В общем виде: дано множество целых чисел. Выделить подмножество, такое что его произведение будет полным квадратом.

Выбранные целые числа не обязаны быть простыми, но они могут разлагаться на простые сомножители. Придется предположить, что такое разложение для выбранных целых сделать несложно. Для того чтобы перемножить несколько целых чисел, достаточно сложить показатели простых в их разложениях. Отсюда вывод – произведение будет полным квадратом, если показатель каждого из простых, входящих в разложение, окажется четным числом. Если ряд простых чисел пронумеровать, то любое целое можно представить в виде вектора (a_1, a_2, \dots, a_n) , где a_k – это показатель степени, с которым простое с номером k входит в разложение целого. Тогда произведение двух целых естественным образом сводится к сложению двух векторов показателей простых. Так как нас интересует только четность, то компоненты вектора суммы показателей (произведения чисел) можно брать по модулю 2. Тогда задача сводится к поиску такой комбинации, которая давала бы нулевой вектор $(0, 0, \dots, 0)$. Сказанное можно записать в виде матричного уравнения: $Tx = 0$. Здесь неизвестным является вектор линейной комбинации, а матрица T состоит из векторов показателей, записанных по столбцам. Поясним сказанное на примере:

Пусть даны следующие целые числа: (34, 4, 784, 56, 115). Построим их разложение на простые:

$$34 = 2 \cdot 17$$

$$4 = 2^2$$

$$784 = 2^4 \cdot 7^2$$

$$56 = 2^3 \cdot 7$$

$$115 = 5 \cdot 23$$

Имеем следующий ряд простых: (2, 5, 7, 17, 23). Матрица показателей выглядит следующим образом:

$$\begin{pmatrix} 1 & 2 & 4 & 3 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Матрица по модулю 2:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Отсюда видно, по крайней мере, одно нетривиальное решение $(0, 1, 1, 0, 0)$. Сложения и вычитания выполняются по правилам матричной арифметике и по модулю 2.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} =$$

Распишем матричные операции:

$$1. (1 * 0 + 0 * 1 + 0 * 1 + 1 * 0 + 0 * 0) \bmod 2 = 0$$

$$2. (0 * 0 + 0 * 1 + 0 * 1 + 0 * 0 + 1 * 0) \bmod 2 = 0$$

И так далее. Полученное решение соответствует произведению $4 * 784 = 3136$.

Это действительно полный квадрат: $\sqrt{3136} = 56$.

Проблему нахождения произведения являющегося, полным квадратом, мы описали. Вернемся к исходной проблеме факторизации. Обозначим факторизуемое число как A . Построим несколько чисел вида $x^2 - A$; пусть x изменяется от $\lceil \sqrt{A} \rceil + 1$ до числа M . M должно быть достаточно велико, для того чтобы матричное уравнение, записанное выше, имело хотя бы одно нетривиальное решение. Насколько оно велико, сказать нельзя, поэтому в реализации метода необходимо предусмотреть возможность управления величиной M . Полученные числа $x^2 - A$ раскладываются на простые множители. Эта задача не равнозначна исходной, так как числа $x^2 - A$ при небольших M существенно меньше исходного A . Кроме того, ряд последовательных чисел можно раскладывать на множители, используя полученную информацию об уже факторизованных числах ряда, но об этом позже.

У некоторых чисел последовательности $x^2 - A$ могут обнаружиться большие делители. Это плохо, так как затруднит решение матричного уравнения. Чем простые множители меньше, тем быстрее работа алгоритма. Но, оказывается, эта проблема не особенно значительна. Оказывается, числа, имеющие в разложении большие простые, можно просто отбросить. Для успешной работы достаточно ограничиться числами, чьи множители не превосходят небольшого B . Такие числа называются B -гладкими.

Таким образом, в реализации необходимо учесть возможность управления двумя величинами M и B . При слишком сложном матричном уравнении эти величины можно уменьшать, в случае неуспеха факторизации – увеличивать. Успех заключается в обнаружении нетривиального полного квадрата. В качестве примера

выполним факторизацию числа: $1009 \cdot 1069 = 1078621$.

$\sqrt{1078621} \approx 1038$, пусть $M = 1055$, построим таблицу:

Таблица 3.1

X	1039	1040	1041	1042	1043	1044	1045	1046	1047
x² – A	900	2979	5060	7143	9228	11315	13404	15495	17588

Таблица 3.2

X	1048	1049	1050	1051	1052	1053	1054	1055	
x² – A	19683	21780	23879	25980	28083	30188	32295	34404	

Построим каноническое разложение:

Таблица 3.3

$900 = 2^2 \cdot 3^2 \cdot 5^2$	$11315 = 5 \cdot 31 \cdot 73$	$21780 = 2^2 \cdot 3^2 \cdot 5 \cdot 11^2$	$32295 = 3 \cdot 5 \cdot 2153$
$2979 = 3^2 \cdot 331$	$13404 = 2^2 \cdot 3 \cdot 1117$	$23879 = 23879$	$34404 = 2^2 \cdot 3 \cdot 47 \cdot 61$
$5060 = 2^2 \cdot 5 \cdot 11 \cdot 23$	$15495 = 3 \cdot 5 \cdot 1033$	$25980 = 2^2 \cdot 3 \cdot 5 \cdot 433$	
$7143 = 3 \cdot 2381$	$17588 = 2^2 \cdot 4397$	$28083 = 3 \cdot 11 \cdot 23 \cdot 37$	
$9228 = 2^2 \cdot 3 \cdot 769$	$19683 = 3^9$	$30188 = 2^2 \cdot 7547$	

Попробуем ограничить анализ значением $B = 47$. Тогда для составления уравнения могут быть использованы следующие числа:

$$900 = 2^2 \cdot 3^2 \cdot 5^2$$

$$5060 = 2^2 \cdot 5 \cdot 11 \cdot 23$$

$$19683 = 3^9$$

$$21780 = 2^2 \cdot 3^2 \cdot 5 \cdot 11^2$$

$$28083 = 3 \cdot 11 \cdot 23 \cdot 37$$

Полученные разложения представляют несколько возможностей получения полного квадрата. Например, полным квадратом является число 900. И уже оно дает решение:

$$(1039 - \sqrt{900}) \cdot (1039 + \sqrt{900}) \Rightarrow 1009 \cdot 1069.$$

Положим, что этого решения невооруженным глазом не видно. Попробуем получить тот же результат решив, матричное уравнение. Итак, имеет место следующий ряд простых: (2, 3, 5, 11, 23, 31). Составим матрицу из показателей:

$$\begin{pmatrix} 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Транспонируем матрицу и приводим её по модулю 2:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Приведем матрицу к диагональному виду методом Гаусса, складывая строки по модулю 2. Сложение по модулю 2 означает замену нечетного элемента на 1 и четного на 0. Не будем приводить здесь все необходимые выкладки, они достаточно просты, запишем лишь конечный результат.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Отсюда следует: $x_5 = 0$; $x_4 = x_5 = 0$; $x_2 = x_4 = 0$; $x_3 = x_5 = 0$; $x_1 = \{0, 1\}$.

Вектор решения таков: $(x \bmod 2, 0 \ 0 \ 0 \ 0)$, где x – целое, таким образом, алгебраические выкладки привели нас к тому же результату. Полный квадрат можно составить единственным образом, и этот полный квадрат есть число 900.

Будем считать, что схема основных расчетов ясна, осталось одно слабое место. Выше мы предполагали, что числа, включенные в базу, легко разлагаются на множители. На самом деле это слишком сильное предположение. Для того чтобы метод стал действительно эффективным, необходим способ определения гладких чисел без разложения на множители.

Заметим, что значение B относительно невелико. Это дает возможность идти не от чисел базы, а от последовательности простых. Идея такова: выше предполагаемые под базу числа раскладывались на множители, этим самым решался вопрос гладкости и одновременно определялись простые делители. Поступим наоборот. Возьмем последовательность простых: $2, 3, 5, \dots L < B$. Те из пробных чисел, которые являются гладкими, очевидно, делятся на простые из факторной базы. Заметим, что базой мы называем числа вида $x^2 - A$, а факторной базой – простые делители чисел из базы.

Более того, гладкое число можно делить на простые из факторной базы до единицы. Процесс будет представлять собой процедуру деления всех пробных чисел на 2, затем на 3 и т. д. для всех чисел базы. Проведем этот процесс для нашего примера:

Таблица 3.4

900	2979	5060	7143	9228	11315	13404	15495	17588
19683	21780	23879	25980	28083	30188	32295		

Если $B = 47$, то факторная база следующая: (2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47).

Шаг 1. Деление на 2:

Таблица 3.5

225	2979	1265	7143	2307	11315	3351	15495	4397
19683	5445	23879	6495	28083	7547	32295		

Шаг 2. Деление на 3:

Таблица 3.6

25	331	1265	2381	769	11315	1117	5165	4397
1	605	23879	2165	9361	7547	10765		

Шаг 3. Деление на 5:

Таблица 3.7

1	331	253	2381	769	2263	1117	1033	4397
1	121	23879	433	9361	7547	2153		

Все остальные шаги. Деление на остальные простые:

Таблица 3.8

1	331	1	2381	769	73	1117	1033	4397
1	1	23879	433	1	7547	2153		

Числа, превратившиеся в единицы, и есть гладкие с границей $B = 47$. Если в процессе делений записывались делители, то тем самым было получено и разложение каждого числа. Мы достаточно подробно описали технику работы и основные идеи. Программа, учитывающая возможные нюансы в управлении величинами M и B , будет достаточно велика, поэтому если пожелаете, можете написать свою реализацию.

Алгоритм Полларда

Во второй главе уже были приведены примеры того, как совершенно определенные величины вычисляются с использованием случайных чисел. Случайные процессы можно задействовать и в задаче факторизации. Вернемся ненадолго к простому перебору. Для поиска делителя числа N строится последовательность возможных

делителей от 2 и до \sqrt{N} . Эта последовательность хороша тем, что она закономерна. Зная очередное число последовательности, мы легко найдем новое, прибавив к очередному единицу, и, кроме того, результат окажется достигнут гарантированно за конечное время. Плоха она тем, что, скорее всего, окажется слишком длинной. Но оказывается, если мы согласны пожертвовать гарантированным результатом, то можно построить более короткую последовательность, завершающуюся делителем, но достижимость результата будет подчиняться уже какому-то вероятностному правилу. Конечно, вероятностный подход нельзя признать идеальным средством, но если вероятность успеха достаточно высока, то игнорировать такую возможность неразумно. Ведь найденный делитель не перестанет быть делителем только от того, что его нашли не в закономерном процессе. Именно такую вероятностную природу имеет предлагаемый в этом параграфе алгоритм Полларда. Его идея основана на двух простых фактах:

Факт 1: если $\text{НОД}(x - x', N) > 1$, где x и x' – два натуральных числа, то, очевидно, найденный НОД и есть делитель числа N . А именно предположим, d – делитель числа N . Выберем два целых x и x' – таких, что

$$x \equiv x' \pmod{d}; \quad 1 < \text{НОД}(x - x', N) < N.$$

Факт 2: пусть $N = pq$, где p и q – простые числа, и пусть x, x' – натуральные, такие что $x \not\equiv x' \pmod{p}$, тогда:

$$(x \equiv x' \pmod{d}) \Leftrightarrow \text{НОД}(x - x', N) = p),$$

Таким образом, для реализации наших целей необходимо выбрать последовательность из некоторого количества случайных чисел: x_0, x_1, \dots, x_k , перебрать все их

возможные пары и найти пару, такую что $1 < \text{НОД}(x - x', N) < N$.

Конечно, проверка каждого с каждым очень затратна, настолько затратна, что простая проверка на делимость с простыми окажется не менее эффективной. Для повышения эффективности метода надо избавиться от необходимости проверки каждого с каждым. В 1978 г. Поллард предложил интересную идею решения проблемы.

Рассмотрим функцию $f(x_i) = f(x_{i-1})$, генерирующую случайную циклическую последовательность: x_0, x_1, \dots, x_{i-1} , начинающуюся со случайного числа. Функция при этом должна быть не просто циклической, а с периодом L . То есть $x_{i+L} = x_i$, здесь L – длина периода, i – индекс вхождения.

Очевидно, что если x_{i+L} совпадает с x_i , то справедливо следующее утверждение:

$$\forall k > 0, \quad x_{i+k+L} = x_{i+L},$$

Этот факт позволяет уйти от сравнения x_i со всеми x_0, x_1, \dots, x_{i-1} . Количество сравнений можно существенно уменьшить, если сравнения проводить по следующей схеме:

$$\begin{array}{c|c|c|c|c} x_0 & x_1 & x_3 & x_7 & \dots \\ \hline x_1 & x_2, x_3 & x_4, x_5, x_6, x_7 & x_8 \dots x_{15} & \end{array}$$

Такова общая идея. В качестве основы функции $f()$ принято брать многочлен, такой чтобы вычисление его значений не требовало значительных ресурсов. Вполне достаточен может быть многочлен $x^2 + 1$. Общий вид функции $f()$ для анализа числа N может быть таким:

$$x' = x^2 + 1(\text{mod } N),$$

Построим реализацию. Ядро программы – цикл, вычисляющий функцию $f()$ до получения нетривиального делителя (большого единицы):

```
x:=N DIV 2;
flag:=TRUE;
WHILE flag DO
  (*Тело цикла*)
END;
```

Оператор $x:=N \text{ DIV } 2$; можно и даже желательно заменить на $x:=\text{RANDOM}(N)$, где $\text{RANDOM}(N)$ – какая-либо функция, вычисляющая случайное число на отрезке $[0, N-1]$. Тело цикла – это последовательность операторов, вычисляющих последовательность чисел, описанную выше. Так как длина последовательности неизвестна, то начнем расчеты из надежды на наименьшую длину, если полученная последовательность не даст результата, то длину удвоим. Роль длины играет переменная j .

Расчет чисел последовательности необходимо прерывать в двух случаях: если текущая длина достигнута (i стало не меньше, чем j) либо найден делитель ($d > 1$). Величина d и есть искомый делитель, его значение рассчитывается алгоритмом Евклида.

```

x1:=x;
i:=1;
WHILE flag & (i<j) DO
  x:=(x*x+1) MOD N;
  d:=Евклид(N,ABS(x-x1));
  IF d>1 THEN flag:=FALSE; END;
  i:=i+1;
END;
j:=2*j;

```

И наконец, полный текст программы.

Листинг 3.2

```

MODULE Модуль;
  IMPORT StdLog, In;
  PROCEDURE Евклид (A,q0:LONGINT):LONGINT;
  VAR
    q1,c:LONGINT;
  BEGIN
    IF (q0>=A) THEN c:=q0;q0:=A;A:=c;END;
    q1:=A MOD q0;
    WHILE q1#0 DO
      c:=q1;
      q1:=q0 MOD q1;
      q0:=c;
    END;
    RETURN q0;
  END Евклид;
  PROCEDURE Поллард*;
  VAR
    N,i,j,x,x1,d:LONGINT;
    flag:BOOLEAN;
  BEGIN
    In.Open;
    In.LongInt(N);
    j:=1;
    x:=N DIV 2;
    flag:=TRUE;
    WHILE flag DO
      x1:=x;

```

```
i:=1;  
WHILE flag & (i<j) DO  
  x:=(x*x+1) MOD N;  
  d:= Евклид (N,ABS(x-x1));  
  IF d>1 THEN flag:=FALSE; END;  
  i:=i+1;  
END;  
j:=2*j;  
END;  
StdLog.Int(d);  
END Поллард;  
END Модуль.
```

Реализовать идею Полларда можно различным образом. Здесь был выбран наиболее простой и наглядный метод, но, вполне возможно, не самый лучший.

Задание для самостоятельной работы

В построенной реализации главный цикл завершает свою работу, только когда найден нетривиальный делитель. Но выше утверждалось, что метод имеет вероятностную природу. Означает ли это возможность зависания программы? Если да, то продумайте, как учесть такую возможность (зависания).

Последнее замечание. Наше изложение алгоритмов достаточно краткое, и охват темы всего лишь достаточен для понимания сути проблемы и возможности её решения. Если у вас возникнет желание более глубоко изучить тему, то это можно, например, сделать по [6], [7], [8].

Алгоритмы поиска простых чисел

Простое число – это число, делящееся только на единицу и на себя. Определение дает и несложный метод поиска простых. Необходимо записать цикл, проходящий все целые от 2 до некоторой границы, и для каждого числа пытаться найти нетривиальный (отличный от 1 и самого числа) делитель. Если такой делитель найден, то число составное, иначе – простое. Однако мы уже знаем из задачи факторизации, что поиск делителей – операция достаточно трудоемкая. Поэтому простейший алгоритм, конечно же, не самый эффективный. Его листинг ниже:

Листинг 3.3

```
MODULE Модуль;  
IMPORT In, StdLog, Math;  
PROCEDURE ПростойПеребор*;  
VAR  
  N,k,j:INTEGER;  
  flag:BOOLEAN;  
BEGIN  
  In.Open;
```

```

In.Int(N);
FOR k:=2 TO N DO
  flag:=TRUE;
  FOR j:=2 TO SHORT(ENTIER(Math.Sqrt(k))) DO
    IF k MOD j =0 THEN flag:=FALSE END;
  END;
  IF flag THEN StdLog.Int(k);END;
END;
END ПростойПеребор;
END Модуль.

```

Еще один элементарный метод поиска простых известен под названием решета Эратосфена. Решето – это способ просеивания целых чисел и отбрасывания составных. Начинается работа алгоритма с записи последовательности целых чисел начиная с 2 в большой массив. Первое число этого массива простое (это всегда число 2). После каждого шага просеивания будет найдено еще одно простое, это первое ненулевое число, следующее за уже известным простым. Шаг просеивания заключается в обнулении всех чисел, кратных очередному простому. В листинге ниже – процедура, реализующая алгоритм решета.

Листинг 3.4

```

PROCEDURE Эратосфен(a:mas;n:INTEGER);
VAR
  k,j:INTEGER;
BEGIN
  FOR k:=2 TO SHORT(ENTIER(Math.Sqrt(n))) DO
    IF a[k]#0 THEN
      FOR j:=k+1 TO n DO
        IF j MOD a[k]=0 THEN a[j]:=0;END;
      END;
    END;
  END;
  FOR k:=2 TO n DO
    IF a[k]#0 THEN StdLog.Int(a[k]); END;
  END;
END Эратосфен;

```

Задание для самостоятельной работы

Приведенная здесь реализация решета – не вполне алгоритм Эратосфена. Алгоритм Эратосфена должен обходиться без операции деления. Но в нашей реализации деление есть, оно скрыто в операции вычисления остатка. Попробуйте улучшить программу, избавившись от вычисления остатка. Вторая проблема заключается во внутреннем цикле (FOR j:=k+1 TO n DO); этот цикл выполняет слишком много операций.

Действительно, пусть очередное простое – это число 13. То есть мы уже знаем, что

обнулять необходимо каждое 13-ое, но указанный цикл FOR будет добросовестно обрабатывать все числа ряда от очередного простого с шагом 1.

Решето Аткина

Алгоритм был разработан А. О. Л. Аткиным и Д. Ю. Берштайном и представляет собой более совершенный вариант решета Эратосфена. Алгоритм сводит исследование всех целых к исследованию чисел по модулю 60. Далее исследуются остатки от деления на небольшое количество делителей. А именно на 4, 6 и 12. При этом:

Деление на 4. Если число n при делении на 4 имеет остаток 1, то число простое тогда и только тогда, когда количество решений уравнения $4x^2 + y^2 = n$ нечетное и само число не является квадратом целого.

Деление на 6. Если число n при делении на 6 имеет остаток 1, то число простое тогда и только тогда, когда количество решений уравнения $3x^2 + y^2 = n$ нечетное и само число не является квадратом целого.

Деление на 12. Если число n при делении на 12 имеет остаток 11, то число простое тогда и только тогда, когда количество решений уравнения $3x^2 - y^2 = n$ нечетное и само число не является квадратом целого.

Листинг 3.5

```
MODULE Модуль;
  IMPORT StdLog, In, Math;
  PROCEDURE Аткин*;
  VAR
    N, n, limit, sqr_lim, x2, y2, i, j: INTEGER;
    Простое: ARRAY 10000 OF BOOLEAN;
  BEGIN
    (*Инициализация массива решета*)
    In.Open;
    In.Int(limit);
    sqr_lim:=SHORT(ENTIER(Math.Sqrt(limit)));
    FOR i:=0 TO limit DO Простое [i]:=FALSE; END;
    Простое [2]:=TRUE;
    Простое [3]:=TRUE;
    x2:=0;
    FOR i:=1 TO sqr_lim DO
      x2:=x2+2*i-1;
      y2:=0;
      FOR j:=1 TO sqr_lim DO
        y2:=y2+2*j-1;
        n:=4*x2+y2;
        IF ((n<=limit) & ((n MOD 12=1) OR (n MOD 12=5))) THEN
          Простое [n]:=~ Простое [n];
        END;
      END;
    END;
```

```

n:=n-x2;
IF (n<=limit) & (n MOD 12=7) THEN
  Простое [n]:=~ Простое [n];
END;
n:=n-2*y2;
IF (i>j) & (n<=limit) & (n MOD 12=11) THEN
  Простое [n]:=~ Простое [n];
END;
END;
END;
FOR i:=5 TO sqr_lim DO
  IF Простое [i] THEN
    n:=i*i;
    j:=n;
    WHILE j<=limit DO
      Простое [j]:=FALSE;
      j:=j+n;
    END;
  END;
END;
FOR i:=2 TO limit DO
  IF Простое [i] THEN StdLog.Int(i); END;
END;
END Аткин;
END Модуль.

```

Решето Сундарамы

Алгоритм разработан индийским математиком С. П. Сундарамом в 1934 году. Алгоритм исключает из ряда натуральных чисел $1 \dots N$ все числа вида $i + j + 2ij$, где $i = 1, 2, 3, \dots [N/3]$; $j = 1, 2, 3, \dots [(N - i)/(2i + 1)]$. Идея решета Сундарамы конечно же не так очевидна, как идея решета Эратосфена, но все же достаточно проста. Действительно, проверять на простоту есть смысл только нечетные числа, то есть числа вида $2m + 1$. Если это число составное, то, очевидно, оно представимо в виде произведения двух нечетных, то есть $2m + 1 = (2i + 1)(2j + 1)$, откуда следует $m = 2ij + i + j$. Следовательно, все такие m – это составные, если их исключить, то останутся простые. К сожалению, несколько большая сложность алгоритма приводит к меньшему быстродействию, и этот алгоритм, несмотря на красивую арифметическую идею, менее эффективен, нежели решето Эратосфена.

Задание для самостоятельной работы

Напишите программу, реализующую решето Сундарамы. Приведите математическое обоснование верхней границы для индекса j .

Тесты простоты

Методы построения простых на практике не всегда полезны, зачастую бывает необходимо решить несколько иную задачу: дано некоторое число, требуется выяснить, является ли оно простым. Алгоритмы, решающие такую задачу, называются тестами на простоту. Среди тестов на простоту можно выделить тесты общего назначения, работающие с любыми числами, и специальные тесты, анализирующие числа особого вида. В качестве примера теста общего вида можно указать теорему Вильсона, утверждающую, что число p является простым тогда и только тогда, когда

$$(p-1)!+1 \text{ делится на } p.$$

Реализовать алгоритм, использующий теорему Вильямса, несложно, но ясно, что такой алгоритм не имеет реального практического применения из-за высокой трудоемкости операции вычисления факториала. Более интересен тест Миллера-Рабина. В его основе лежит теорема Рабина, утверждающая, что:

Теорема Рабина. Пусть m – нечетное число, большее 1. Тогда:

$$m-1 = t * 2^s,$$

где t нечетно. Целое число q называется свидетелем простоты числа m , если выполняются следующие условия:

- m не делится на q ;
- $q^t \equiv 1 \pmod{m}$, или существует целое k , где $0 \leq k < s$, такое что $q^{2^k t} \equiv -1 \pmod{m}$.

Согласно теореме, составное нечетное m имеет не более $\varphi(m)/4$ свидетелей. Здесь $\varphi(m)$ – функция Эйлера.

Функция Эйлера. Функция, равная количеству чисел, взаимно простых с её аргументом. Если аргумент функции – число m , то её значение – количество взаимно простых с m в ряду $0, 1, 2, \dots, m-1$. Примеры:

$$\varphi(1) = 1 \quad \varphi(2) = 1$$

$$\varphi(3) = 2 \quad \varphi(4) = 2$$

$$\varphi(5) = 4 \quad \varphi(6) = 2$$

Понятие вычета. В данном параграфе и ниже будут использоваться термин *вычет* и запись вида $a \equiv b \pmod{m}$. Запись читается так: числа a и b сравнимы по модулю m . Это означает, что остаток от деления a на m равен остатку от деления b на m . Числа, имеющие по некоторому модулю одинаковый остаток, образуют класс чисел, называемый классом вычетов, а каждое число этого класса соответственно называется вычетом.

Реализация

В качестве подготовительной работы к главному циклу необходимо представить величину $m-1$ в виде $t \cdot 2^s$. Разложение заключается в вычислении величин t и s . Следующий фрагмент решает эту задачу:

```
s:=0;t:=m-1;
WHILE t MOD 2=0 DO
  t:=t DIV 2;
  s:=s+1;
END;
```

Подготовительная работа выполнена, теперь основной цикл. Его работа заключается в попытке найти свидетелей простоты. Чем свидетелей больше, тем свидетельство надежнее, мы ограничимся десятью:

```
FOR k:=1 TO 10 DO
  a:=Ran.Int(m-3)+2;
  (*Если не свидетель, то будет возвращена ложь*)
END;
RETURN TRUE;
```

Если цикл пройти удалось, то необходимое количество свидетелей найдено, и следовательно, можно сделать вывод о высокой вероятности того, что число простое. Поэтому по завершении основного цикла возвращается истина. Проверка свидетель—не свидетель выполняется по правилам, предписанным теоремой Рабина, поэтому детальное обсуждение тела цикла опустим. Единственное, обратите внимание на следующий фрагмент:

```
x:=1;
FOR j:=1 TO t DO x:=x*a; END;
```

Здесь вычисляется большая степень. Для учебного примера это не страшно, но для сколько-нибудь интересных чисел этот фрагмент станет причиной неработоспособности программы.

Задание для самостоятельной работы

Решите проблему, указанную абзацем выше.

А сейчас полный листинг решения:

Листинг 3.6

```
MODULE Модуль;
  IMPORT StdLog, In, Ran := Info21sysRandom, Math;
  PROCEDURE Рабин(m:INTEGER):BOOLEAN;
  VAR
    s, t, a, k, j, x:INTEGER;
  BEGIN
    s:=0;t:=m-1;
```

```
WHILE t MOD 2=0 DO
  t:=t DIV 2;
  s:=s+1;
END;
FOR k:=1 TO 10 DO
  a:=Ran.Int(m-3)+2;
  x:=1;
  FOR j:=1 TO t DO x:=x*a; END;
  x:=x MOD m;
END;
IF (x#1) & (x#m-1) THEN
  j:=1;
  WHILE (x#m-1) & (j<s) DO
    x:=x*x MOD m;
    IF x=1 THEN RETURN FALSE; END;
    j:=j+1;
  END;
  IF (x#m-1) THEN RETURN FALSE; END;
END;
RETURN TRUE;
END Рабин;
PROCEDURE Главная*;
VAR
  m:INTEGER;
BEGIN
  In.Open;
  In.Int(m);
  IF Рабин(m) THEN
    StdLog.String('Вероятно простое');
  ELSE
    StdLog.String('Составное');
  END;
END Главная;
END Модуль.
```

Техническое замечание. В нашей реализации для генерации случайного числа используется модуль **Info21sysRandom**, вы можете использовать любую функцию, выдающую случайное целое число, если она дает число в нужном интервале.

Числа Мерсенна

Алгоритм Эратосфена и другие рассмотренные выше, безусловно, хороши для поиска небольших простых. Но большие числа потребуют слишком много памяти. Существует ряд методов, опирающихся на некоторые математические закономер-

ности и позволяющих получать большие числа. Некоторые из этих методов носят вероятностный характер. Таким, например, является способ ловли простых среди чисел Мерсенна. Общая идея такова: простые числа представляют собой последовательность. Было бы хорошо, если математике была бы известна общая формула такой последовательности в какой-нибудь легко вычисляемой форме, например в виде возвратной последовательности, то есть последовательности, в которой очередной член определяется через некоторое количество предыдущих (примеры возвратных последовательностей: арифметическая и геометрическая прогрессии, числа Фибоначчи), но, к сожалению, такой формулы нет. Но если неизвестна «родная» последовательность, то можно придумать последовательность, такую что простые числа в ней будут встречаться часто, настолько часто, что плотность простых в этой последовательности будет выше плотности простых в натуральном ряду. Одной из таких последовательностей являются числа Мерсенна: числа вида $2^n - 1$. Например, в первой сотне натуральных числами Мерсенна являются: 1, 3, 7, 15, 31, 63. На шесть чисел – четыре простых. И далее плотность простых в их ряду настолько высока, что числа Мерсенна уже долгое время держат лидерство среди методов вычисления больших простых. На сегодня самое большое простое, найденное в ряду Мерсенна, – это число

$$2^{43112609} - 1.$$

Длина этого числа составляет 12 978 189 десятичных цифр. Можно определить кратные числа. Например, двойное:

$$2^{2^n - 1} - 1.$$

Однако высокая плотность простых в ряду чисел Мерсенна не избавляет нас от необходимости каждое конкретное число проверять на простоту. Поэтому вернемся к проблеме теста на простоту и рассмотрим специализированный тест, предназначенный именно для чисел такого вида.

Тест Люка-Лемера

В основе теста лежит тот факт, что простота числа $2^p - 1$ означает простоту числа p . И кроме того, оказывается, что:

для простого $p \geq 3$ число $2^p - 1$ является простым тогда и только тогда, когда оно делит число L_{p-1} , где числа L_k определяются следующим рекуррентным соотношением:

$$L_1 = 4, \quad L_{k+1} = L_k^2 - 2 \quad .$$

Однако числа L растут слишком быстро, поэтому для реальных расчетов используется следующая формула:

$$L_{k+1} = (L_k^2 - 2) \bmod (2^p - 1).$$

Последнее значение

$$(L_{p-1}^2 - 2) \bmod (2^p - 1)$$

называется вычетом Люка-Леммера. И теорема получает немного иную формулировку: число $2^p - 1$ является простым тогда и только тогда, когда p – простое и вычет Люка-Леммера равен нулю.

Реализация

Договоримся в качестве аргумента использовать величину p – показатель степени. Тогда до основного цикла необходимо выполнить вспомогательную работу по расчету соответствующего числа Мерсенна. Эту работу выполняет нижеследующий фрагмент:

```
Мр:=1;  
FOR k:=1 TO p DO Мр:=Мр*2; END;  
Мр:=Мр-1;
```

Главный же цикл тривиален. На каждом его шаге вычисляется очередной вычет, его работа завершается вычетом Люка-Леммера.

Листинг 3.7

```
MODULE Модуль;  
  IMPORT StdLog, In;  
  PROCEDURE Люк*;  
  VAR  
    p, k, L, Мр: INTEGER;  
  BEGIN  
    In.Open;  
    In.Int(p);  
    Мр:=1;  
    FOR k:=1 TO p DO Мр:=Мр*2; END;  
    Мр:=Мр-1;  
    k:=2;  
    L:=4;  
    WHILE k<p DO  
      L:=(L*L-2) MOD Мр;  
      k:=k+1;  
    END;  
    IF L=0 THEN  
      StdLog.String('Простое');  
    ELSE  
      StdLog.String('Составное');  
    END;  
  END Люк;  
END Модуль.
```

Числа Ферма

Выше были рассмотрены числа специального вида, ряд которых претендует на высокую плотность простых. Предмет нового параграфа – также числа специального вида, с даже еще большей претензией – числа Ферма, – числа вида

$$2^{2^m} + 1.$$

Ферма выдвинул гипотезу, что вообще все числа такого вида – простые. И действительно, первые числа 3, 5, 17, 257, 6553 – простые, и лишь в отношении пятого числа 4294967297 Эйлер в 1732 г. нашел разложение: $4294967297 = 641 \cdot 6700417$. Далее нужен специальный критерий, так как числа Ферма растут очень быстро. Мы рассмотрим специальный тест, но сначала небольшое замечание:

Зачем нужна высокая плотность. Собственно, об этом уже говорили: чем выше плотность простых, тем быстрее можно найти число. Однако простые численные эксперименты показывают, что среди натуральных чисел с ростом значений плотность простых падает. Ситуацию резко усугубляет следующий факт теории чисел: для любого числа M найдется отрезок натурального ряда длины M , вообще не содержащий простых. А сейчас тест:

Тест Пепина

Число Ферма $F_n = 2^{2^n} + 1$ является простым тогда и только тогда, когда

$$3^{\frac{F_n-1}{2}} \equiv -1 \pmod{F_n}.$$

Задание для самостоятельной разработки

Если в формулировке теоремы не все понятно, то обратитесь к теории чисел, а реализация теста остается на самостоятельную работу.

Псевдослучайные числа

В различных технических и математических приложениях часто необходим ряд случайных чисел. Числа называются случайными в силу того, что в их ряду нет никакой закономерности, зная некоторое количество последовательных чисел ряда, нельзя вычислить следующее. Устройство или программа, получающая такие числа, называется генератором случайных чисел. О принципиальной возможности существования случайных последовательностей можно спорить. Нетрудно встретить мнение, что все процессы, существующие в природе или созданные человеком, строго закономерны. Но, во всяком случае, если это и так, то некоторые из закономерностей настолько сложны и управляются настолько большим количеством параметров, что фактически их невозможно отследить за разумное время с использованием разумного количества ресурсов.

Таких процессов достаточно много. Это, например, космическое излучение, ионизирующее радиационное излучение и т. д. Но не природные генераторы будут предметом нашего изучения. Мы посвятим параграф математическим методам построения генераторов. Здесь сразу необходимо несколько уточняющих фраз. Математические методы обладают закономерной природой, иначе в них не было бы большого смысла. Это означает, что любая числовая последовательность, полученная тем или иным математическим методом, будет содержать строгую закономерность, и, следовательно, любая такая последовательность не может быть признана случайной.

Спасает идея, высказанная в первом параграфе. Последовательность чисел, конечно же, будет закономерной, но не факт, что эту закономерность так легко удастся обнаружить. Следовательно, при построении генератора случайных чисел наша настоящая задача заключается в построении такой последовательности, которая по природе своей, будучи закономерной, выглядела бы как случайная. Такие ряды мы в дальнейшем будем называть псевдослучайными последовательностями, а числа – соответственно псевдослучайными.

Построение псевдослучайной последовательности, вообще говоря, – не тривиальная задача. Рассмотрим несколько простых методов и на их примере попробуем понять, какими свойствами должны обладать методы генерации псевдослучайных чисел. Первый претендент – алгоритм Евклида. Найдем НОД(55913, 1155):

$$\left[\frac{55913}{1155} \right] = 473; \rightarrow \left[\frac{1155}{473} \right] = 209; \rightarrow \left[\frac{473}{209} \right] = 55; \rightarrow \left[\frac{209}{5} \right] = 44; \rightarrow \left[\frac{5}{4} \right] = 11;$$

Получаем такой ряд остатков: 473, 209, 55, 44, 11. Последние три не выглядят случайными, но о закономерности для всего ряда трудно что-либо предположить, если не знать метода, которыми они получены, и не знать чисел, с которых ряд начался. То, что ряд короток, проблемы не представляет. Для любого числа N можно найти пару чисел, дающую последовательность остатков длины N . Проблема в другом. Ряд, очевидно, убывающий. А это означает, что для небольших чисел вероятность оказаться в этом ряду значительно выше, чем для больших. Это, конечно, не жестко детерминированная закономерность, но тем не менее такое свойство делает алгоритм Евклида непригодным генератором. Отсюда следует важный вывод. Отсутствие жесткой закономерности есть условие недостаточное. Необходимо обеспечить равную вероятность чисел из интервала, на котором будет определена псевдослучайная последовательность. Алгоритм Евклида такую равномерную случайность очевидно не обеспечивает.

Еще один простой способ, называемый способом середин квадратов, предложил фон Нейман. Его идея такова. Предположим, что некоторое случайное, достаточно большое число уже известно. Определить такое число совершенно не представляет проблему: так как оно первое, то оно может быть просто любым. Пусть, например, нас интересуют 5-значные случайные числа. Возьмем в качестве первого число 14 563. Возведем его в квадрат, получим 212 080 969. Возьмем из середины пять цифр 20 809. Это и будет следующее случайное число. Как и в

случае алгоритма Евклида, псевдослучайность получаемой последовательности очевидна. Каждое следующее число совершенно однозначно определяется предыдущим. Но нас это смущать не должно. Если неизвестно, как получены числа, то они выглядят вполне случайно. Но у метода есть более серьезный недостаток. Если последовательность продолжить, то может проявиться так называемый период. Периодом называется строго повторяющаяся последовательность чисел. Впрочем, для алгоритма фон Неймана можно подобрать исходное число, дающее период только после очень большого количества шагов.

Задание для самостоятельной работы

Постройте пример периодических чисел Неймана.

Появление периода повторяемости наряду с проблемой равномерности – вторая большая проблема любого генератора псевдослучайных чисел. Третья важная проблема – это количество расчетов, необходимых для перехода от одного случайного числа к другому. Два рассмотренных нами способа, не являясь очень уж честными генераторами (алгоритм Евклида совсем нечестен, он явно отдает предпочтение малым числам), по-разному решают проблему скорости. Каждый шаг алгоритма Неймана – это всегда умножение числа одной и той же длины. Алгоритм Евклида способен породить длинную последовательность только при очень длинных исходных числах. Это означает падение скорости счета при увеличении требований к длине случайной последовательности.

Требование скорости лишает большого смысла еще один интересный метод. Известно, что иррациональное число можно сколь угодно точно, то есть с любым количеством знаков после запятой, приблизить числом рациональным. Отсюда следует идея. Предположим, нам необходимо 10 трехзначных случайных чисел. Рассчитаем, с точностью до 30 знаков, какое-либо иррациональное число, например число π . Это вполне возможно, затем разобьем последовательность знаков на группы по 3 и получим требуемую случайную последовательность. Это возможно, но проблема – как уже было сказано, в скорости. Вычисление 30-го знака числа π требует значительно большего объема расчетов, чем первых, и чем дальше, тем требования к ресурсам становятся выше.

Прежде чем мы перейдем к изучению других методов получения псевдослучайных чисел, еще раз акцентируем внимание на имеющихся проблемах. Их несколько:

- Может оказаться, что некоторые числа менее случайны, нежели другие.
- Интервал, в котором располагаются числа, может содержать периоды, то есть циклически повторяющиеся числа. Конечно же, период есть всегда, потому, собственно, числа и называются псевдослучайными, речь идет о периоде, дающем о себе знать слишком скоро.
- Может оказаться, что для расчета чисел потребуется слишком много ресурсов. Генераторы псевдослучайных чисел являются всего лишь инструментальными средствами для других прикладных алгоритмов и не должны отнимать на себя слишком много расчетного времени.

- Псевдослучайные числа должны располагаться в счетном интервале равномерно. Это означает следующее: если мы проведем N вычислительных серий, в каждой из которых вычисляется одинаковое количество случайных чисел, то мы не должны обнаружить область, в которой плотность чисел будет выше средней для каждой серии.

Последний пункт – на самом деле производный от первого. Появление такой области или областей как раз и говорит о том, что одни числа менее случайны, нежели другие. Но этот критерий полезно выделить как отдельный, так как псевдослучайные генераторы работают все же с большим количеством чисел. Еще заметим, что здесь очень важно понятие серии. В каждой отдельно взятой серии, скорее всего, будут такие области с более высокой плотностью, так как постоянно и абсолютно равномерное распределение – это такой идеал, который сильно походит на регулярную закономерность, поэтому плотность распределения также должна быть случайной величиной, и – следовательно, для хорошей псевдослучайной последовательности плотности разных областей должны быть разными. Важно лишь отсутствие выделенной области, то есть такой, в которой плотность постоянно от серии к серии была бы выше, чем в других областях интервала.

Критерии правильности случайных чисел

Выше мы уже показали, что вопрос о случайности числовой последовательности – вопрос непростой. Любой программист, немного разбирающийся в математике, может изобрести алгоритм, генерирующий псевдослучайные числа, и ему придется искать ответ на вопрос, насколько хорош его генератор. Существуют методы, позволяющие получить ответ на этот вопрос. Будет называть такие методы критериями случайности. К сожалению, в силу вероятностной природы исследуемого метода ответ любого критерия также будет вероятностным. То есть получить ответ вида «Изобретенный метод дает последовательность действительно случайных чисел» не удастся. Полученный ответ может быть, в лучшем случае, только таков: «Изобретенный метод с высокой долей достоверности дает последовательность случайных чисел».

Это означает отсутствие критериев, гарантирующих, что принятый вами на вооружение генератор случайных чисел не даст сбой в конкретной прикладной задаче. Но если вам необходимо получить большую уверенность, вы можете испытать ваш генератор несколькими критериями. Каждое успешное испытание будет увеличивать уверенность в качестве генератора.

Критерий, основанный на квадратичном отклонении

Предположим, что генератор случайных чисел выдает их конечное количество. С теоретической точки зрения, это, наверное, очень ограниченное утверждение, но на практике оно не мешает. Трудно придумать прикладную задачу, нуждающуюся в бесконечно большом количестве случайных чисел. Просьба не путать бесконечное множество чисел с бесконечно длинной псевдослучайной последователь-

ностью. Бесконечно длинная псевдослучайная последовательность невозможна по определению.

Пусть множество случайных чисел состоит из N чисел. Пусть проводится серия из L испытаний, результат каждого из которых – появление одного из чисел. Если все числа равновероятны, то ожидаемая частота появления каждого из чисел будет L/N . Обозначим эту ожидаемую частоту переменной p . Реальную частоту для числа с номером k обозначим через p_k . По следующей формуле вычислим сумму, называемую суммой квадратичных отклонений:

$$V = \sum_k (p_k - p)^2.$$

Ясно, что в случае идеально случайной последовательности (так похожей на закономерную) эта величина должна быть равна нулю. На самом деле псевдослучайный генератор должен давать разные значения V . Единственное – они не должны быть слишком большими. Естественно, сейчас встает вопрос о том, что такое слишком большое V . В общем-то, постановка вопроса несколько неточна. Если величины V могут иметь различные значения, значит, они могут иметь и даже обязаны иметь и любые большие значения (в пределах, естественно, значений, могущих возникнуть в серии испытаний). Более точно было бы поставить вопрос о распределении значений V при большом количестве серий испытаний. Хорошее значение V – это значение, близкое к нулю. Плохое значение – это значение, далекое от нуля. Если генератор «честен» и у него нет любимых чисел, то, очевидно, в большом количестве серий числа V будут располагаться плотно около нуля. Тогда можно полагать, что при стремлении количества серий к бесконечности среднее значение чисел V должно стремиться к нулю.

Конечно, при экспериментальной проверке генератора говорить о стремлении какой-либо величины к нулю не вполне корректно, поэтому заменим сказанное на такое утверждение: с увеличением количества серий мы должны наблюдать, возможно, неравномерное уменьшение среднего значения чисел V . Теперь можно записать вычислительную схему:

Количество серий = 1

Выполнять многократно следующие действия:

Номер серии = 1

Пока Номер серии < Количества серий выполнять

 Рассчитать очередную серию

 Вычислить величину V_k для серии с номером k

 Номер серии = Номер серии + 1

Рассчитать среднее арифметическое чисел V_k

Вывести среднее арифметическое в поток вывода

Кол-во серий = Кол-во серий + Число

Из вычислительной схемы должно быть ясно, что серии объединяются в па-

кеты и среднее арифметическое – это величина, характеризующая разброс чисел V внутри пакета. Из изложенной теории следует, что этот разброс с увеличением размера пакета серий должен уменьшаться.

Примечание. Данный критерий есть несколько упрощенное изложение так называемого χ^2 -критерия. Детальную информацию можно получить в [2].

Задание для самостоятельной работы

Напишите реализацию предложенной выше вычислительной схемы критерия псевдослучайности.

Линейный конгруэнтный метод

Простой и достаточно быстрый метод предложен Д. Г. Лехмером в 1949 году. Выражается метод следующей рекуррентной формулой:

$$X_{n+1} = (aX_n + c) \bmod m.$$

Здесь $m > 0$; $0 \leq a < m$; $0 \leq c < m$; $0 \leq X_0 < m$.

Необходимо заметить, что идея использовать остатки по модулю достаточно распространена. Возможно, это обусловлено тем, что остатки от деления ведут себя довольно хаотично. Нельзя сказать, что какая-либо последовательность остатков будет представлять собой хорошую последовательность вроде арифметической прогрессии или хотя бы несколько сложнее. Чтобы в этом убедиться, попробуйте взять любое достаточно большое число, рассчитайте остатки от деления его на последовательные натуральные числа и попробуйте найти в этом ряду хотя бы какую-нибудь закономерность.

Остатки от деления на фиксированное число m отличаются тем полезным качеством, что они располагаются в фиксированном интервале $(0, m-1)$, и можно рассчитывать, что в данном интервале они будут располагаться равномерно. Однако на большом интервале закономерность все же проявится. На достаточно большом интервале появятся циклы повторяющихся чисел. Эти циклы называются периодами. Собственно появление периодов составляет проблему любого метода получения псевдослучайных чисел, вопрос только в том, чтобы сделать этот период по возможности большим. Очевидно, что в линейном конгруэнтном методе размер периода каким-то образом зависит от коэффициентов a , c , m . Естественно предположить, что небольшой модуль (m) быстро приведет к периоду. Проведем эксперимент. Пусть $m = 10$; $X_0 = 5$; $a = 7$; $c = 2$. Посмотрим, какой ряд дадут эти исходные данные:

$$\begin{aligned} X_0 &= 5 \\ X_1 &= (7 \cdot 5 + 2) \bmod 10 = 7 \\ X_2 &= (7 \cdot 7 + 2) \bmod 10 = 1 \\ X_3 &= (7 \cdot 1 + 2) \bmod 10 = 9 \\ X_4 &= (7 \cdot 9 + 2) \bmod 10 = 5 \end{aligned}$$

Мы получили ряд: 5, 7, 1, 9, 5.... Очевидно, далее числа начнут повторяться. Не надо никакого доказательства, чтобы увидеть, что как только число в ряду повторится, появится период. Отсюда следует, что величина модуля есть верхняя граница для длины псевдослучайной последовательности. Зависимость длины периода от прочих участвующих в расчетах коэффициентах, к сожалению, не так тривиальна. Величина m является верхней границей длины периода, и лучший выбор коэффициентов a и c может приблизить реальную длину к этой определенной границе. Существует теорема, определяющая оптимальные соотношения между коэффициентами. Мы приведем её без доказательства.

Теорема. Последовательность псевдослучайных чисел имеет период длины m тогда и только тогда, когда:

- числа c и m взаимно простые;
- $a-1$ кратно для каждого простого делителя m ;
- $a-1$ кратно 4, если m кратно 4.

Более глубоко соотношения между коэффициентами исследовать не будем. Заметим только, что наряду с возможностью получить последовательность с большой величиной периода метод довольно прост с точки зрения объема вычислений. Достаточно легко на его основе построить и несколько более сложные методы. Однако здесь следует быть очень осторожным. Усложнение линейного конгруэнтного метода не всегда дает более случайную последовательность. Например, следующее усложнение:

$$X_n = (aX_{n-1}) \bmod m1 + c) \bmod m2$$

дает не лучшую последовательность, а худшую. Но все же варианты усиления метода есть. Например, его можно обобщить до квадратного следующим образом:

$$X_n = (aX_n^2 + bX_n + c) \bmod m.$$

Теория утверждает, что получить псевдослучайную последовательность из квадратичного метода не более сложно, чем из линейного. Еще одна модификация принадлежит Р. Р. Ковзю. Она выражается следующими формулами:

$$X_0 \bmod 4 = 2; \quad X_{n+1} = X_n(X_n + 1) \bmod 2^m.$$

Достаточно легко увидеть, что эта модификация сильно связана с методом Неймана. Главная часть выражения справа – это почти квадрат величины, а операция деления на 2 в двоичной системе счисления – это не что иное, как смещение числа. Можно получить и другие модификации, например построив зависимость не от одного, предыдущего члена последовательности, а от их группы. Джон Мочли предложил использовать следующую зависимость:

$$X_{n+1} = \text{СерединаЧисла}(X_n * X_{n-k}).$$

Опять-таки не вдаваясь в теоретические исследования, заметим, что подобный

подход может дать серьезное увеличение периода. Рекуррентные последовательности, чье построение выполняется на группе уже посчитанных чисел, известны давно. Самая простая и, наверное, самая знаменитая последовательность такого вида – это числа Фиббоначи:

$$F_n = F_{n-1} + F_{n-2}.$$

Собственно ряд Фиббоначи начинается двумя единицами $F_1 = 1$ и $F_2 = 1$. Но в качестве обобщения можно принять эти значения любыми. Максимально широкое обобщение ряда Фиббоначи – это возвратная последовательность. То есть последовательность вида:

$$F_n = f(F_{n-1}, F_{n-2}, \dots, F_1).$$

Задание для самостоятельной работы

Постройте рекуррентную, псевдослучайную последовательность на числах Фиббоначи.

Собственно линейный конгруэнтный метод и квадратичный – это варианты возвратной последовательности. Еще один вариант возвратной последовательности, дающей псевдослучайные числа, может быть выражен следующей формулой:

$$X_{n+1} = (a_1 X_n + a_2 X_{n-1} + \dots + a_k X_{n-k}) \bmod m.$$

Данная формула предполагает первые k чисел последовательности известными. Такая последовательность может иметь очень большой период, но, к сожалению, для обоснования этой возможности мы опять сталкиваемся с необходимостью изучения достаточно серьезной теории. Единственное – можно дать совет: при построении линейной возвратной последовательности такого вида в качестве модуля брать большое простое число.

Методы перемешивания

Понятие «случайная последовательность» само не имеет четкого, предельно понятного определения. Теория вероятностей, которая есть единственное теоретическое основание для изучаемых методов, довольно упорно и последовательно уклоняется от необходимости определить свой предмет и свои основные понятия. Вполне возможно, что случайное число кажется таковым только в силу субъективного восприятия, в действительности не существуя. Альберт Эйнштейн, например, был уверен, что «Бог не играет в кости». Может быть, и так. Но тогда что для одного случайность, то для другого закономерность. Простой пример:

$$1, 4, 6, 7, 1, 3, 8, 4, 6, 9, 7, 2.$$

Последовательность не выглядит как закономерная, но разобьем её на две группы по шесть чисел и заметим, что:

$$1 + 4 + 6 = 7 + 1 + 3 \text{ и } 8 + 4 + 6 = 9 + 7 + 2.$$

Мы легко увидели закономерность, и совершенно не экзотическую. Эту последовательность можно продолжить, и более того, её вполне можно продолжить таким образом, что она будет удовлетворять некоторым критериям случайности. Для такой последовательности необходимо было бы ввести новый критерий случайности, запрещающий именно такую закономерность. Иначе говоря, можно придумать последовательность, которая будет случайна по всем критериям, кроме какого-то специального, искусственного, именно для неё придуманного, как в примере выше.

Такого рода последовательности можно придумывать до бесконечности, и даже более того, интуитивно ясно, что для любой последовательности можно придумать закономерность, которая именно в этой последовательности проявится. Еще один занятный пример:

1865, 2865, 1980, 8995, 3458, 1908 и т. д.

Заметим, что в каждом числе ровно один раз присутствует цифра 8. Можно ли считать это закономерностью? А почему бы и нет. Если мы это заметили, то сразу большое количество чисел полностью потеряли шанс появиться в ряду, что, впрочем, не означает, что ряд не сможет пройти статистических тестов.

Можно, конечно, возразить, что это искусственные закономерности и вероятность того, что такие закономерности окажутся значимыми, в реальном процессе равна нулю, но проблема в том, что событий с нулевой вероятностью не существует! Каждое хоть сколько-нибудь осмысленное событие имеет вероятность, отличную от нуля. Если мы миллион раз бросим монетку, то выпадение миллиона решек кажется делом совершенно не возможным, но миллион решек настолько же вероятен, как и та последовательность, которая выпадет, если вы не поленитесь бросить монетку миллион раз. Так что любая придуманная последовательность может иметь место, как и любая другая, а называть закономерность искусственной только на том основании, что именно сейчас мы её придумали, нельзя, интуитивные истины очень обманчивы.

Это небольшое философское отступление имело смысл показать, что все не так просто, как хотелось бы. Теория вероятностей уходит от проблемы традиционным математическим трюком. Она формулирует некоторое количество аксиом и выводит из них теоремы, которые мы вынуждены признать истинными, если уже согласились с аксиомами. Полезная же интерпретация понятия случайности, случайного числа, случайной последовательности – это дело конкретной практики.

Поэтому, как бы ни был хорош метод получения псевдослучайных чисел, приставка «псевдо» всегда будет нас смущать, и мы всегда должны подумать, а что еще можно сделать для большей «псевдослучайности». Сделать действительно кое-что можно.

Предположим, мы получили две псевдослучайные последовательности A и B . Если порядок элементов одной из них, например последовательности A , изменить с помощью последовательности B , то полученная последовательность C , очевидно, будет свободна если не от экзотических закономерностей, присутствующих в A и B , то, во всяком случае, от имеющихся в них периодов. Можно даже справедливо

предположить, что качество последовательности C может оказаться вполне приемлемым и при совершенно закономерных A и B . Рассмотрим, как можно построить такую гибридную последовательность.

Небольшой поучительный пример. Пусть последовательность A – вполне закономерная, например последовательность чисел Фибоначчи. Последовательность B возьмем еще более закономерную, пусть это будет ряд натуральных чисел. Если длины двух последовательностей одинаковы, то числа ряда B можно считать нумерацией чисел ряда A . Исходные данные – в следующей таблице:

Таблица 3.9

0	1	2	3	4	5	6	7	8	9
1	1	2	3	5	8	13	21	34	55

Выполним преобразование нумерации следующим образом: номер $a_k = (a_k * a_{k+1}) \bmod 10$, числа нижнего ряда расставляются соответственно новой нумерации:

Таблица 3.10

0	2	6	2	0	0	2	6	2	9
1	2	13	2	1	1	2	13	2	55

Последнее число мы оставили без изменений, его не на что умножить. Видно, что получившаяся последовательность несколько хуже в некотором смысле, числа последовательности менее разнообразны, но главная проблема ряда Фибоначчи – строгое возрастание чисел – ушла.

Более интересный метод. Возможно, впервые такой подход предложили М. Д. Мак-Ларен и Дж. Марсалья в 1965 году. Пусть опять есть две псевдослучайные последовательности A и B . Мы будем последовательность A использовать для перемешивания последовательности B . Для этого потребуются две длинные последовательности A и B , и пусть все $B_k < m$. Генерируемая последовательность C выбирается фиксированной длины. Пусть длина C равна L . Заполним L элементов C числами ряда A . Затем будем полученную последовательность изменять по следующему алгоритму (ниже описан один шаг):

1. Взять следующие элементы A и B .
2. Вычислить $N = [B * L / m]$.
3. Выполнить присвоение $C[N] = A$.

Для демонстрации метода перемешиваем последовательность чисел Фибоначчи из предыдущего примера: очевидно $m = 9$; значение L примем равное 5. Исходный набор данных будет таков:

Таблица 3.11

0	1	2	3	4
1	1	2	3	5

Шаг 1: $A = 5; B = 8; N = [5*5/9] = 2$.

Таблица 3.12

0	1	2	3	4
1	1	8	3	5

Шаг 2: $A = 6; B = 13; N = [5*6/9] = 3$.

Таблица 3.13

0	1	2	3	4
1	1	8	13	5

Шаг 3: $A = 7; B = 21; N = [5*7/9] = 3$.

Таблица 3.14

0	1	2	3	4
1	1	8	21	5

Уже из трех шагов видно, что закономерные последовательности своим результатом имеют вполне случайный результат.

Задание для самостоятельной работы

Напишите программу, генерирующую две последовательности линейным конгруэнтным методом, и постройте на них третью методом перемешивания.

Закключение. Пожалуй, самая большая проблема главы – это знание теории чисел. Если появится желание изучить представленные алгоритмы и методы более глубоко или понять как следует их математические основы, то потребуется специальная литература. Возможно, наиболее доступная книга для человека без хорошей математической базы – это [9], затем если желание углубиться в теорию останется, то рекомендуем вам [10] и [11].

Наверное, можно напомнить, что цель изложенных алгоритмов и методов – познакомить вас с некоторым набором идей и показать возможные механизмы реализации. Поэтому наши программы носят учебный характер и не претендуют на идеальность реализации. И если у вас появится желание использовать какой-либо алгоритм в бою, то над его реализацией надо будет ещё поработать. Но, во всяком случае, есть надежда, что текст главы был для вас бесполезен.

Арифметика

Представление числа в позиционной системе счисления	90
Проблемы технической реализа- ции арифметики	93
Реализация арифметики на уровне алгоритмического языка	97
Некоторые другие алгоритмы ...	115
Двоичная арифметика	120

Четвертая глава посвящена алгоритмам выполнения арифметических операций. Арифметические алгоритмы вполне можно назвать числовыми и внести их в предыдущую главу. Но алгоритмы арифметики представляют собой достаточно компактную тему со своими проблемами и методами решения, поэтому разумно их все-таки изучать как нечто самостоятельное. Здесь же немного места оставлено под алгоритмы перевода чисел между позиционными системами счисления с разными основаниями, так как представление числа – это основа любой арифметики. Завершается глава несколькими числовыми алгоритмами, без которых вполне можно обходиться, но иногда такие алгоритмы бывают полезны, кроме того, в их основе лежат идеи, понимание которых полезно для развития математического аппарата программиста.

Представление числа в позиционной системе счисления

Заметим, что число, записанное на бумаге, например 235687, это сложившаяся в течении многих веков договоренность о форме записи. Эта запись есть сокращенная форма следующего выражения:

$$7 + 8 \cdot 10^1 + 6 \cdot 10^2 + 5 \cdot 10^3 + 3 \cdot 10^4 + 2 \cdot 10^5.$$

Здесь заметна особая роль числа 10. Оно называется основанием позиционной системы счисления. Для позиционной системы характерно то, что вклад цифры в общую величину определяется её положением в записи числа. Чем цифра старше, тем её вклад больше. При этом даже меньшая цифра, стоящая в старшей позиции, дает больший вклад. Системы могут быть и непозиционными. Для непозиционных систем каждый знак имеет вполне определенный вес, независимый от расположения. Пример непозиционной системы счисления – римская система: X – десять, V – пять, I – единица и т. д. В этой системе число 30, например, запишется так: XXX. Возможности непозиционной системы очень ограничены. Если в римской системе появится необходимость записи большого числа, то для этого есть только две возможности: либо увеличивать длину записи, что очень скоро станет крайне неудобным, либо придумывать новые цифры, что скоро станет не менее неудобно. Поэтому ограничимся исследованием только позиционных систем. Общая формула представления числа:

$$A = a_0 D^0 + a_1 D^1 + \dots + a_n D^n.$$

Величина D называется основанием системы счисления. В качестве основания можно взять любое целое положительное число. Числа a_i называются цифрами. Из них составляется запись числа:

$$A = a_n a_{n-1} \dots a_0.$$

Цифры позиционной записи – неотрицательные целые числа: $a_i = 0, 1, \dots, D-1$. В качестве цифр принято использовать арабские знаки 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 и, если их не хватает, латинские буквы, как, например, в шестнадцатеричной системе

счисления. Выбор знаков для цифр – в принципе, тоже результат некоторой договоренности. Как записываются числа своими цифрами, было указано чуть выше, сейчас заметим только, что запись $a_n a_{n-1} \dots a_0$ содержит исчерпывающую информацию о цифрах и ничего не говорит о значении основания. Таким образом, запись 123456 может означать запись числа в системах с основанием от 7 и далее, и одна и та же запись может означать совершенно различные величины. Для ликвидации неопределенности принято в записи числа указывать основание следующим образом:

$$(a_n a_{n-1} \dots a_0)_D.$$

Единственное исключение сделано для наиболее распространенной десятичной системы. Если основание не указано, то предполагается, что $D=10$. Иногда значение основания ясно из текста, в котором используется числовая запись, в этом случае значение основания также не указывается, во всех иных случаях это надо делать во избежание неопределенности значений. Вот такая краткая информация об устройстве позиционного представления числа, далее займемся вопросами перевода.

Если число дано в некоторой системе счисления, то, очевидно, его можно переписать в любой другой системе по другому основанию. Рассмотрим два способа, как это сделать.

Простой подбор

Для вводного примера покажем перевод десятичного числа в двоичное представление. Пусть дано десятичное число 3456. Двоичное число – это запись в виде:

$$a_0 2^0 + a_1 2^1 + \dots a_n 2^n,$$

то есть мы должны представить число 3456 в виде степеней двойки, умноженных на коэффициенты $a_k = \{0, 1\}$. Первым шагом наберем достаточное количество таких степеней:

Таблица 4.1

$2^0 = 1$	$2^1 = 2$	$2^2 = 4$	$2^3 = 8$	$2^4 = 16$	$2^5 = 32$	$2^6 = 64$
$2^7 = 128$	$2^8 = 256$	$2^9 = 512$	$2^{10} = 1024$	$2^{11} = 2048$		

Заметим, что $2^{12} = 4096 > 3456$, то есть 12-ая степень двойки для представления числа 3456, очевидно, не пригодится. Это была подготовительная работа, сейчас начнем подбор нужных степеней:

Шаг 1. $3456 - 2048 = 1408 > 0$, следовательно, старший знак 1.

Шаг 2. $1408 - 1024 = 384 > 0$, знак разряда 1.

Шаг 3. $384 < 512$, знак разряда 0.

Шаг 4. $384 - 256 = 128 > 0$, знак разряда 1.

Шаг 5. $128 - 128 = 0$, знак разряда 1.

Число «закончилось», следовательно, получено следующее разложение по степеням двойки:

$$0*2^0 + 0*2^1 + 0*2^2 + 0*2^3 + 0*2^4 + 0*2^5 + 0*2^6 + 1*2^7 + 1*2^8 + 0*2^9 + 1*2^{10} + 1*2^{11}.$$

Перепишем двоичные цифры в обратном порядке и получим двоичное число:

$$110110000000.$$

Теперь пусть 3456 есть представление некоторого числа в семеричной системе, то есть $(3456)_7$ получим подбором его же запись и в пятиричной. Для этого мы должны записать степени пятерки по правилам семеричной системы. Имеем следующие степени числа 5:

Таблица 4.2

$5^0 = 1_7$	$5^1 = 5_7$	$5^2 = 34_7$	$5^3 = 236_7$	$5^4 = 1552_7$	$5^5 = 12053_7$
-------------	-------------	--------------	---------------	----------------	-----------------

Заметим, что последняя степень превышает исходное число, и, следовательно, оно при построении разложения учитываться не будет. Возможно, для тех, кто никогда не выполнял арифметические операции за пределами десятичной системы, не вполне понятно, как получается результат. Покажем на двух примерах механизм. Здесь все дело в правилах учета переполнения и переноса. Например, 5^2 в привычной десятичной арифметике – это 25 (две цифры). Почему? Потому что 5^2 – это две десятки и еще 5 единиц. В семеричной 5^2 – это три семерки и еще 4 единицы. Так как таблицы умножения для иных систем счисления никто из нас не учил, то можно цифры умножать десятично, а затем учитывать переполнение и перенос.

Выполним для закрепления еще одну операцию. $5^3 = 34_7 * 5$. Первая операция $4*5 = 20$ (десятичное умножение). Это число состоит из двух 7 и 6 единиц. Это то, что называется 6 пишем, 2 на ум пошло. Вторая операция: $3*5 + 2 = 17$. Здесь 2 семерки и 3 единицы. То есть 3 пишем, 2 на ум пошло. И окончательный результат 236_7 . Начинаем выполнять вычитания. Все операции по основанию 7.

Шаг 1. $3456 - 1552 = 1604 - 1552 = 22$. Число 5^4 входит в разложение дважды.

Шаг 2. $22 - 5 = 14 - 5 = 6 - 5 = 1$. Число 5^1 входит в разложение трижды.

Шаг 3. $1 - 1 = 0$. Число 5^0 входит в разложение один раз.

Таким образом, получаем: $3456_7 = 20031_5$. Выполним проверку, переведя оба числа в десятичное представление:

$$\begin{aligned} 3456_7 &= 3*7^3 + 4*7^2 + 5*7 + 6 = 1266; \\ 20031_5 &= 2*5^4 + 0*5^3 + 0*5^2 + 3*5 + 1 = 1266. \end{aligned}$$

Что, собственно, и ожидалось.

Задание для самостоятельной разработки

Напишите программную реализацию метода подбора из системы счисления с основанием D_1 в систему счисления с основанием D_2 .

Алгоритм подбора выглядит очень естественным. И наверное, его понимание

не должно составить проблемы. Но он обладает явными недостатками, от которых можно и нужно избавиться. Во-первых, он требует выполнения некоторой подготовительной работы. Во-вторых, ему нужен массив, который для перевода больших чисел может оказаться весьма велик. Поэтому реализуем несколько иную идею. Сначала для перевода из десятичной системы счисления в любую другую.

Для построения алгоритма перевода достаточно заметить, что все цифры являются остатками от деления числа A , представленного в десятичной системе счисления, на основание D . Таким образом, реализация такого перевода – это цикл деления переводимого числа на основание до тех пор, пока частное от деления не станет равно нулю. Получившаяся при этом последовательность остатков и есть последовательность цифр записи числа в новой системе счисления.

Листинг 4.1

```
PROCEDURE Расчет(Число, Основание:INTEGER);  
VAR  
  цифра:INTEGER;  
BEGIN  
  WHILE Число#0 DO  
    цифра:=Число MOD Основание;  
    StdLog.Int(цифра);  
    Число:= Число DIV Основание;  
  END;  
END Расчет;
```

Задание для самостоятельной разработки

Усложним задачу. Пусть дано число по основанию D_1 , получить его представление по основанию D_2 . Обратите внимание, что произвольность оснований не позволит непосредственно использовать операции DIV и MOD. Поэтому главная проблема задачи – разработка собственных операций целочисленного деления с остатком.

Проблемы технической реализации арифметики

Процесс вычисления сложных арифметических выражений можно описать на языке программирования. При этом сложная арифметика сведется к некоторому набору простых арифметических операций. Таких операций известно 4. Но и они сводимы друг к другу. Например, вычитание можно представить как сложение положительного числа с отрицательным. Умножение – как многократное сложение. Деление – как умножение делимого на число, обратное делителю. Но какую-то из операций придется принять в качестве базовой и обеспечить её выполнение уже не средствами программирования, а средствами аппаратуры. Наверное, самый естественный выбор такой операции – это операция сложения.

Итак, надо не просто сложить два числа, а сложить их, используя вполне определенные технические устройства, обладающие фиксированными особенностями и возможностями. Аппаратная реализация арифметики имеет свои сложности. Для машины безразличен выбор системы счисления, есть проблема с ограни-

чением длины числа и обработкой переполнения, если длина числа в реальных расчетах превысит допустимую величину. Не вполне тривиальна проблема учета знака числа. Определенные сложности вносит необходимость обеспечения точности расчетов, так как большинство рациональных чисел для своего представления требуют большого количества знаков после запятой, а длина машинного числа ограничена. Очень важны вопросы оптимизации по скорости выполнения операций. Арифметические операции – наиболее затратная часть любой программы, поэтому далеко не безразлично, сколько процессорных тактов уходит, например, на сложение. Все это очень емкие вопросы попробуем рассмотреть некоторые из них, в первом приближении.

Двоичный сумматор

На сегодняшнем техническом уровне проблема аппаратного представления числа решается устройством, называемым триггером. Не вдаваясь в схемотехнические подробности, отметим, что триггер может находиться только в двух состояниях: он может быть включенным или выключенным. Включенный (пропускающий электрический сигнал) триггер обозначает единицу, выключенный (не пропускающий) обозначает нуль. Последовательность триггеров есть аппаратная реализация представления числа. Ограниченность состояний триггеров приводит к ограничению в выборе системы счисления. Естественным образом приходим к выбору двоичной системы, состоящей из двух цифр: 0 (триггер выключен), 1 (триггер включен).

Простое аппаратное представление чисел дает также и преимущество в оценке погрешности. Более детальный анализ двоичной арифметики дан в конце этой главы, здесь же только отметим, что необходимость учета погрешности может возникать при переводе числа из одной системы счисления в другую. Одна и та же величина, будучи в одной системе, представлена конечной дробью, в другой системе может оказаться бесконечной периодической дробью. Например, $(0,1)_{10}$ в двоичной системе превращается в бесконечную дробь: $0,000110001100011\dots$. Но это не создает серьезных неприятностей. При таком переводе погрешность двоичного представления не превышает единицы младшего разряда.

Вернемся к двоичному сумматору. Это устройство решает задачу сложения двух чисел одинаковой длины. Старшие незначащие разряды при этом заполняются нулями. При сложении двух разрядов может возникнуть ситуация переполнения. В этом случае «лишняя» единица должна быть учтена при суммировании старшего разряда. Таким образом, в операции сложения участвуют три операнда и получаются два результата:

- операнд a_k – очередной разряд слагаемого a ;
- операнд b_k – очередной разряд слагаемого b ;
- операнд p_k – регистр для хранения единицы переноса от операции сложения над младшим разрядом (разрядом с номером $k - 1$);
- результат c_k – очередной разряд числа – суммы;
- результат p_{k+1} – регистр для хранения единицы переноса от операции сложения над текущим разрядом.

Аппаратно p_k и p_{k+1} могут совпадать, но мы их для удобства представления разделим. Работу двоичного сумматора над текущим разрядом можно описать следующей таблицей:

Таблица 4.3

a_k	b_k	p_k	c_k	p_{k+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Для выполнения операции вычитания можно построить аппаратный вычитатель. Технически в этом нет ничего сложного, но все же сумматор реализуется чуть более просто, а самое главное – иметь два разных устройства для похожих операций не разумно, так как в расчете сложного выражения, содержащего и сложения, и вычитания, придется тратить довольно существенные усилия для координации деятельности сумматора и вычитателя. В действительности проблема вычитания вынесена в разряд задач программирования, а не аппаратной реализации. Операция вычитания может быть представлена как сложение отрицательного числа с положительным. Единственное, при таком подходе необходимо решить проблему представления отрицательного числа. Для этого к двоичному представлению числа добавляется еще один бит, называемый знаковым. В отношении этого бита есть договоренность, что ноль в бите означает положительное число, а единица – отрицательное.

Ускорение операции сложения

Итак, вся машинная арифметика сводится к одной-единственной операции сложения двух двоичных чисел. Это означает, что борьба за такты процессора также сводится к ускорению этой единственно интересной арифметической операции. Мы уже знаем, что операция сложения – это два действия: поразрядное сложение и переносы. Существуют разные методы ускорения операции сложения, но, наверное, все они так или иначе используют идею раздельной обработки поразрядного сложения и переносов. Поясним идею на примере:

$$0100011100 + 0101110111 = 1010010011.$$

Таблица 4.4

0	1	0	0	0	1	1	1	0	0
0	1	0	1	1	1	0	1	1	1
	0	0	1	1	0	1	0	1	1
1	0	0	0	1	0	1	0	0	0
1	0	0	1	0	0	0	0	1	1
0	0	0	1	0	1	0	0	0	0
1	0	0	0	0	1	0	0	1	1
0	0	1	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	1	1
0	0	0	0	0	0	0	0	0	0

Здесь в качестве промежуточных значений используются два числа, промежуточная сумма и число переносов. Процесс завершается тогда, когда число переносов полностью обнуляется. На первом шаге суммируются исходные числа, на каждом последующем – промежуточная сумма и число переносов предыдущего шага.

Промежуточная сумма формируется согласно следующему правилу: *k -ый разряд суммы равен 0, если равна 0 или 2 сумма k -ых разрядов слагаемых, k -ый разряд суммы равен 1, если равна 1 сумма k -ых разрядов слагаемых.*

Число переносов формируется согласно следующему правилу: *самый младший разряд числа всегда равен 0. k -ый разряд равен 1, если сумма $(k-1)$ -ых разрядов равна 2. В иных случаях разряд числа переносов равен 0.*

Представление чисел в форме с фиксированной и плавающей десятичной точкой

Все затронутые выше проблемы обсуждались на примерах целых чисел, которыми, как известно, числовое множество не ограничивается. Если говорить о числах, содержащих дробную часть, а совсем оставить их без внимания нельзя, то необходимо обсудить формы их представления. Таковых существует две: представление числа с фиксированной точкой и представление числа с плавающей десятичной точкой.

Для представления с фиксированной точкой оговариваются две вещи: общая длина числа и длина его дробной или, наоборот, целой части. Тогда информацию о десятичной точке можно опустить и все разряды использовать под информацию о числе (значение и знак). С машинной точки зрения, число с фиксированной точкой представляет собой более простой объект для построения алгоритмов обработки, но такое представление и больше подвержено потерям точности. Например, становится возможным появление машинного нуля – числа, отличающегося

от нуля, но настолько малого, что его младшие разряды не помещаются в машинное слово.

В представлении с плавающей точкой машинное слово, содержащее, число делится на две части: мантиссу и порядок. Например, число 0.001 может быть записано как $0.1 \cdot 10^3$ в этом случае, и для очень малого числа можно избежать появления машинного нуля и существенных потерь точности. Однако разбиение числа на две неравнозначные по смыслу части предъявляет и большие запросы к алгоритмам обработки. Такие алгоритмы более требовательны к ресурсам аппаратуры, и надо полагать, что доминирующее положение плавающей точки было обеспечено бурным ростом быстродействия процессоров.

На этом мы завершим обзор проблем машинного представления чисел и арифметических операций. Желющие ознакомиться более детально могут обратиться, например, к [12].

Реализация арифметики на уровне алгоритмического языка

Следующие несколько параграфов посвящены собственно четырем арифметическим операциям: сложению, вычитанию, умножению и делению. Речь пойдет об операциях над длинными числами, поэтому договоримся о структуре данных, представляющих такие числа:

TYPE

Число=RECORD

цифра:ARRAY 1000 OF INTEGER;

длина:INTEGER;

END;

В реальных задачах, возможно, более целесообразно заменить статические массивы на связные списки. Возможно, для некоторых целей более выгодно работать с двоичным представлением числа, так как операции на двоичных числах выполняются несколько проще. Собственно, прежде чем что-то излагать, нам придется ограничить собственную задачу. Договоримся о двух простых вещах. Во-первых, все арифметические операции будем рассматривать только в десятичной системе счисления, и, во-вторых, ограничимся только целыми числами.

Это не слишком большие ограничения. Переход от одной системы счисления к другой требует только лишь переопределения правил переноса при переполнении разряда и переопределения таблицы умножения. А некоторые проблемы, возникающие в арифметике чисел с дробной частью, обсудим отдельно.

Сложение двух чисел

Операция сложения не требует большого количества операций, она настолько проста, что, наверное, общеизвестный способ сложения столбиком можно считать вполне эффективным. Единственное – следует помнить, что алгоритм и его реали-

зация это не вполне одно и то же. Поэтому сложение столбиком можно написать самыми различными способами. Рассматривайте нашу реализацию, как одну из возможных.

Листинг 4.2

```

PROCEDURE Суммирование(OUT Сумма:Число;IN Слагаемое1,Слагаемое2:Число);
VAR
    max,i,sum:INTEGER;
BEGIN
    (*Выбор числа, содержащего большее количество знаков*)
    IF Слагаемое1.длина>Слагаемое2.длина THEN
        max:=Слагаемое1.длина;
    ELSE
        max:=Слагаемое2.длина;
    END;
    i:=0;
    sum:=0;
    WHILE i<=max DO
        IF i<=Слагаемое1.длина THEN
            (*Суммирование очередного разряда первого числа*)
            sum:=sum+Слагаемое1.цифра[i];
        END;
        IF i<=Слагаемое2.длина THEN
            (*Суммирование очередного разряда второго числа*)
            sum:=sum+Слагаемое2.цифра[i];
        END;
        IF sum<10 THEN
            Сумма.цифра[i]:=sum;
            sum:=0;
        ELSE (*Расчет с переносом*)
            Сумма.цифра[i]:=sum-10;
            sum:=1;
        END;
        i:=i+1;
    END;
    IF sum>0 THEN (*Обработка переполнения*)
        Сумма.длина:=max+1;
        Сумма.цифра[max+1]:=sum;
    ELSE
        Сумма.длина:=max;
    END;
END Суммирование;

```

Замечание о переносе. После сложения очередных разрядов со значением переноса как максимум может получиться число 19. Следовательно, максимальное

значение переноса в десятичной системе счисления равно 1, а значение разряда результата `sum-10`.

Замечание о выборе длинного числа. Предложенный алгоритм предполагает известную длину большего слагаемого, что необходимо для определения количества операций суммирования. От этого действия можно отказаться в двух случаях:

- если доподлинно известно, что незначащие старшие разряды заполнены нулями. Гарантировать этот факт можно только специальными усилиями, требующими некоторого счетного времени;
- в алгоритме при выполнении каждого поразрядного сложения проверяется, не превышает ли номер разряда действительную длину числа. Общий цикл суммирования можно прекратить, если номера обоих суммируемых разрядов превысили длины своих слагаемых. Для этого можно определить логический флаг, который примет значение Ложь в случае неуспешности обеих операций суммирования. Но в этом случае придется выполнять некоторое количество дополнительных присвоений.

Задание для самостоятельной работы

Выше была изложена идея ускорения суммирования двоичных чисел. Реализуйте сложение десятичных чисел с использованием механизма ускорения.

Задание для самостоятельной работы

Пусть теперь число задано в произвольной системе счисления с основанием D . Напишите реализацию операции сложения.

Вычитание из большего меньшего

Приведенная здесь процедура также опирается на алгоритм вычитания столбиком. Еще одно ограничение – процедура вычитает из большего числа меньшее. Устранение этого ограничения можете считать заданием для самостоятельной работы.

Идея алгоритма. Вычитание выполняется поразрядно. Разряд вычитаемого отнимается от разряда уменьшаемого, если это возможно, а если нет, то из ближайшего старшего ненулевого занимает единица. Чтобы не иметь дела с тремя числами (уменьшаемым, вычитаемым и разностью), перед началом процесса положим, что разность уже есть уменьшаемое. Можно этого не делать и выполнять вычитание из уменьшаемого, но в этом случае значение уменьшаемого будет потеряно, а оно, возможно, необходимо для других арифметических операций.

Итак, первый оператор **Разность:=Уменьшаемое;** превращает уменьшаемое в разность, но так как это еще не настоящая разность, то начинается цикл поразрядного вычитания из разности разрядов вычитаемого.

```
i:=0;
```

```
WHILE i<=Вычитаемое.длина DO
```

```
  (*Собственно операция вычитания*)
```

```
i:=i+1;  
END;
```

При вычитании из разряда возможна простая ситуация: разряд уменьшаемого больше разряда вычитаемого. Тогда так:

```
IF Вычитаемое.цифра[i]<=Разность.цифра[i] THEN  
    Разность.цифра[i]:=Разность.цифра[i]-Вычитаемое.цифра[i];  
ELSE (*Вторая ситуация*)  
END;
```

Вторая ситуация – это разряд уменьшаемого меньше разряда вычитаемого. В этом случае необходимо найти разряд разности, в котором можно занять единицу. Такой разряд, в предположении, что уменьшаемое не меньше вычитаемого, всегда есть, но он не обязательно следующий. Поэтому придётся организовать цикл для поиска:

```
k:=i+1;  
WHILE Разность.цифра[k]=0 DO  
    k:=k+1;  
END;
```

После того как необходимый разряд найден, в нем можно занять единицу:

```
Разность.цифра[k]:=Разность.цифра[k]-1;
```

Но это не все. Вспомним, что в поиске разряда для заимствования, возможно, было пройдено некоторое количество нулевых разрядов. Их по правилам заимствования требуется заполнить девятками:

```
Разность.цифра[k]:=Разность.цифра[k]-1;  
j:=i+1;  
WHILE j<k DO  
    Разность.цифра[j]:=9;  
    j:=j+1;  
END;
```

И только после этого окончательно вычисляем разряд, из-за которого все эти расчеты были выполнены:

```
Разность.цифра[i]:=Разность.цифра[i]+10-Вычитаемое.цифра[i];
```

В этом выражении число 10 представляет собой сухой остаток от занятой где-то в старших разрядах единицы. Последняя проблема – это расчет длины числа. Согласно нашей договоренности о структуре данных, длина числа хранится в специальной переменной, и её значение надо рассчитать. Мы это делаем достаточно примитивно. Максимально возможная длина разности равна длине уменьшаемого. Отбросим от этого значения все нулевые старшие разряды и получим искомую длину. Правда, такой прием требует, чтобы все старшие незначащие разряды были

нулями. Ниже – полный листинг процедуры.

Листинг 4.3

```
PROCEDURE Вычитание(OUT Разность:Число;IN Уменьшаемое,Вычитаемое:Число);
VAR
  i,k,j:INTEGER;
BEGIN
  Разность:=Уменьшаемое;
  i:=0;
  WHILE i<=Вычитаемое.длина DO
    IF Вычитаемое.цифра[i]<=Разность.цифра[i] THEN
      Разность.цифра[i]:=Разность.цифра[i]-Вычитаемое.цифра[i];
    ELSE
      (*Поиск ненулевого разряда*)
      k:=i+1;
      WHILE Разность.цифра[k]=0 DO
        k:=k+1;
      END;
      Разность.цифра[k]:=Разность.цифра[k]-1;
      j:=i+1;
      WHILE j<k DO
        Разность.цифра[j]:=9;
        j:=j+1;
      END;
      Разность.цифра[i]:= Разность.цифра[i]+10-Вычитаемое.цифра[i];
    END;
    i:=i+1;
  END;
  (*Уточнение длины числа - разности*)
  WHILE Разность.цифра[Разность.длина]=0 DO
    Разность.длина:=Разность.длина - 1;
  END;
END Вычитание;
```

Задание для самостоятельной работы

Известно, что вычитание можно свести к сложению уменьшаемого с отрицательным вычитаемым. В условиях нашего представления числа представить его отрицательным можно, заменив все разряды на отрицательные, после чего воспользоваться уже готовой процедурой сложения. Проанализируйте идею на предмет её корректности, нужно ли процедуру суммирования как-то модифицировать, и если да, то как.

Задание для самостоятельной работы

В нашей реализации недостающая единица занимается из первого ненулевого,

старшего разряда. Но так как разряды числа представлены типом INTEGER, то недостающую единицу можно занять уже из следующего разряда, даже если он нулевой. Разряд, из которого произошел заем, при этом станет отрицательным, что в общем-то не мешает выполнению операций. Мы можем утверждать, что если уменьшаемое не меньше вычитаемого, то к концу операции вычитания отрицательных разрядов не останется. Проанализируйте идею на предмет её корректности и при положительном решении напишите реализацию.

Задание для самостоятельной работы

Возможно, к этому моменту у вас есть три реализации операции вычитания. Проведите эксперимент и выясните, какая из них наиболее эффективна по скорости.

Умножение

Экзотический способ умножения. В конце главы дано несколько арифметических алгоритмов. В их числе – алгоритм быстрого возведения в степень. Различие между возведением в степень и умножением таково, что позволяет алгоритм быстрой степени легко модифицировать в алгоритм умножения. Идея, возможно, интересная, но все же более экзотическая, чем интересная. В действительности такой алгоритм проиграет обычному умножению столбиком, которым мы займемся ниже.

Операция умножения существенно отличается по количеству выполняемых операций от сложения и вычитания. Настолько существенно, что есть смысл подумать о более быстрых алгоритмах, нежели умножение столбиком. Но начнем анализ проблемы мы все же со столбика. Умножение двух чисел можно рассматривать как умножение двух многочленов вида:

$$\begin{aligned} A &= a_0D^0 + a_1D^1 + \dots a_nD^n \\ B &= b_0D^0 + b_1D^1 + \dots b_mD^m \end{aligned}$$

После перемножения многочленов необходимо привести подобные. После приведения подобных некоторые из коэффициентов многочлена результата могут оказаться больше либо равны основанию системы счисления (переполнение разряда). Справившись с проблемой переполнения, получим многочлен, представляющий число-результат.

Как определить подобные. Многочлен-результат представляет собой сумму, каждое из слагаемых которой есть произведение вида $a_kD^kb_jD^j = a_kb_j(D^{k+j})$, где $k = 0 \dots n$ и $j = 0 \dots m$. Подобными будут все слагаемые, имеющие одинаковую сумму индексов.

Как справиться с переполнением. Обозначим сумму индексов через S . Тогда числовой коэффициент при D^S равен сумме произведений a_kb_j , таких что $S = k + j$. Эта сумма может оказаться больше либо равна D (переполнение разряда). Предположим, это произошло, и некий L_s больше D , так что: $L_s = pD + q$ (результат целочисленного деления D на L_s). Это означает необходимость выполнить перерасчет коэффициентов следующим образом:

$$\begin{aligned} L_{s+1} &= L_{s+1} + p \\ L_s &= q \end{aligned}$$

Реализация. Каждый разряд числа-произведения рассчитывается как сумма произведений цифр множителей. Сумма считается с накоплением, изначально все разряды результата должны быть равны нулю, поэтому начнем с инициализации.

```
k:=0;  
WHILE k<=Множитель.длина+1 DO  
  Произв.цифра[k]:=0;  
  k:=k+1;  
END;
```

Расчет поразрядного произведения – это перемножение каждого разряда с каждым, что возможно реализовать следующей конструкцией:

```
k:=0;  
(*Перебор всех разрядов первого множителя*)  
WHILE k<=Множ1.длина DO  
  i:=0;  
  (*Перебор всех разрядов второго множителя*)  
  WHILE i<=Множ2.длина DO  
    i:=i+1;  
  END;  
  k:=k+1;  
END;
```

Вложенный цикл, собственно, и реализует умножение. Умножается разряд с индексом i на разряд с индексом k . Результат умножения суммируется с разрядом числа результата с номером $i + k$.

Произв.цифра[i+k]:=Произв.цифра[i+k]+Множ1.цифра[i]*Множ2.цифра[k];

Естественно, в результате этой операции может возникнуть ситуация переполнения. Собственно разряду с номером $i + k$ принадлежит только остаток от деления на основание системы счисления. Переполнение – это целая часть от деления:

Произв.цифра[i+k+1]:=Произв.цифра[i+k+1]+(Произв.цифра[i+k] DIV 10);
Произв.цифра[i+k]:=Произв.цифра[i+k] MOD 10;

И наконец, полный листинг. В конце листинга процедуры есть еще небольшой фрагмент, вычисляющий длину числа-произведения, как он работает, проведите самостоятельный анализ.

Листинг 4.4

```
PROCEDURE Умножение(OUT Произв:Число;IN Множ1, Множ2:Число);  
VAR  
  k,i,c:INTEGER;  
BEGIN  
  (*Инициализация числа - результата*)  
  k:=0;
```

```

WHILE k<=Множ1.длина+Множ1.длина+1 DO
  Произв.цифра[k]:=0;
  k:=k+1;
END;
(*Расчет произведения*)
k:=0;
WHILE k<=Множ2.длина DO
  i:=0;
  WHILE i<=Множ1.длина DO
    Произв.цифра[i+k]:=Произв.цифра[i+k]+Множ1.цифра[i]*Множ2.цифра[k];
    Произв.цифра[i+k+1]:=Произв.цифра[i+k+1]+(Произв.цифра[i+k] DIV 10);
    Произв.цифра[i+k]:= Произв.цифра[i+k] MOD 10;
    i:=i+1;
  END;
  k:=k+1;
END;
(*Определение длины произведения*)
IF Произв.цифра[Множ1.длина+Множ2.длина+1]#0 THEN
  Произв.длина:= Множ1.длина+Множ2.длина+1;
ELSE
  Произв.длина:= Множ1.длина+Множ2.длина;
END;
END Умножение;

```

Задание для самостоятельной работы

Напишите реализацию алгоритма умножения столбиком в системе счисления с основанием D .

Еще один метод умножения

Метод умножения, который мы разберем ниже, пожалуй, не имеет самостоятельной ценности. Но он может быть полезен для длинной арифметики при условии, что есть эффективные алгоритмы умножения для основных числовых типов. Простые алгоритмы длинной арифметики, рассматриваемые в этой главе, подразумевают, что число представлено массивом цифр. Однако если элементы массива представлены целым типом или даже длинным целым, то тогда элемент массива может содержать сразу группу цифр. Например, число 1324 6785 можно представить двумя числами:

$$1324\ 6785 = 1324 \cdot 10^4 + 6785 \cdot 10^0$$

Соответственно, для его представления потребуются не 8 элементов массива, а только два. Рассмотрим умножение двух восьмизначных чисел: $A = A_1 10^4 + A_2$ и $B = B_1 10^4 + B_2$. Их произведение запишется так:

$$A*B = (A_1 + B_1)*10^8 + (A_1B_2 + A_2B_1)*10^4 + A_2B_2.$$

Тогда умножение двух восьмизначных чисел сведется к трем операциям умножения и четырем операциям сложения двух 4-разрядных чисел. Причем можно использовать операции сложения и умножения, определенные для основных языковых типов. Для лучшего понимания метода выполним умножение двух 16-разрядных чисел, сведя их к двум 8-разрядным.

Выполним умножение числа $M = 20243\ 97459\ 71664\ 32102$ на число $m = 69732\ 82428\ 43662\ 95023$. Разобьем каждый сомножитель на два числа следующим образом: $M_0 = 20243\ 97459$ и $M_1 = 71664\ 32102$; $m_0 = 69732\ 82428$ и $m_1 = 43662\ 95023$. Тогда:

$$M*m = M_0*m_0*10^{20} + (M_0*m_1 + M_1*m_0)*10^{10} + M_1*m_1.$$

Для расчета произведения выполним последовательно следующие операции:

1. Вычислим $M_1m_1 = 31290\ 75681\ 96300\ 28346$ и выпишем цифры 96300 28346, находящиеся в произведении с 1 по 10 разряды.
2. Перенесем цифры 31290 75681, записанные с 11 по 20 разряды, в разряды 1–10.
3. Вычислим $M_1m_0 + M_0m_1 + 31290\ 75681 = 58812\ 67160\ 12663\ 25894$ и выпишем цифры из разрядов 1–10.
4. Перенесем цифры 58812 67160 из разрядов 11–20 в разряды 1–10.
5. Вычислим $M_0m_0 + 58812\ 67160 = 14116\ 69523\ 40138\ 17612$.

Ниже показано, как получается результат:

Таблица 4.5

						96300	28346
				12663	25894		
14116	69523	40138	17612				
14116	69523	40138	17612	12663	25894	96300	28346

Задание для самостоятельной работы

Напишите реализацию изложенного выше метода.

Задание для самостоятельной работы

Здесь от вас потребуется небольшой теоретический анализ, но в принципе можете решить задачу, и проведя эксперимент. Задача же следующая. Понятно, что количество операций в умножении зависит от длины числа: чем число длиннее, тем операций больше. Переведя число в систему с большим основанием, мы получим число меньшей длины и, следовательно, потратим при умножении меньше операций. Вывод – выполнение операций над числами с большим основанием системы счисления дает вычислительную экономию. Верный ли это вывод? Ресурсы, затраченные на перевод числа, можно проигнорировать, так как число можно сразу задавать в нужной системе счисления.

Умножение. Метод Карацубы

Умножение столбиком можно назвать естественным алгоритмом, в том смысле, что столбиковое умножение непосредственно опирается на таблицу умножения цифр и определение операции умножения через сложение. Естественные идеи такого рода хороши для реализации, так как предполагают только некоторые технические сложности и не более того, но вряд ли так можно получить максимум эффективности. Кстати, есть основание полагать, что наша реализация дает возможный максимум для столбика, но если есть желание, вы можете попытаться написать свой более быстрый столбик.

Мы же займемся менее естественными идеями, дающими реальный эффект. Надо сказать, что достаточно долго математики полагали, что умножение столбиком и есть самый быстрый из возможных алгоритмов. Математическая в том уверенность была настолько велика, что 1956 году Колмогоров эту уверенность выразил в виде гипотезы, утверждающей, что любой возможный метод умножения будет иметь сложность порядка cmn , где m и n – количество разрядов множителей, а величина c – некоторая константа. Математиков понять можно, несколько тысяч лет умножения столбиком сыграли свою роль, но уже в 1960 году Анатолий Карацуба высказал достаточно простую идею, позволяющую умножать числа существенно быстрее. Его идея основана на следующем очевидном соотношении:

$$4ab = (a + b)^2 - (a - b)^2, \text{ откуда } ab = \frac{(a + b)^2 - (a - b)^2}{4}.$$

Сложностью операций сложения и вычитания можно пренебречь. Остается операция возведения в квадрат. Таким образом, проблема быстрого умножения становится проблемой быстрого возведения в квадрат. Поставим следующую задачу: дано число X , вычислить X^2 . Из данной немного выше оценки следует, что обычное умножение при возведении в квадрат дает количество операций, сопоставимое с квадратом длины числа, то есть количество операций растет намного быстрее длины. Отсюда идея – разбить исходное число на два меньших, это должно дать экономию. Идея действительно выигрышная. Число X можно разными способами представить в виде суммы двух: $X = X_1 + X_2$. Найдем такое представление, что длина X_1 равна длине X и половина младших разрядов X_1 равна нулю. Тогда длина X_2 равна половине длины X . Пример:

$$4578394\ 9002338 = 45783940000000 + 9002338 = 4578394 * 10^7 + 9002338.$$

Пример требует, чтобы число разрядов исходного числа было четным, это существенная деталь, которую мы обсудим позже, сейчас заметим главное – большое число можно представить в виде короткого числа и степени числа 10. Умножение на число 10^N сводится к сдвигу числа, и это даст возможность избавиться от некоторого количества поразрядных операций (столбиковых умножений и сложений). Запишем сказанное в общем виде. Если длина числа X есть $N = 2k$, то X представимо в виде:

$$X = X_1 + 10^k X_2,$$

где $X_1 = a_0 + 10a_1 + \dots + 10^{k-1}a_{k-1}$ и $X_2 = a_k + 10a_{k+1} + \dots + 10^{k-1}a_{k-1}$.

Тогда квадрат представим формулой

$$X^2 = (X_1 + 10^k X_2)^2 = X_1^2 + (X_1 + X_2)^2 - (X_1^2 - X_2^2) 10^k + X_2^2 10^N.$$

Вернемся к вопросу о четности/нечетности длины числа X , кстати, не путайте длину числа с номером старшего разряда. При нумерации разрядов с нуля, для четной длины, номер старшего разряда будет нечетным числом. Нечетность длины X означает, что сумма $X_1 + X_2$ имеет $k+1$ разряд. Тогда X представимо в виде суммы $2X_3 + X_4$, где X_3 – k -значное и X_4 – однозначное. В этом случае:

$$X^2 = (2X_3 + X_4)^2 = 4X_3^2 + 4X_3X_4 + X_4^2.$$

Таким образом, вычисление квадрата числа в зависимости от четности его длины сводится к двум простым формулам, оперирующим числами с вдвое меньшими длинами. В этом и заключается сущность метода Карацубы. Доказательство его превосходства над умножением столбиком мы опускаем.

Задание для самостоятельной работы

Напишите реализацию метода Карацубы в длинном целом типе.

Задание для самостоятельной работы

Напишите реализацию метода Карацубы, используя процедуры длинной арифметики, и сравните экспериментально скорость работы вашей реализации и приведенного в этой главе умножения столбиком. Для эксперимента рекомендуем использовать не менее чем 50-значные числа.

В заключение об умножении. Умножение – в некотором смысле ключевая операция длинной арифметики, да и вообще арифметики. Она, конечно, не так фундаментальна, как сложение, к которому в конечном итоге сводится вся арифметика, но сложение – достаточно простая вещь, мало что позволяющая в плане придумывания быстрых алгоритмов. Умножение таит в себе массу возможностей оптимизации. Мы рассмотрели только один быстрый алгоритм, но после алгоритма Карацубы многое изменилось, и быстрых способов умножения сейчас довольно много. Некоторые алгоритмы используют вполне школьную математику, хотя и несколько изощренную, есть алгоритмы, основанные на серьезном математическом знании, но это уже выходит за рамки нашего популярного изложения, заинтересовавшихся отправим к [2], а мы перейдем к следующей нетривиальной арифметической операции.

Деление

Пожалуй, наиболее тяжелая арифметическая операция и в плане реализации, и в плане требуемых вычислительных усилий. В случае действительной операции

деления проблему создает необходимость достижения определенной точности. В случае целочисленного деления необходимо определить не один, а два результата: частное и остаток.

Применение метода половинного деления

Пусть некоторое время нас интересует только частное. Частное можно представить как неизвестную величину в уравнении вида:

$$A - Bp = 0, \text{ где } A - \text{ делимое, } B - \text{ делитель, } p - \text{ частное.}$$

Как известно (см. гл. 2), такие уравнения решаемы методом половинного деления. Здесь $f(x) = A - Bp$, левая граница отрезка, содержащего корень, есть 0 и правая граница равна делимому A . Алгоритм расчета приведен во второй главе. Остаток находится еще проще. Если частное получено с точностью до целого, то остаток легко определяется из формулы

$$A = Bp + q \Leftrightarrow q = A - Bp.$$

Изложенный метод деления самостоятельной ценности не имеет, так как проблема быстрого деления сводится к проблеме быстрого умножения, которая не намного проще. Кроме того, эффективная реализация деления столбиком будет работать быстрее. Но если получить быстрый алгоритм умножения, то идея сведения деления к умножению может оказаться достаточно плодотворной, если вместо половинного деления использовать более быстрые алгоритмы, например метод Ньютона. А сейчас столбиковый алгоритм:

Реализация алгоритма деления столбиком. Деление столбиком сводится к вычитанию делителя из делимого до тех пор, пока это возможно, то что, останется по завершении всех вычитаний, является остатком. Процесс разобьем на этапы. На каждом этапе от делимого, вернее от того, что от него осталось, отсчитывается некоторое количество разрядов, достаточное для того, чтобы сформировать число, большее либо равное делителю. Это «число» не содержится в специальной структуре данных, оно обозначается в делимом двумя границами: R – старший разряд, L – младший. Вычитание делителя выполняется из обозначенного R и L отрезка разрядов. Процедура вычитания и процедура формирования очередного отрезка выполняются в теле цикла. Но так как первый отрезок формируется немного особым, он выделен из цикла. То есть перед началом большого цикла выполняется небольшая вспомогательная работа.

Заметим, что здесь, как в реализации вычитания, операции выполняются не над самим делимым. Это необходимо для того, чтобы число делимое оставить в сохранности для других арифметических операций, в которых оно, возможно, участвует. В нашей реализации деление – это вычитание, в конце которого получается остаток, поэтому целесообразно в начале процесса скопировать значение делимого в остаток.

Остаток:=Делимое; (будущий остаток в начале процесса равен делимому)
(*Расчет первого отрезка*)

```
Остаток.цифра[Остаток.длина+1]:=0;  
R:=Остаток.длина;  
L:=Остаток.длина-Делитель.длина;  
IF Остаток.цифра[R]<b.цифра[Делитель.длина] THEN  
  (*Сдвиг еще на один разряд*)  
  R:=R-1;L:=L-1;  
  Частное.длина:=Делимое.длина-Делитель.длина-1;  
ELSE  
  Частное.длина:=Делимое.длина-Делитель.длина;  
END;  
Номер:=Частное.длина+1;
```

Одновременно с первым отрезком вычисляется длина будущего частного, чтобы получаемые разряды частного записывать сразу в старшие элементы массива частного. Длина определяется из того соображения, что сумма длин частного и остатка должны быть равны длине делимого. Под словом «длина» конечно же подразумеваем количество разрядов в числе, просто сказать «длина» несколько короче.

Задание для самостоятельной работы

Ответьте на вопрос: почему нельзя длину частного определить одной формулой как разность длины делимого и делителя? В каких случаях это можно, а в каких нельзя? Ответ на этот вопрос в нашем программном фрагменте уже есть, ваша задача – сформулировать его в терминах арифметики.

Далее необходимо запустить главный цикл. Но есть одно "но". Выполняя предварительную работу, мы выполняем оператор $L=L-1$;; можно предполагать, что величина L может оказаться меньше нуля.

Задание для самостоятельной работы

Сформулируйте арифметическое условие, приводящее к событию $L < 0$.

Это условие необходимо учесть перед началом работы главного цикла:

```
IF L>=0 THEN  
  WHILE (R>=Делитель.длина) DO  
    (*Вычисление очередной цифры*)  
  END;  
ELSE  
  Частное.длина:=0;  
  Частное.цифра[0]:=0;  
END;
```

Цикл работает до тех пор, пока правая граница (R) не станет меньше, чем длина делителя. Это естественное условие, если правая граница стала меньше, то делитель уже нельзя вычестить из текущего отрезка разрядов. На каждом шагу этого цикла должна вычисляться очередная цифра частного, очевидно, равная числу

вычитаний делителя из отрезка разрядов [L, R].

```

sum:=0;
WHILE Проверка() DO
  (*Очередное вычитание*)
  k:=L;
  WHILE k<=R DO
    Остаток.цифра[k]:=Остаток.цифра[k]-b.цифра[k-L];
    IF Остаток.цифра[k]<0 THEN
      Остаток.цифра[k]:=Остаток.цифра[k]+10;
      Остаток.цифра[k+1]:=Остаток.цифра[k+1]-1;
    END;
    k:=k+1;
  END;
  sum:=sum+1;
END;
```

Собственно вычитание выполняет внутренний цикл. Это обычное вычитание столбиком, с заемом десятки у старшего разряда, если есть в том необходимость. Такие вычитания выполняются до тех пор, пока это возможно. Возможность устанавливает функция **Проверка()**, k – номер вычитаемых разрядов пробегает от левой границы до правой. И на каждом шагу выполняется поразрядное вычитание. Величина sum считает количество вычитаний, следовательно, по завершении двух вложенных циклов **WHILE** значение sum – это искомая цифра частного.

По завершении расчета очередной цифры она заносится в соответствующий разряд частного, и выполняется смещение отрезка [L, R].

```

R:=R-1;
L:=L-1;
Номер:=Номер-1;
Частное.цифра[Номер]:=sum;
```

По завершению главного цикла выполняется дополнительная работа по отбрасыванию возможных ведущих нулей.

```

WHILE (Остаток.длина>0) & (Остаток.цифра[Остаток.длина]=0) DO
  Остаток.длина:=Остаток.длина-1;
END
WHILE (Частное.длина>0) & (Частное.цифра[Частное.длина]=0) DO
  Частное.длина:=Частное.длина-1;
END;
```

Задание для самостоятельной работы

Проанализируйте необходимость двух записанных выше циклов.

Все описанные выше операции заключены в условие

```
IF Делимое.длина >= Делитель.длина THEN  
  (* Все описанные операции *)  
ELSE  
  Частное.длина := 0;  
  Частное.цифра[0] := 0;  
END;
```

Это условие учитывает возможность делителя, большего, чем делимое, по длине.

Задание для самостоятельной работы

Условие, упомянутое выше, необязательно. Попробуйте модифицировать программу так, чтобы без него можно было обойтись. Фактически в нашей реализации два условия, учитывающие, ситуацию, когда делитель больше делимого.

Перейдем к анализу функции **Проверка()**. Её работа заключается в проверке возможности вычитания делителя из отрезка [L, R]. Вычитание возможно, если разряд делимого, следующий за R, ненулевой и, невозможно, если левая граница меньше нуля:

```
IF Остаток.цифра[R+1] > 0 THEN  
  RETURN TRUE  
ELSE  
  IF L < 0 THEN  
    RETURN FALSE  
  ELSE  
    END;  
END;
```

Это очевидные ситуации. Во всех иных случаях необходим более детальный анализ числа. Пусть два числа даны массивом своих цифр. Как можно их сравнить? Очень просто. Необходимо пройти все цифры, начиная со старшей. Если окажется, что очередная цифра делимого больше цифры делителя, то делимое пока еще превосходит делитель:

```
IF Остаток.цифра[k] > Делитель.цифра[Делитель.длина+k-R] THEN  
  RETURN TRUE;  
END;
```

Если же больше цифра делителя, то делимое (а роль делимого играет массив остатка) уже меньше делителя:

```
IF Остаток.цифра[k] < Делитель.цифра[Делитель.длина+k-R] THEN  
  RETURN FALSE;  
END;
```

Если же удалось пройти все цифры и ни одно из условий не выполнилось, то числа равны и еще одно вычитание возможно, и в этом случае также возвращается Истина. Осталось лишь привести полный листинг.

Листинг 4.5

```

PROCEDURE Деление(OUT Частное,Остаток:Число; IN Делимое,Делитель:Число);
VAR
  L,R,sum,k,c,t,Номер:INTEGER;
  flag:BOOLEAN;
PROCEDURE Проверка():BOOLEAN;
VAR
  k:INTEGER;
BEGIN
  IF Остаток.цифра[R+1]>0 THEN
    RETURN TRUE
  ELSE
    IF L<0 THEN
      RETURN FALSE
    ELSE
      k:=R;
      WHILE k>=L DO
        IF Остаток.цифра[k]>Делитель.цифра[Делитель.длина+k-R] THEN
          RETURN TRUE;
        END;
        IF Остаток.цифра[k]<Делитель.цифра[Делитель.длина+k-R] THEN
          RETURN FALSE;
        END;
        k:=k-1;
      END;
      RETURN TRUE;
    END;
  END;
END Проверка;
BEGIN
  Остаток:=Делимое;
  IF Делимое.длина>=Делитель.длина THEN
    Остаток.цифра[Остаток.длина+1]:=0;
    R:=Остаток.длина;
    L:=Остаток.длина-Делитель.длина;
    IF Остаток.цифра[R]<Делитель.цифра[Делитель.длина] THEN
      (*Сдвиг еще на один разряд*)
      R:=R-1;L:=L-1;
      Частное.длина:=Делимое.длина-Делитель.длина-1;
    ELSE
      Частное.длина:=Делимое.длина-Делитель.длина;
    END;
    Номер:=Частное.длина+1;
  
```



```
IF L>=0 THEN
WHILE (R>=Делитель.длина) DO
  (*Вычисление очередной цифры*)
  sum:=0;
  WHILE Проверка() DO
    (*Очередное вычитание*)
    k:=L;
    WHILE k<=R DO
      Остаток.цифра[k]:=Остаток.цифра[k]-Делитель.цифра[k-L];
      IF Остаток.цифра[k]<0 THEN
        Остаток.цифра[k]:=Остаток.цифра[k]+10;
        Остаток.цифра[k+1]:=Остаток.цифра[k+1]-1;
      END;
      k:=k+1;
    END;
    sum:=sum+1;
  END;
  (*Смещение отрезка*)
  R:=R-1;
  L:=L-1;
  Номер:=Номер-1;
  Частное.цифра[Номер]:=sum;
END; (*Главный цикл завершен*)
WHILE (Остаток.длина>0) & (Остаток.цифра[Остаток.длина]=0) DO
  Остаток.длина:=Остаток.длина-1;
END
WHILE (Частное.длина>0) & (Частное.цифра[Частное.длина]=0) DO
  Частное.длина:=Частное.длина-1;
END;
ELSE
  Частное.длина:=0;
  Частное.цифра[0]:=0;
END;
ELSE
  Частное.длина:=0;
  Частное.цифра[0]:=0;
END;
END Деление;
```

Последнее замечание по реализации. Реализация процедуры деления великовата по сравнению с другими операциями. Попробуйте написать более короткую программу. Но имейте в виду, что ваша более короткая реализация деления столбиком должна работать, как минимум, так же эффективно. Автору этих строк известен более короткий вариант решения, но он более медленный.

Небольшой трюк для быстрого деления

Сейчас мы опишем прием, который не имеет высокой практической ценности, но тем не менее может оказаться полезным в некоторых частных случаях и, возможно, допускает какое-то расширение своей области. Предположим, что результат деления с остатком большего A на меньшее B уже известен. Этот результат выражается так:

$$A = pB + q.$$

Найдем результат деления A на $B+1$, не выполняя деления явным образом, а лишь используя уже известные p и q . То, что это возможно, видно из следующей формулы:

$$A = p(B + 1) - p + q \text{ при } -p + q > 0.$$

Листинг 4.6

```
PROCEDURE БыстроеДеление(OUT НовоеЧастное,НовыйОстаток:Число;IN
Частное,Остаток:Число;VAR Делитель:Число);
VAR
  C,quot,rem:Число;
BEGIN
  Вычитание(C,Частное,Остаток); (*Какая то процедура вычитания*)
  ПлюсЕдиница(Делитель); (*Увеличение делителя на 1*)
  Деление(rem,quot,C,Делитель); (*Какая-то процедура деления*)
  IF (ostatok.long=0) & (ostatok.digit[0]=0) THEN
    Вычитание(НовоеЧастное,Частное,rem);
    НовыйОстаток.длина:=0;
    НовыйОстаток.цифра[0]:=0;
  ELSE
    ПлюсЕдиница(rem);
    Вычитание(НовоеЧастное,Частное,rem);
    Вычитание(НовыйОстаток,Делитель,quot);
  END;
END БыстроеДеление;
```

Задание для самостоятельной работы

Попробуйте довести процедуру быстрого деления до полноценной программы.

Задание для самостоятельного исследования

Полученная процедура быстрого деления действительно быстро делит A на $B+1$. Но если речь идет о переходе не на 1, а на большее значение, то выигрыш в скорости очень быстро превращается в проигрыш. Вопрос: можно ли, пользуясь информацией о частном и остатке, получить выигрыш в делении A на $B+k$? Если да, то как? Если нет, то почему?

Некоторые другие алгоритмы

Алгоритм быстрого возведения в степень

Пусть необходимо вычислить B^N . Показатель степени может быть как четным, так и нечетным числом. В зависимости от четности показателя степень сводится к степени от меньшего показателя. А именно:

- если показатель четен: $N = 2m$, то $B^N = (B * B)^m$;
- если показатель нечетен: $N = 2m + 1$, то $B^N = (B^{N-1}) * B$.

Таким образом, получено рекуррентное определение степени, которое можно реализовать в виде рекурсивной процедуры. Единственное – здесь необходимо добавить условие завершения рекурсии, то есть случай, когда степень можно вычислить непосредственно – это следующее условие: $a^1 = a$. Возможен вариант $a^0 = 1$.

Листинг 4.7

```
PROCEDURE РасчетСтепени(a,n:INTEGER):INTEGER;  
BEGIN  
  IF n=1 THEN  
    RETURN a  
  ELSE  
    IF n MOD 2=0 THEN  
      RETURN РасчетСтепени(a*a, n DIV 2);  
    ELSE  
      RETURN a*РасчетСтепени(a,n-1);  
    END;  
  END;  
END РасчетСтепени;
```

Задания для самостоятельной работы

- Задача умножения двух чисел имеет много общего с задачей возведения в степень. Попробуйте небольшим изменением процедуры превратить её в процедуру умножения двух чисел.
- Напишите нерекурсивный вариант процедуры.

Замечание. Использовать данную процедуру в качестве средства умножения вряд ли разумно, так как она использует не менее трудоемкую операцию деления. Но если речь идет о длинной арифметике, в которой для представления числа используются, например, массивы, ситуация может измениться. Для числа, представленного массивом, легко воспользоваться признаком делимости на 2, что позволит избавиться от операции MOD. Для целочисленного деления на два также нет необходимости в делении общего вида, вполне можно разработать специальную процедуру деления на 2. Проанализируйте эту идею, попробуйте выяснить, не удастся ли таким образом получить способ, более быстрый, чем умножение столбиком.

Быстрый перевод из десятичной в двоичную систему счисления

Перевод десятичного числа в двоичное – довольно трудоёмкое дело. Конечно, небольшие трёх- или четырехзначные числа перевести не слишком сложно, но с ростом количества знаков объем выполняемой работы быстро растет. Рассмотрим способ, позволяющий быстро переводить в двоичное представление большие десятичные числа.

Такой способ был придуман французским математиком Лежандром. Пусть, например, дано число 11183445. Делим его на 64, получаем остаток 21 и частное 174741. Это число делим опять на 64, получаем в остатке 21 и частное 2730. Наконец, 2730, деленное на 64, даёт в остатке 42 и частное 42. Но 64 в двоичной системе есть 1000000, 21 в двоичной системе – 10101, а 42 есть 101010. Поэтому исходное число запишется в двоичной системе следующим образом:

$$\begin{array}{cccc} 42 & 42 & 21 & 21 \\ \hline 101010 & 101010 & 010101 & 010101 \end{array}$$

Чтобы было более понятно, ещё один пример с числом поменьше. Переведём в двоичное представление число 235. Поделим 235 на 64 с остатком. Получим:

$$\begin{aligned} \text{ЧАСТНОЕ} &= 3, \text{ двоичное } 11, \text{ или } 000011, \\ \text{ОСТАТОК} &= 43, \text{ двоичное } 101011. \end{aligned}$$

Тогда $235 = 11101011$. Проверим этот результат:

$$11101011 = 2^7 + 2^6 + 2^5 + 2^3 + 2^1 + 2^0 = 128 + 64 + 32 + 8 + 2 + 1 = 235.$$

Примечания:

1. Нетрудно заметить, что в окончательное двоичное число включаются все остатки и на последнем шаге и остаток и, частное.
2. Частное записывается перед остатком.
3. Если полученное частное или остаток имеют меньше 6 разрядов в двоичном представлении (6 нулей содержит двоичное представление числа $64 = 1000000$), то к нему добавляются незначащие нули.

И еще один сложный пример. Число 25678425.

Шаг 1: 25678425 делим на 64.

$$\text{Частное} = 401225; \text{ Остаток} = 25 = 011001.$$

Шаг 2: 401225 делим на 64.

$$\text{Частное} = 6269; \text{ Остаток} = 9 = 001001.$$

Шаг 3: 6269 делим на 64.

$$\text{Частное} = 97; \text{ Остаток} = 61 = 111101.$$

Шаг 4: 97 делим на 64.

$$\text{Частное} = 1 = 000001; \text{ Остаток} = 33 = 100001.$$

Число-результат = 1.100001.111101.001001.011001.

В этом числе точкой отделены входящие в него промежуточные результаты.

Задание для самостоятельной работы

В наших примерах использовано число 64. Здравый смысл говорит, что число 64 вряд ли принципиально отличается от других степеней двойки. Попробуйте проделать приведенные выше расчеты с другими степенями двойки и подумайте о оптимальности выбора степени двойки в зависимости от величины переводимого числа.

Задание для самостоятельной работы

Напишите реализацию изложенного метода.

Решение диофантовых уравнений

Любое уравнение вида $f(x_1, x_2, \dots, x_n) = 0$ превращается в уравнение Диофанта, если потребовать поиска только целых корней. Это требование делает интересными даже простые уравнения. Например, уравнение $ax + by = c$ в действительных числах решается элементарно: $y = (c - ax)/b$, где x – независимая переменная, а y – зависимая. Такое уравнение имеет бесконечно много корней, и это все, что можно сказать в данной ситуации. Требование целочисленности корней все меняет. Рассмотрим самый простой вариант уравнения Диофанта – уравнение первой степени с одной переменной:

$$ax + c = 0.$$

Его решение пока не создает проблем, но уже здесь просто поделить $-c$ на a нельзя, нужно выяснить, делится ли величина c на величину a нацело. Уравнение первой степени с двумя переменными – уравнение вида:

$$ax + by = c$$

уже не тривиально. Оно может иметь бесконечно много корней, и одношаговыми преобразованиями их не получить. Проведем небольшой математический анализ. Сразу заметим, что есть смысл потребовать $\text{НОД}(a, b) = 1$. Предположим, что это не так и $\text{НОД}(a, b) = d$, тогда исходное уравнение приобретает вид:

$$d(qx + py) = c.$$

Очевидно, что в этом случае уравнение имеет решение, только если c делится нацело на d , $c = dr$. В этом случае исходное уравнение $ax + by = c$ преобразуется в уравнение того же вида, но с несколько иными коэффициентами: $qx + py = r$. Итак, пусть $\text{НОД}(a, b) = 1$. Оказывается, решение нашего уравнения тесно связано с понятием цепной дроби. Рассмотрим пример: $15x - 23y = 7$. Разложим в цепную дробь отношение коэффициентов при x и y .

$$\frac{15}{23} = \frac{1}{\frac{23}{15}} = \frac{1}{1 + \frac{8}{15}} = \frac{1}{1 + \frac{1}{\frac{15}{8}}} = \frac{1}{1 + \frac{1}{1 + \frac{7}{8}}} = \frac{1}{1 + \frac{1}{1 + \frac{1}{\frac{8}{7}}}} = \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{7}{7}}}}$$

Далее просто отбросим последний результат и свернем цепную дробь обратно:

$$\frac{1}{1 + \frac{1}{1+1}} = \frac{1}{1 + \frac{1}{2}} = \frac{1}{\frac{3}{2}} = \frac{2}{3}$$

Вычтем полученное значение из исходной дроби:

$$\frac{15}{23} - \frac{2}{3} = \frac{15 * 3 - 2 * 23}{3 * 23} = -\frac{1}{3 * 23}$$

Отбросим общий знаменатель $15 * 3 - 2 * 23 = -1$. Чтобы получить исходное уравнение, умножим обе части на -7 : $15 * (-21) - 23 * (-14) = 7$; сравнив полученное выражение с исходным, сделаем вывод, что корни уравнения следующие: $x = -21$; $y = -14$. Это не единственные корни. В книге Гельфонда [13] можно найти детальное изложение процедуры получения всех корней. Мы лишь приведем расчетные формулы в общем виде:

$$x = (-1)^{n-1} cQ_{n-1} - bt \quad \text{и} \quad y = (-1)^n cP_{n-1} + at, \quad \text{где } t = 0, \pm 1, \pm 2, \dots$$

Формулы говорят о том, что решений бесконечно много и все они отличаются от одного-единственного на величины bt и at . Следовательно, достаточно найти одно решение, все остальные выражаются через него. Процедура определения одного решения выше описана. Единственное немного поясним смысл величин Q и P . Выражение P/Q называется подходящей дробью. В нашем примере результат был получен посредством подходящей дроби $2/3$. Она была получена отбрасыванием последней дроби в цепном представлении отношения коэффициентов. Поэтому её номер $n-1$. Если вместе с последней отбросить предпоследнюю, то получившаяся подходящая дробь будет иметь номер $n-2$ и т. д.

Задание для самостоятельной работы

Процесс построения цепной дроби можно представить как расчет последовательности подходящих дробей. Более того, зная очередную подходящую дробь, можно вычислить следующую. Попробуйте найти рекуррентное выражение для подходящих дробей.

В заключение об уравнениях Диофанта. Мы рассмотрели самую простую форму уравнения, допускающую хорошее алгоритмическое решение. Есть и дру-

гие виды диофантовых уравнений, для которых удалось найти хорошее решение. Это, например, уравнение Пелля:

$$x^2 - Ay^2 = 1.$$

есть и другие. Самым знаменитым целочисленным уравнением, наверное, является уравнение

$$x^n + y^n = z^n,$$

в отношении которого французский математик Ферма сформулировал теорему, названную Большой теоремой Ферма, или даже Великой теоремой. Эта теорема утверждает, что для $n > 2$ это уравнение не имеет решений в целых числах. Есть методы решений для самых разных уравнений, но все же уравнения Диофанта содержат в себе больше вопросов, чем ответов. Кроме того, уже доказано, что оптом на все вопросы решения целочисленных уравнений ответить нельзя. Проблема решения уравнения Диофанта оказывается алгоритмически неразрешима. Этот факт доказан. Поэтому каждое уравнение есть тема отдельного исследования. Если проблемы поиска целых корней вас интересуют, то кроме уже упоминавшейся книги Гельфонда можем посоветовать книгу польского математика Серпинского [14].

Двоичная арифметика

В начале главы немного сказано о системах счисления. Их может быть сколько угодно много. С точки зрения фундаментальной математики, выбор системы счисления ничего не решает. Но мы имеем дело не с высокой математикой, наша цель — вычисление величин, по возможности быстро и просто, с учетом возможности машинной реализации. С этой точки зрения не все равно, какую систему выбрать. Выбор двоичной системы выше уже был обоснован. Поэтому сейчас еще раз небольшой рассказ об арифметике, но уже только двоичной. Некоторые вопросы мы повторим, повтор часто бывает нелишним, и кроме того, систематическое изложение именно двоичных операций сделает этот параграф полезным, даже если вы во всей главе прочитаете только его. Итак, число называется двоичным, если оно представимо в виде

$$a_n a_{n-1} \dots a_2 a_1 = a_n * 2^{n-1} + a_{n-1} * 2^{n-2} + \dots + a_2 * 2^1 + a_1 * 2^0,$$

где a_i — это символ из набора $\{0, 1\}$.

Рассмотрим вопросы перевода из десятичного представления в двоичное и из двоичного в десятичное.

Двоичное в десятичное. Это очень просто. Метод такого перевода даёт наш способ записи чисел. Возьмём, к примеру, следующее двоичное число 1011. Разложим его по степеням двойки. Получим следующее:

$$1011 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0.$$

Выполним все записанные действия и получим:

$$1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 8 + 0 + 2 + 1 = 11.$$

Таким образом, получаем, что 1011(двоичное) = 11 (десятичное). Сразу видно и небольшое неудобство двоичной системы. То же самое число, которое в десятичной системе записано двумя знаками, в двоичной системе для своей записи требует четыре знака. Но это плата за простоту (бесплатно ничего не бывает). Но выигрыш двоичная система даёт огромный в арифметических действиях. И далее мы это увидим.

Задание для самостоятельной работы

Представьте в виде десятичного числа следующие двоичные числа:

а) 10010; б) 11101; в) 1010; г) 1110; д) 100011; е) 1100111; ж) 1001110.

Сложение двоичных чисел

Способ сложения столбиком – такой же, как и для десятичного числа. Вспомним, как это делается. Сложение выполняется поразрядно, начиная с младшей цифры. Если при сложении двух цифр получается СУММА больше девяти, то записывается цифра = СУММА – 10, а ЦЕЛАЯ ЧАСТЬ (СУММА/10) добавляется к старшему разряду. Так и с двоичным числом. Складываем поразрядно, начиная с младшей цифры. Если получается 2, то записывается 0 и 1 добавляется к старшему разряду (говорят «на ум пошло»).

Выполним пример: 10011 + 10001.

Таблица 4.6

	1	0	0	1	1
	1	0	0	0	1
1	0	0	1	0	0

Первый разряд: 1+1 = 2. Записываем 0, и 1 на ум пошло.

Второй разряд: 1+0+1 (запомненная единица) = 2. Записываем 0, и 1 на ум пошло.

Третий разряд: 0+0+1(запомненная единица) = 1. Записываем 1.

Четвертый разряд: 0+0 = 0. Записываем 0.

Пятый разряд: 1+1 = 2. Записываем 0 и добавляем к шестым разрядом 1.

Переведём все три числа в десятичную систему и проверим правильность сложения.

$$10011 = 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 16 + 2 + 1 = 19.$$

$$10001 = 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 16 + 1 = 17.$$

$$100100 = 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 = 32 + 4 = 36.$$

$$17 + 19 = 36, \text{ верное равенство.}$$

Задание для самостоятельной работы

- а) $11001 + 101 =$
- б) $11001 + 11001 =$
- в) $1001 + 111 =$
- г) $10011 + 101 =$
- д) $11011 + 1111 =$
- е) $11111 + 10011 =$

Как десятичное число перевести в двоичное. На очереди следующая операция – вычитание. Но этой операцией мы займёмся немного позже, а сейчас рассмотрим метод преобразования десятичного числа в двоичное.

Для того чтобы преобразовать десятичное число в двоичное, его нужно разложить по степеням двойки. Но если разложение по степеням десятки получается сразу, то как разложить по степеням двойки, надо немного подумать. Для начала рассмотрим, как это сделать методом подбора. Возьмём десятичное число 12.

Шаг первый. $2^2 = 4$, этого мало. Также мало и $2^3 = 8$, а $2^4 = 16$ – это уже много.

Поэтому оставим $2^3 = 8$. $12 - 8 = 4$. Теперь нужно представить в виде степени двойки 4.

Шаг второй. $4 = 2^2$.

Тогда наше число $12 = 2^3 + 2^2$. Старшая цифра имеет номер 4, старшая степень = 3, следовательно, должны быть слагаемые со степенями двойки 1 и 0. Но они нам не нужны, поэтому, чтобы избавиться от ненужных степеней и оставить нужные, запишем число так: $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 1100$ – это и есть двоичное представление числа 12. Нетрудно заметить, что каждая очередная степень – это наибольшая степень двойки, которая меньше разлагаемого числа. Чтобы закрепить метод, рассмотрим ещё один пример. Число 23.

Шаг 1. Ближайшая степень двойки $2^4 = 16$. $23 - 16 = 7$.

Шаг 2. Ближайшая степень двойки $2^2 = 4$. $7 - 4 = 3$.

Шаг 3. Ближайшая степень двойки $2^1 = 2$. $3 - 2 = 1$.

Шаг 4. Ближайшая степень двойки $2^0 = 1$. $1 - 1 = 0$.

Получаем следующее разложение: $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$.

А наше искомое двоичное число 10111.

Рассмотренный выше метод хорошо решает поставленную перед ним задачу, но есть способ, который алгоритмизируется значительно лучше. Алгоритм этого метода записан ниже:

Пока ЧИСЛО больше нуля, делать

Начало

ОЧЕРЕДНАЯ ЦИФРА = остаток от деления ЧИСЛА на 2

ЧИСЛО = целая часть от деления ЧИСЛА на 2

Конец

Когда этот алгоритм завершит свою работу, последовательность вычисленных

ОЧЕРЕДНЫХ ЦИФР и будет представлять двоичное число. Для примера поработаем с числом 19.

Начало процесса

ЧИСЛО = 19

Шаг 1

ОЧЕРЕДНАЯ ЦИФРА = 1

ЧИСЛО = 9

Шаг 2

ОЧЕРЕДНАЯ ЦИФРА = 1

ЧИСЛО = 4

Шаг 3

ОЧЕРЕДНАЯ ЦИФРА = 0

ЧИСЛО = 2

Шаг 4

ОЧЕРЕДНАЯ ЦИФРА = 0

ЧИСЛО = 1

Шаг 5

ОЧЕРЕДНАЯ ЦИФРА = 1

ЧИСЛО = 0

Итак, в результате имеем следующее число 10011. Заметьте, что два рассмотренных метода отличаются порядком получения очередных цифр. В первом методе первая полученная цифра – это старшая цифра двоичного числа, а во втором – первая полученная цифра наоборот, младшая.

Как преобразовать в двоичное число дробную часть

Известно, что любое рациональное число можно представить в виде десятичной и обыкновенной дроби. Обыкновенная дробь, то есть дробь вида A/B , может быть правильной и неправильной. Дробь называется правильной, если $A < B$, и неправильной, если $A > B$.

Если рациональное число представлено неправильной дробью и при этом числитель дроби делится на знаменатель нацело, то данное рациональное число – число целое, во всех иных случаях возникает дробная часть. Дробная часть зачастую бывает очень длинным числом и даже бесконечным (бесконечная периодическая дробь возникает, например, при попытке перевода в десятичную следующей дроби $20/6$), поэтому в случае с дробной частью возникает не просто задача перевода одного представления в другое, а перевод с определённой точностью.

Правило точности. Предположим, дано десятичное число, которое в виде десятичной дроби представимо с точностью до N знаков. Для того чтобы соответствующее двоичное число было той же точности, в нём необходимо записать M знаков, так чтобы

$$2^M > 10^N.$$

А теперь попробуем получить правило перевода и для начала рассмотрим пример 5,401.

Решение: целую часть получим по уже известным правилам, и она равна двоичному числу 101. А дробную часть разложим по степеням 2.

Шаг 1: $2^{-2} = 0,25$; $0,401 - 0,25 = 0,151$ – это остаток.

Шаг 2: сейчас необходимо степенью двойки представить 0,151. Сделаем это: $2^{-3} = 0,125$; $0,151 - 0,125 = 0,026$.

Таким образом, дробную часть можно представить в виде $2^{-2} + 2^{-3}$. То же самое можно записать таким двоичным числом: 0,011. В первом дробном разряде стоит ноль, это потому, что в нашем разложении степень 2^{-1} отсутствует.

Из первого и второго шагов ясно, что это представление неточное и, может быть, разложение желательно продолжить. Обратимся к правилу. Оно говорит, что нам нужно столько знаков M , чтобы 10^3 было меньше, чем 2^M . То есть $1000 < 2^M$. То есть в двоичном разложении у нас должно быть не менее десяти знаков, так как $2^9 = 512$ и только $2^{10} = 1024$. Продолжим процесс.

Шаг 3: сейчас работаем с числом 0,026. Ближайшая к этому числу степень двойки $2^{-6} = 0,015625$; $0,026 - 0,015625 = 0,010375$; теперь наше более точное двоичное число имеет вид: 0,011001. После запятой уже шесть знаков, но этого пока недостаточно, поэтому выполняем ещё один шаг.

Шаг 4: сейчас работаем с числом 0,010375. Ближайшая к этому числу степень двойки $2^{-7} = 0,0078125$;

$$0,010375 - 0,0078125 = 0,0025625.$$

Шаг 5: сейчас работаем с числом 0,0025625. Ближайшая к этому числу степень двойки $2^{-9} = 0,001953125$;

$$0,0025625 - 0,001953125 = 0,000609375.$$

Последний получившийся остаток меньше, чем 2^{-10} , и если бы мы желали продолжать приближение к исходному числу, то понадобилось бы 2^{-11} , но это уже превосходит требуемую точность, а следовательно, расчёты можно прекратить и записать окончательное двоичное представление дробной части.

$$(0,401)_{10} = (0,011001101)_2.$$

Как видно, преобразование дробной части десятичного числа в двоичное представление немного более сложно, чем преобразование целой части.

А сейчас запишем алгоритм преобразования:

Исходные данные алгоритма: через A обозначим исходную правильную десятичную дробь. Пусть эта дробь содержит N знаков.

Алгоритм

Действие 1. Определим количество необходимых двоичных знаков M из неравенства $10^N < 2^M$.

Действие 2. Цикл вычисления цифр двоичного представления (цифры после нуля). Номер цифры обозначим символом K .

Номер цифры = 1

Если $2^{-K} > A$ То

в запись двоичного числа добавляем нуль

Иначе

в запись двоичного числа добавляем 1

$A = A - 2^{-K}$

$K = K + 1$

Если $K > M$

То работа алгоритма завершена

Иначе переходим на пункт 2.

Задание для самостоятельной работы

Переведите десятичные числа в двоичные:

а) 3,6; б) 12,0112; в) 0,231; г) 0,121; д) 23, 0091.

Задание для самостоятельной работы

Реализуйте предложенный алгоритм на КП.

Вычитание двоичных чисел

Вычитание столбиком выполняется поразрядно с заемом единицы в старшем разряде, если это необходимо. Пример:

Таблица 4.7

	1	1	0	1
–		1	1	0
		1	1	1

Первый разряд. $1 - 0 = 1$. Записываем 1.

Второй разряд. $0 - 1$. Не хватает единицы. Занимаем её в старшем разряде. Единица из старшего разряда переходит в младший как две единицы (потому что старший разряд представляется двойкой большей степени); $2 - 1 = 1$. Записываем 1.

Третий разряд. Единицу этого разряда мы занимали, поэтому сейчас в разряде 0 и есть необходимость занять единицу старшего разряда. $2 - 1 = 1$. Записываем 1.

Проверим результат в десятичной системе:

$$1101 - 110 = 13 - 6 = 7 \text{ (111)}.$$

Верное равенство.

Еще один интересный способ выполнения вычитания связан с понятием дополнительного кода, который позволяет свести вычитание к сложению. Получается число в дополнительном коде исключительно просто, берём число, заменяем нули на единицы, единицы, наоборот, заменяем на нули и к младшему разряду добавляем единицу. Например, 10010 в дополнительном коде будет 01110.

Правило вычитания через дополнительный код утверждает, что вычитание можно заменить на сложение, если вычитаемое заменить на число в дополнительном коде.

Пример: $34 - 22 = 12$

Запишем этот пример в двоичном виде: $100010 - 10110 = 1100$.

Дополнительный код числа 10110 будет такой:

$$01001 + 00001 = 01010.$$

Тогда исходный пример можно заменить сложением так:

$$100010 + 01010 = 101100.$$

Далее необходимо отбросить одну единицу в старшем разряде. Если это сделать, то получим 001100. Отбросим незначащие нули и получим 1100, то есть пример решён правильно.

Задание для самостоятельной работы

Выполните вычитания. Обычным способом и в дополнительном коде, переведя предварительно десятичные числа в двоичные:

а) $456 - 112 =$

б) $234 - 12 =$

в) $345 - 232 =$

г) $456 - 78 =$

д) $567 - 109 =$

е) $67 - 45 =$

Выполните проверку, переведя двоичный результат в десятичную систему счисления.

Умножение в двоичной системе счисления

Для начала рассмотрим следующий любопытный факт. Для того чтобы умножить двоичное число на 2 (десятичная двойка – это 10 в двоичной системе), достаточно к умножаемому числу слева приписать один ноль.

Пример: $10101 * 10 = 101010$.

Проверка:

$$10101 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 16 + 4 + 1 = 21$$

$$101010 = 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 32 + 8 + 2 = 42$$

$$21 * 2 = 42$$

Если мы вспомним, что любое двоичное число разлагается по степеням двойки, то становится ясно, что умножение в двоичной системе счисления сводится к умножению на 10 (то есть на десятичную 2), а стало быть, умножение – это ряд последовательных сдвигов. Общее правило таково: как и для десятичных чисел, умножение двоичных выполняется поразрядно. И для каждого разряда второго множителя к первому множителю добавляется один нуль справа. Пример (пока не столбиком):

$$1011 * 101.$$

Это умножение можно свести к сумме трёх поразрядных умножений:

$$1011 * 1 + 1011 * 0 + 1011 * 100 = 1011 + 101100 = 110111.$$

В столбик это же самое можно записать так:

Таблица 4.8

		1	0	1	1
	*		1	0	1
		1	0	1	1
	0	0	0	0	
1	0	1	1		
1	1	0	1	1	1

Проверка:

$$101 = 5 \text{ (десятичное).}$$

$$1011 = 11 \text{ (десятичное).}$$

$$110111 = 55 \text{ (десятичное).}$$

$$5 * 11 = 55, \text{ верное равенство.}$$

Задание для самостоятельной работы

а) $1101 * 1110 =$

б) $1010 * 110 =$

в) $1011 * 11 =$

г) $101011 * 1101 =$

д) $10010 * 1001 =$

Деление в двоичной системе счисления

Мы рассмотрели три действия, и уже понятно, что в общем-то действия над двоичными числами мало отличаются от действий над десятичными числами. Разница появляется только в том, что цифр две, а не десять, что только упрощает арифметические операции. Так же обстоит дело и с делением, но для лучшего понимания алгоритм деления разберём более подробно. Пусть нам необходимо разделить два десятичных числа, например 234 разделить на 7. Как мы это делаем?

Таблица 4.9

2	3	4	7	

Мы выделяем справа (от старшего разряда) такое количество цифр, чтобы получившееся число было как можно меньше и в то же время больше делителя. 2 – меньше делителя, следовательно, необходимое нам число 23. Затем делим полученное число на делитель с остатком. Получаем следующий результат:

Таблица 4.10

	2	3	4	7	
–	2	1		3	
		2	4		

Описанную операцию повторяем до тех пор, пока полученный остаток не окажется меньше делителя. Когда это случится, число, полученное под чертой, – это частное, а последний остаток – это остаток операции. Так вот операция деления двоичного числа выполняется точно так же. Попробуем.

Пример: 10010111 / 101.

Таблица 4.11

1	0	0	1	0	1	1	1	1	0	1

Ищем число от старшего разряда, которое первое было бы больше, чем делитель. Это четырехразрядное число 1001. Оно выделено жирным шрифтом. Теперь необходимо подобрать делитель выделенному числу. И здесь мы опять выигрываем в сравнении с десятичной системой. Дело в том, что подбираемый делитель – это обязательно значащая цифра, а цифры у нас только две и значащая только одна, это единица. Так как 1001 явно больше 101, то с делителем всё понятно, это 1. Выполним шаг операции.

Таблица 4.12

	1	0	0	1	0	1	1	1	1	0	1
–		1	0	1					1		
		1	0	0							

Итак, остаток от выполненной операции 100. Это меньше, чем 101, поэтому, чтобы выполнить второй шаг деления, необходимо добавить к 100 следующую цифру, это цифра 0. Теперь имеем следующее число:

Таблица 4.13

	1	0	0	1	0	1	1	1	1	0	1
–		1	0	1					1		
		1	0	0	0						

1000 больше 101, поэтому на втором шаге мы опять допишем в частное цифру 1 и получим следующий результат (для экономии места сразу опустим следующую цифру).

Таблица 4.14

	1	0	0	1	0	0	1	1	1	0	1
–		1	0	1					1	1	
		1	0	0	0						
	–		1	0	1						
				1	1	0					

Третий шаг. Полученное число 110 больше 101, поэтому и на этом шаге мы запишем в частное 1. Получится так:

Таблица 4.15

	1	0	0	1	0	0	1	1	1	0	1
–		1	0	1					1	1	1
		1	0	0	0						
	–		1	0	1						
				1	1	0					
			–	1	0	1					
						1	1				

Полученное число 11 меньше 101, поэтому записываем в частное цифру 0 и опускаем вниз следующую цифру. Получается так:

Таблица 4.16

	1	0	0	1	0	0	1	1		1	0	1		
–		1	0	1						1	1	1	0	
		1	0	0	0									
	–		1	0	1									
				1	1	0								
			–	1	0	1								
						1	1	1						

Полученное число больше 101, поэтому в частное записываем цифру 1 и опять выполняем действия. Получается такая картина:

Таблица 4.17

	1	0	0	1	0	0	1	1		1	0	1		
–		1	0	1						1	1	1	0	1
		1	0	0	0									
	–		1	0	1									
				1	1	0								
			–	1	0	1								
						1	1	1						
					–	1	0	1						
							1	0						

Полученный остаток 10 меньше 101, но у нас закончились цифры в делимом, поэтому 10 – это окончательный остаток, а 1110 – это искомое частное.

Проверим в десятичных числах:

$$10010011 = 147.$$

$$101 = 5.$$

$$10 = 2.$$

$$11101 = 29.$$

Таблица 4.18

	1	4	7	5	
–	1	0		2	9
		4	7		
	–	4	5		
			2		

Задание для самостоятельной работы

Простота подбора очередной цифры частного (только один вариант – 1) должна существенно упростить реализацию алгоритма деления столбиком. Напишите столбиковый алгоритм для деления двоичных чисел и сравните скорости работы нашего десятичного столбика и вашего двоичного. Есть ли выигрыш в скорости? Если да, то почему? Если нет, то опять же почему?

В заключение. На этом мы заканчиваем описание простейших арифметических операций. Наверное, вы уже убедились, что они не так просты, как может показаться, даже если реализовать столбиковую арифметику. Арифметика, как оказывается, становится довольно сложной вещью, если поставить цель не просто выполнить операцию, а выполнить её по возможности быстро. А специальная организация длинной арифметики требует некоторого технического мастерства, даже без борьбы за скорость.

Рекурсия и динамическое программирование

Общее определение.....	132
Задача о ханойской башне	135
Переход от рекурсивного к нерекурсивному решению.....	138
Рекурсия как метод поиска.....	143
Динамическое программиро- вание	144

Общее определение

Говоря о рекурсии, мы имеем в виду специальный способ построения программы, при котором в тексте появляются процедуры, вызывающие сами себя в процессе выполнения. Собственно, этим рекурсия определена полностью. Но возможность «обращения к себе» дает рекурсивной процедуре совершенно уникальные свойства, делающие рекурсию мощным и многогранным инструментом. Рассмотрением граней мы в этой главе и займемся, но начнем с ответа на два простых вопроса: зачем это может понадобиться и как это организовать.

Зачем это может понадобиться. Достаточно часто программист работает с величинами, расчет которых имеет рекуррентный характер. Рассмотрим в качестве первого примера задачу счета факториала. Определение факториала гласит, что

$$N! = 1 * 2 * \dots * N.$$

Это не рекуррентное определение. Выполнив простое преобразование, получим:

$$N! = N * (N - 1)!$$

А это уже рекуррентное определение, то определение, в котором функция факториала определяется через неё же, но от меньшего аргумента. Как только получено рекуррентное определение, можно считать, что программа уже написана. В таком простом случае разработка программы сводится к записи определения на языке программирования.

Листинг 5.1

```
MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE Главная*;
VAR
  N:INTEGER;
PROCEDURE Факториал(n: INTEGER):INTEGER;
BEGIN
  IF n=1 THEN
    RETURN 1; (*Прямое вычисление*)
  ELSE
    RETURN n*Факториал(n-1); (*Переход к меньшему аргументу*)
  END;
END Факториал;
BEGIN
  In.Open;
  In.Int(N);
  StdLog.Int(Факториал(N));
END Главная;
END Модуль.
```

Пример с факториалом использован только для демонстрации природы рекурсии. Мы хотели лишь показать, что рекурсия естественным образом возникает там, где удастся дать величине рекуррентное определение. Факториал можно определить рекуррентно, поэтому для факториала возможно построение рекурсивной процедуры. Однако использование рекурсии не всегда оправдано. Пример расчета факториала не только хорошо показывает механизм рекурсии, но это также хороший пример того, когда рекурсию не надо использовать. Но из примера факториала все же следует ответ на вопрос "*зачем это нужно?*" Это нужно для вычисления рекуррентно определенных величин. Просто нужно привести хороший пример. Такие примеры в главе будут, а сейчас еще немного отрицательной информации.

Почему поиск рекурсивного решения не всегда оправдан? Если в тексте процедуры существует вызов её же, то в момент передачи управления на этот вызов выполняется активация новой копии процедуры с новым набором локальных переменных, при этом набор локальных величин той копии, которая явилась причиной активации, также сохраняется в оперативной памяти. Таким образом, рекурсивный процесс ведет к тому, что в оперативной памяти сохраняются наборы данных всех активированных копий процедур. Уничтожение данных каждой копии происходит только по завершении работы копии и возврату управления в точку вызова процедуры.

Например, для вычисления $20!$ будет выполнено 20 активаций и в ОЗУ возникнет 20 наборов данных. Понятно, что это не всегда разумная трата памяти. Тем более что счет факториала легко реализуется элементарной последовательностью итераций. Отсюда – простое правило: *если задача реализуется линейной последовательностью итераций, то рекурсивное решение возможно, но не разумно*. Поэтому поиск рекурсивного решения оправдан, если процесс нельзя представить в виде линейной последовательности итераций, то есть если процесс ветвящийся. Немного позже попробуем сформулировать общие свойства рекурсии, а сейчас пример ветвящегося процесса, для которого применение рекурсии уже вполне оправдано.

Задача Дейкстры. Пусть дана функция, определяемая следующими соотношениями:

$$\begin{cases} F(1) = 1 \\ F(2N) = F(N) \\ F(2N + 1) = F(N) + F(N + 1) \end{cases}$$

Требуется вычислить значение функции. О чем говорят формулы: вторая формула утверждает, что если аргумент функции четен, то функция выражается через функцию с вдвое меньшим аргументом. Третья формула утверждает, что если аргумент нечетен, то функция выражается через сумму двух функций, таких что сумма их аргументов равна аргументу исходной функции, а друг от друга они отли-

чаются на единицу. И только первая формула дает возможность определить значение функции непосредственно. Благодаря третьей формуле процесс вычисления функции ветвится. На каждом шагу расчетов появляется некоторое количество функций с четным аргументом и некоторое количество с нечетным. Эти количества, скорее всего, подчиняются хорошей закономерности. Но закономерность еще нужно найти, в то время как рекурсивное решение практически лежит на ладони. Достаточно записать три формулы условия в виде условных операторов. Листинг ниже показывает, как может выглядеть реализация такой функции.

Листинг 5.2

```
PROCEDURE ФункцияДейкстры(n: INTEGER):INTEGER;  
BEGIN  
  IF n=1 THEN RETURN 1;  
  ELSIF (n MOD 2=0) THEN RETURN ФункцияДейкстры(n DIV 2);  
  ELSE RETURN ФункцияДейкстры(n DIV 2)+ФункцияДейкстры(n DIV 2 + 1)  
  END;  
END ФункцияДейкстры;
```

Как рекурсию организовать? Приступим к детальному разбору реализации рекурсии. Самый общий принципиальный момент уже ясен. Рекурсия – это запись рекуррентного определения. Это во-первых. Во-вторых, необходимо заметить, что рекурсия, как и конструкция цикла, дает программисту возможность организовать многократно повторяющееся действие. Для конструкции цикла во избежание зависания предусмотрено либо условие продолжения, либо условие завершения работы цикла, в общем, некоторое, условие ограничивающее процесс. Очевидно, такое условие необходимо и для рекурсии. Для рекурсии, впрочем, надо несколько больше. Необходимо, чтобы глубина вызовов была не слишком большой, это требование обусловлено ограничениями по памяти, но это уже забота программиста, компилятор здесь программисту не помощник. А вот критерий завершения процесса рекурсивных вызовов записать необходимо. В процедуре, решающей задачу Дейкстры, это следующее условие:

```
IF n=1 THEN RETURN 1;
```

соответствующее единственной формуле, умеющей вычислять искомую величину непосредственно. Отсюда можно сделать вывод, что для рекурсивных вычислительных процессов необходимо задать минимальное или максимальное значение некоторого параметра, при котором искомая величина будет явно посчитана.

Но сказанное справедливо только как частный случай. Достаточно легко придумать задачу, цель которой – не вычисление величин. В качестве простого примера можно взять задачу рисования следующей фигуры (рис. 5.1).

На рисунке показаны четыре этапа рисования фигуры. Заметим, что вся фигура состоит из отрезков уменьшающейся длины, и каждый из нарисованных отрезков порождает еще два. Вертикальный отрезок порождает два горизонтальных, горизонтальный порождает два вертикальных. Налицо рекуррентное определение

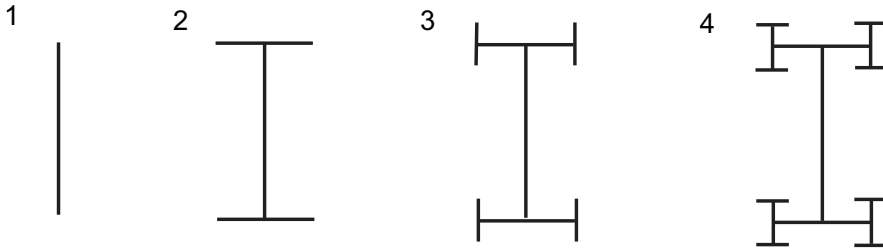


Рис. 5.1

процесса, следовательно, возможно рекурсивное решение. Но здесь не вычисляется никакой величины, следовательно, нет никакого тривиального числового условия, которым можно было бы завершить последовательность вызовов. Результат работы программы – фигура, рисуемая на каждом вызове. Здесь процесс рисования можно завершить по достижении некоторой глубины вызовов. Например, так:

```
PROCEDURE РекурсивнаяПроцедура(список формальных параметров);  
BEGIN  
  IF Номер вызова < Максимально возможного THEN  
    Операции рисования и рекурсивный вызов  
  END;  
END РекурсивнаяПроцедура;
```

И мы подошли к определению. Для завершения последовательности рекурсивных вызовов необходимо определить некоторый параметр, в общем случае числовую функцию, изменение которой вычисляется на каждом шаге рекурсивного процесса и для которой определяется одно или несколько значений, являющихся сигналом для прерывания рекурсивных активаций. Достаточно часто эта функция сводится к числовому параметру, имеющему начальное значение, запускающее рекурсивный процесс и, конечно, его останавливающее. Рассмотрим более сложный пример рекурсивного определения.

Задача о ханойской башне

Старая классическая задача, встречающаяся во всех учебниках программирования. Смысл задачи в следующем: даны 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.

Попробуем обнаружить рекуррентную природу задачи. Это легко сделать, посмотрев на два рисунка (рис. 5.2 и рис. 5.3).

Ханойская башня, исходное положение.



Рис. 5.2

Ханойская башня, промежуточное положение.



Рис. 5.3

Если удастся получить ситуацию, изображенную на рис. 5.3, 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. Ветвящийся характер процесса очевиден, следовательно, поиск рекурсивного решения вполне оправдан. Тривиальный случай (когда действие выполняется непосредственно) для построения рекурсивной процедуры тоже понятен – это перестановка башни из одного диска.

Листинг 5.3

```
MODULE Модуль;
IMPORT In, StdLog;
TYPE ТипМассивДисков=ARRAY 10 OF INTEGER;
PROCEDURE Главная*;
VAR
  Диски:ARRAY 4 OF ТипМассивДисков;
  Высота:ARRAY 4 OF INTEGER;
  n,k:INTEGER;
PROCEDURE Печать;
VAR
  k,j:INTEGER;
BEGIN
  FOR k:=1 TO 3 DO
    StdLog.String(' ');
    FOR j:=1 TO Высота[k] DO
      StdLog.Int(Диски[k,j])
```



```

END;
StdLog.String(' ');
END;
StdLog.Ln;
END Печать;
PROCEDURE ХанойскаяБашня(a1,a2,a3,m:INTEGER);
PROCEDURE Перестановка;
BEGIN
    Высота[a3]:= Высота[a3]+1;
    Диски[a3, Высота[a3]]:= Диски[a1, Высота[a1]];
    Диски[a1, Высота[a1]]:=0;
    Высота[a1]:= Высота[a1]-1;
    Печать;
END Перестановка;
BEGIN
    IF m=1 THEN
        Перестановка;
    ELSE
        ХанойскаяБашня(a1,a3,a2,m-1);
        Перестановка;
        ХанойскаяБашня(a2,a1,a3,m-1);
    END;
END ХанойскаяБашня;
BEGIN
    In.Open;
    In.Int(n);
    Высота[1]:=n; Высота[2]:=0; Высота[3]:=0;
    FOR k:=1 TO n DO
        Диски[1,k]:=n-k+1;
    END;
    Печать;
    ХанойскаяБашня(1,2,3,n);
END Главная;
END Модуль.

```

Здесь рекурсивный переход немного сложнее. В предыдущих примерах рекурсивная постановка задачи заключалась в сведении задачи к той же задаче, но от меньшего аргумента. Здесь же обратите внимание на фрагмент:

```

ХанойскаяБашня(a1,a3,a2,m-1);
Перестановка;
ХанойскаяБашня(a2,a1,a3,m-1);

```

Этот фрагмент означает, что задача о переносе башни высоты m с подставки $a1$ на подставку $a3$ с помощью подставки $a2$ сводится к двум задачам:

1. Перенос башни высоты $m-1$ с подставки a_1 на подставку a_2 с помощью подставки a_3 .
2. Перенос башни высоты $m-1$ с подставки a_2 на подставку a_3 с помощью подставки a_1 .

И между этими двумя операциями выполняется тривиальная операция по перестановке башни из одного диска. Отсюда видно, что тривиальная задача всегда присутствует, но не обязательно на этапе завершения. Реализация тривиального случая может быть необходима на промежуточных этапах. Задача о ханойской башне – хороший пример достаточно сложной проблемы, решаемой принципиально так же, как и задача о счете факториала. А именно если дано точное рекуррентное определение, то разработка рекурсивной процедуры фактически сводится к записи этого определения.

Переход от рекурсивного к нерекурсивному решению

Любую задачу можно решить, как рекурсивно так и нерекурсивно. Вопрос заключается только в целесообразности. Иногда бывает, что рекурсивное решение сильно упрощает жизнь программисту и при этом не требует слишком больших ресурсов, в этом случае рекурсивное решение безусловно оправдано. Иногда бывает, как в задаче о факториалах, ресурсов требуется немного, но нерекурсивное решение все же проще. А иногда бывает, что рекурсивное решение очень просто, но требует столько ресурсов, сколько мы не можем себе позволить, в этом случае необходимо искать нерекурсивное решение. В таком поиске программист может пойти сложным путем – путем поиска математического решения и чисто техническим путем – путем имитации рекурсии.

Сложный путь. Вспомним задачу Дейкстры. Она имеет красивое нерекурсивное решение, основанное на зависимости между количествами четных и нечетных аргументов. Если вас заинтересует детальный разбор этого решения, то вы можете посмотреть его в [4]. Понимание такой закономерности не требует ничего за пределами элементарной математики, но это не означает того, что такую закономерность легко найти. В любом случае время, потраченное на поиск, существенно превышает время, необходимое на разработку рекурсивного варианта. Сложный путь возможен достаточно часто, но его сложность – пожалуй, даже не самый главный недостаток. К сожалению, принимая решение о движении по сложному пути, мы не имеем никакой гарантии, что у пути будет конец за разумное время, или даже хуже, есть ли вообще красивое математическое решение. Его может и не быть. Однако время от времени пытаться пройти «путем воина» надо. Во-первых, это надо для себя, для тренировки своего интеллекта. Во-вторых, одно красивое решение стоит множества достаточно эффективных.

Технический путь. Этот путь предполагает, что известно достаточно производительное рекурсивное решение и от программиста требуется только моделирование рекурсии другими средствами. Этих средств два:

Во-первых. Рекурсия – это многократное выполнение одних и тех же операций над данными. Следовательно, рекурсия моделируется циклом.

Во-вторых. Каждая копия данных для каждой активации процедуры сохраняется в стеке. Стек – это структура данных с некоторыми особенностями, и для её реализации нужны специальные усилия, программист должен создать модель стека на основе либо массива, либо связанного списка.

Из второго пункта видно, что моделирование рекурсии требует затрат памяти. Возникает естественный вопрос: стоит ли игра свеч, если моделирование рекурсивного процесса необходимо для экономии ресурсов, и при этом оно же их и потребляет. Это вопрос оптимизации. При активации рекурсивной процедуры в стеке автоматически сохраняются все локальные переменные, объявленные в процедуре, программист же, взявшийся моделировать рекурсивный процесс, может управлять объемом сохраняемых данных. Так что выгода целиком определяется умениями конкретного программиста.

Вопрос перехода от рекурсивного к нерекурсивному варианту очень важен и сам по себе, для более качественного понимания рекурсивных решений, поэтому не поленимся и разберем пару примеров. Для первого примера возьмем наш уже решенный факториал. Повторимся только, что пример хорош только как учебный.

Листинг 5.4

```
PROCEDURE Факториал*;
VAR
  N,k:INTEGER;
  Результат: ARRAY 100 OF INTEGER;
BEGIN
  In.Open;
  In.Int(N);
  FOR k:=N TO 1 BY -1 DO
    Результат[k]:=k;
  END;
  FOR k:=2 TO N DO
    Результат[k]:= Результат[k]* Результат[k-1];
  END;
  StdLog.Int(Результат[N]);
END Факториал;
```

Первый цикл моделирует спуск к тривиальному случаю. Второй цикл – это цикл подъема от случая 1! вверх. На каждом шагу элемент массива – результат с индексом -1 – считается уже посчитанным значением факториала, и это значение умножается на очередной множитель. Как видите, программа удвоилась в размере. Но свою задачу моделирования мы выполнили.

Следующая попытка нерекурсивной реализации рекурсии – это задача Дейкстры. Здесь мы имеем ветвящийся процесс, поэтому реализация будет немно-

го сложнее. Сделаем так: пусть массив используется для хранения аргументов. В момент запуска процесса запишем в нулевой элемент исходное число. Затем на каждом шаге выполним преобразование верхнего элемента массива по следующим правилам:

1. Если элемент массива равен единице, то значение функции (в программе переменная **sum**) увеличим на 1 и длину массива уменьшим на 1.
2. Если элемент массива четен, то длина массива остается без изменений, но верхний элемент уменьшается вдвое.
3. Если элемент нечетен, то он по формуле из условия разбивается на два. Один из новых аргументов записывается вместо текущего верхнего элемента, а для следующего нового массив удлинняется на единицу.

Процесс завершается, когда индекс, пробегающий по массиву-стеку, становится равен -1. Этот факт означает, что стек пуст, в терминах рекурсии активных вызовов больше нет.

Листинг 5.5

```
PROCEDURE ЗадачаДейкстры*;
VAR
  Стек:ARRAY 100 OF INTEGER;
  Аргумент,Сумма,k:INTEGER;
BEGIN
  In.Open;
  In.Int(Аргумент);
  k:=0;
  Стек[k]:=Аргумент;
  Сумма:=0;
  WHILE k>=0 DO
    IF Стек[k]=1 THEN
      Сумма:=Сумма+1;
      k:=k-1;
    ELSIF Стек[k] MOD 2=0 THEN
      Стек[k]:= Стек[k] DIV 2;
    ELSE
      Стек[k+1]:=(Стек[k] DIV 2)+1;
      Стек[k]:=Стек[k] DIV 2;
      k:=k+1;
    END;
  END;
  StdLog.Int(Сумма);
END ЗадачаДейкстры;
```

Конечно описанное решение – не вполне точное описание того, что происходит в рекурсивном процессе. Но весьма близко, да и точное совпадение не нужно.

Необходимо решение, а его мы получили. Основные свойства рекурсии налицо – циклический процесс и реализация стека.

Мы показали примеры нерекурсивной реализации рекурсивных алгоритмов. Идея во всех реализациях одна и та же: необходимо организовать циклический процесс по управлению стеком. Этот цикл может быть простым, как в случае с расчетом факториала, может быть довольно сложным, но это уже детали алгоритмизации, стержень же – четкое понимание того, как выполняется управление стеком.

Еще один хороший пример пары рекурсивное/нерекурсивное решение дан в главе «Комбинаторные задачи», это задача о построении перестановок. Достаточно содержательная задача, проведите анализ этой пары решений с точки зрения перехода от рекурсивного решения к модели рекурсии.

Рекурсия с возвратом и без. В этом пункте мы будем рассматривать только вычислительные алгоритмы, то есть алгоритмы, целью которых является расчет некоторой рекуррентно определенной числовой величины. Мы рассмотрели два примера таких алгоритмов: счет факториала и задача Дейкстры. Несмотря на различную природу алгоритмов и различный уровень сложности, у них есть одна общая черта. А именно окончательное решение о считаемой величине принимает первая активация рекурсивной процедуры, но ей для принятия решения не хватает данных, и она инициирует рекурсивный процесс для сбора недостающей информации. Для факториала процесс линейный, в задаче Дейкстры процесс представлен деревом, но суть одна. Необходимо идти вглубь, до тривиального случая, и когда он будет достигнут, что-то явным образом посчитать и затем, сворачивая дерево или последовательность активаций, через промежуточные вычисления результат вернуть наверх.

Такое построение рекурсии – не единственно возможное. Существует подход, работающий в обратном порядке. А именно решение о посчитанной величине принимается в самой глубокой активации. Процесс начинается с тривиального случая и идет вглубь, ведя расчеты с нарастающим итогом. В качестве первого примера возьмем опять наш старый добрый факториал.

Листинг 5.6

```
MODULE Модуль;  
IMPORT StdLog,In;  
PROCEDURE Главная*;  
VAR  
  Аргумент:INTEGER;  
PROCEDURE Факториал(НомерВызова,Результат,Аргумент:INTEGER);  
BEGIN  
  IF НомерВызова=Аргумент THEN  
    StdLog.Int(Результат);  
  ELSE  
    НомерВызова:= НомерВызова +1;
```

```

Результат:=Результат*НомерВызова;
Факториал(НомерВызова,Результат,Аргумент);
END;
END Факториал;
BEGIN
  In.Open;
  In.Int(Аргумент);
  Факториал(1,1,Аргумент);
END Главная;
END Модуль.

```

Как видно из текста, первая активация не принимает решения о значении факториала, она запускает процесс, передавая вглубь информацию о шаге счета, уже вычисленном факториале и значении аргумента, от которого факториал должен быть посчитан. Как всегда, не будем ограничивать исследование вопроса одним факториалом. Построим в качестве следующего примера процедуру рекурсивного расчета чисел Фибоначи. Если точнее, вычислим все числа, не превышающие некоторое заданное число. Ряд Фибоначи задается рекуррентным равенством.

$$a_n = a_{n-1} + a_{n-2}.$$

Из формулировки задачи ясно, что на первой активации принять решение о вычисленном числе сложно, так как неизвестен номер этого числа в ряду. Поэтому здесь рекурсия без возврата будет выглядеть естественно. Первые два числа известны: $a_1 = 1$ и $a_2 = 1$. С них и начнет свою работу первая активация. Затем на каждом шагу два очередных числа будем передавать следующей активации. Процесс завершится активацией, которая обнаружит число, уже большее заданного. Листинг смотрим чуть ниже:

Листинг 5.7

```

MODULE Модуль;
IMPORT StdLog,In;
PROCEDURE Главная*;
VAR
  Аргумент:INTEGER;
PROCEDURE ЧислаФибоначи(Первое,Второе,Аргумент:INTEGER);
BEGIN
  IF Второе>Аргумент THEN
    StdLog.Int(Второе);
  ELSE
    Первое:=Первое+Второе;
    ЧислаФибоначи(Второе,Первое,Аргумент);
  END;
END ЧислаФибоначи;
BEGIN
  In.Open;

```

```
In.Int(Аргумент);  
ЧислаФиббоначи(1,1,Аргумент);  
END Главная;  
END Модуль.
```

Пример ряда Фиббоначи – это пример, в котором рекурсия без возврата, возможно, более прозрачна, чем рекурсия с возвратом.

Последнее замечание о рекурсии без возврата. Несмотря на свое название, возврат все же происходит. Как бы процедура ни работала, но все активации должны завершить свою работу в установленном порядке. Термин «без возврата» означает только то, что нет возвращаемого значения. Рекурсивные вызовы сворачиваются как бы в холостую. А сейчас взглянем на рекурсию еще под одним полезным углом зрения.

Рекурсия как метод поиска

Далее будем считать линейные процессы неинтересными. Действительно, любая линейная задача легко реализуется последовательностью итераций. Рекурсия становится весьма полезной при ветвящихся процессах. Ветвящемуся процессу соответствует структура данных дерева, в вершинах которого могут находиться данные самой различной природы. С построением такого дерева можно связать два вида задач. Например, найти вершину, обладающую неким свойством, или, что более интересно, найти путь по дереву, обладающий неким свойством. Именно такую природу имеет следующая задача.

Условие задачи. Дана доска размером $N \times N$ клеток. В одном из её углов находится конь. Найти все пути коня, отвечающие следующим условиям:

- путь проходит через все поля доски ровно по одному разу;
- путь завершается на том поле, с которого конь вышел.

Принципиально задача имеет простое переборное решение. Пронумеруем все поля доски числами от 1 до N^2 . Далее построим все возможные перестановки номеров и для каждой перестановки проверим, является ли она путем коня и удовлетворяет ли этот путь озвученным свойствам. Написать программу, реализующую эту идею, совершенно не сложно, но бессмысленно. Действительно, пусть в нашем распоряжении стандартная шахматная доска 8×8 . Она состоит из 64 клеток. Пусть конь стоит на поле, чей номер – единица. Тогда остается рассмотреть перестановки 63 номеров. Количество перестановок равно астрономическому числу $63! = 2 \cdot 3 \cdot \dots \cdot 63$. Программа будет работать, но решения за разумное время мы не увидим.

Подойдем к задаче немного с другой стороны. Пусть какой-то путь уже пройден, и конь находится на некотором поле с номером L . Это означает, что исходную задачу с полем, чей номер – единица, мы заменили на задачу поиска пути, начинающегося с поля с большим номером. Номер больше, следовательно, искомый путь короче, но задача та же – поиск пути. Таким образом, обнаружен рекуррентный

характер задачи. Рекурсивный процесс строит все возможные варианты движения, среди которых остается только найти путь, заканчивающийся на том же поле, с которого конь вышел.

Опишем процесс построения. Для этого достаточно понять, что происходит на очередном поле. Предположим, на некотором шаге конь попал на поле A . С этого поля у него есть ряд возможностей для хода, которые мы обозначим как A_1, A_2, \dots, A_n . Из этого списка необходимо вычеркнуть поля, через которые строящийся путь уже прошел, этого требует условие задачи. После чего для каждой из оставшихся возможностей необходимо активировать процедуру дальнейшего построения пути. Для новых активаций каждое из полей станет исходной точкой. Вернемся на поле A . Возможно, некоторые из продолжений, активированных с этого поля, закончатся тупиком. В этом случае конь, отработав продолжение, опять попадет на поле A . В этом случае поле A_k , давшее тупиковое продолжение, необходимо из списка вычеркнуть.

Такой процесс дает возможность избежать полного перебора. Действительно, каждое поле, попавшее на путь коня, генерирует в качестве своего возможного продолжения не все оставшиеся на доске поля, а только те, которые с этого поля достижимы ходом коня и удовлетворяют условию непересечения путей. Задача, таким образом, становится решаемой для досок значительного размера. А рекурсивное решение приобретает некоторые дополнительные свойства и превращается в метод динамического программирования. Чуть позже мы вернемся к этой задаче и доведем её до полного решения.

Динамическое программирование

Необходимо понимать, что рекурсия – это механизм, обеспеченный возможностями языка и возможностями компилятора. Нетрудно предположить, что этот механизм допускает обобщение до метода мышления, не сводимого к свойствам языка программирования. Таким методом является метод динамического программирования.

Формулировка. Пусть решается некая задача, требующая определенного размера вычислений. Пусть данная задача представима как сумма задач меньшего вычислительного объема. Тогда решение исходной задачи может быть получено решением задач меньшего объема и их «суммированием».

Естественно, термин «суммирование» нельзя понимать в арифметическом смысле. Возможность суммирования означает, что подзадачи независимы, в том смысле, что решение одной из них не влияет на решение другой. Рекурсивное определение задачи через подзадачи есть частный случай динамической задачи. Действительно, динамическое программирование не требует одинаковой природы подзадач. И есть еще одна важная корректива.

Перекрывающиеся подзадачи. Простая рекурсия зачастую требует многократного вычисления одних и тех же величин. В качестве иллюстрации можно взять рекурсивный расчет чисел Фибоначчи и задачу Дейкстры. Выше был по-

строен алгоритм счета ряда Фибоначи рекурсией без возврата. Функция, записанная ниже, делает то же самое, но с возвратом:

Листинг 5.8

```
PROCEDURE Фибоначи(Аргумент:INTEGER):INTEGER;  
BEGIN  
  IF (Аргумент=2) OR (Аргумент=1) THEN  
    RETURN 1;  
  ELSE  
    RETURN Фибоначи(Аргумент-1)+Фибоначи(Аргумент-2);  
  END;  
END Фибоначи;
```

Реализация с возвратом выглядит немного короче, но здесь есть существенная проблема. Предположим, считается **Фибоначи**(5):

Фибоначи(5) = **Фибоначи**(3) + **Фибоначи**(2).

Далее расчет **Фибоначи**(3) дает следующую сумму:

Фибоначи(3) = **Фибоначи**(2) + **Фибоначи**(1).

Таким образом, функция **Фибоначи**(2) будет считаться дважды: один раз её вызов появляется в попытке посчитать **Фибоначи**(5) и второй раз её вызов появляется в расчете **Фибоначи**(3). Естественно, при расчете чисел с большими номерами приведет к большому количеству повторов **Фибоначи**(2), кроме того, появятся повторы и других аргументов. Количество повторов начнет расти лавинообразно. Такого рода повторы называются перекрывающимися задачами.

Второй простой пример перекрывающихся задач это рекурсивное дерево задачи Дейкстры. Покажем пример расчета **ФункцияДейкстры**(7):

ФункцияДейкстры(7) = **ФункцияДейкстры**(4) + **ФункцияДейкстры**(3) =
ФункцияДейкстры(2) + **ФункцияДейкстры**(2) + **ФункцияДейкстры**(1).

Повтор задачи расчета **ФункцияДейкстры**(2) очевиден.

Преимущество динамического программирования перед простой рекурсией – именно в выявлении перекрывающихся задач. Борьба с перекрытием сводится к организации специальной памяти – кэша, в котором хранятся все уже посчитанные подзадачи до того момента, пока в них не отпадет необходимость. Конечно, кэширование не только предлагает решение, но и создает новые проблемы:

- не всегда легко выявить критерий необходимости данной подзадачи или, наоборот, отсутствия необходимости;
- размер кэша. Память безгранична. Поэтому необходимо следить за размещением новых подзадач.

Преимущества очевидны:

- серьезное ускорение вычислительного процесса;
- возможная экономия памяти. Этот пункт не очевиден. С одной стороны, нужно тратить память на кэш, но, с другой стороны, кэширование задач

может привести к существенному укорачиванию рекурсивного процесса, а если мы вспомним, что каждая активация рекурсивной процедуры требует памяти в стеке, то ясно, что короткая рекурсия – это всегда экономия памяти. Что будет существеннее, размер кэша или полученная экономия, – большой вопрос, решаемый только для конкретно взятой задачи.

Два вида динамического программирования. Вернемся к задаче обхода доски конем. Это, очевидно, задача динамического программирования. Если доска 8×8 , то искомым путь имеет длину 63 хода (без учета первого поля). Если первый ход сделан, то задача сводится к задаче поиска пути длиной 62 хода и т. д. То есть задача с каждым ходом сводится к задаче поиска пути длины на 1 короче. Здесь мы имеем спуск к простым задачам. Такая стратегия динамического программирования называется нисходящим динамическим программированием.

Обратный метод, естественно, называется восходящим программированием. При восходящем программировании работа начинается с определения полного набора перекрывающихся задач, и исходная задача конструируется из них. Простой пример восходящего динамического программирования – это рекурсивный счет чисел Фибоначчи без возврата. В этом варианте исходными, уже решенными задачами являются задачи **Фибоначчи(1)** и **Фибоначчи(2)**. Из них конструируется задача **Фибоначчи(3)** и т. д. Восходящее программирование более выгодно в плане экономии кэша. Но в общем-то, выбор между нисходящим и восходящим программированием должен выполняться, исходя из природы задачи.

Рассмотрение вопросов применения рекурсии и динамического программирования завершим двумя сложными примерами, в качестве одного из которых возьмем незавершенную задачу об обходе конем шахматной доски.

Задача обхода конем шахматной доски

Еще раз укажем, что это задача динамического программирования. Действительно, пусть уже известен путь в L шагов на доске размером N^2 . Тогда задача построения пути длиной в N^2 шагов сводится к задаче построения пути в $N^2 - L$ шагов, то есть к задаче меньшего объема. Далее от каждого поля возможно построение нескольких путей, следовательно, для этих путей задача построения пути до поля является перекрывающейся.

Немного уточним формулировку. Исходная формулировка, конечно же, радикальному изменению не подлежит, мы просто дадим более функциональное толкование того, что требуется сделать. На доске размером $N \times N$ необходимо построить все пути со следующими свойствами:

- каждый путь строится за $N^2 - 1$ ходов коня;
- путь проходит через все поля доски;
- путь завершается на том поле, на котором конь стоял изначально.

Нетрудно заметить, что данная формулировка задачи равнозначна исходной. Но здесь указана ключевая конструкция – путь, построение которого и означает решение задачи. Ядром программы должна стать рекурсивная процедура, продол-

жающая построение пути от текущего поля или возвращающая коня на предыдущее поле, если все пути с текущего поля уже отработаны. Очевидно, что входное данное для такой процедуры необходимо одно – поле (две координаты), на которое должен пойти конь в следующей активации. Получив координаты поля, процедура должна выполнить ход, определить список возможных продолжений и для каждого продолжения выполнить очередную активацию. Завершение активаций происходит по отработке всех продолжений и заключается в возврате коня на предыдущее поле. Программно возврат хода – это передача управления предыдущей активации, которая владеет информацией о предыдущем ходе.

Договоримся об используемых структурах данных. Роль модели шахматной доски поручим двумерному массиву. Пустая доска – это массив, инициализированный нулями. Путь коня будем отмечать ненулевыми числами. Каждая активация, очевидно, должна хранить информацию о возможных продолжениях. Таковую информацию можно ограничить координатами полей и длиной уже построенного пути. Напишем текст процедуры в первом приближении. Назовем её процедурой хода. На вход процедура получает следующие параметры:

- **Длина** – длина уже построенного пути;
- **X, Y** – координаты поля, на которое необходимо выполнить ход.

IF **Длина** < N*N THEN

 Выполнить ход

 Рассчитать возможные продолжения

Длина := **Длина** + 1

 Номер = 0

 WHILE Номер < Номера последнего продолжения DO

 Активировать рекурсивную процедуру с параметрами X, Y

 END;

END;

Фразы, записанные на русском языке, – это последовательности операторов, которые еще необходимо построить. Первая операция «Выполнить ход» реализуется одним оператором: **Доска[X,Y]:=Число**;, активация рекурсивной процедуры – тоже пункт не слишком интересный: **Ход(X,Y,Длина)**;. Расчет возможных продолжений уже не столь тривиален. Этим расчетом придется заняться детально.

Расчет возможных продолжений

Договоримся записывать посчитанные продолжения в массив записей вида:

ТипХод = RECORD

 X,Y:INTEGER;

END;

Массив, возможно, определить так: **Ход:ARRAY 10 OF ТипХод**;. Кстати, запись фактических параметров в вызове процедуры придется немного уточнить. Заметим, что поле имеет ограниченные размеры, поэтому в зависимости от положения коня

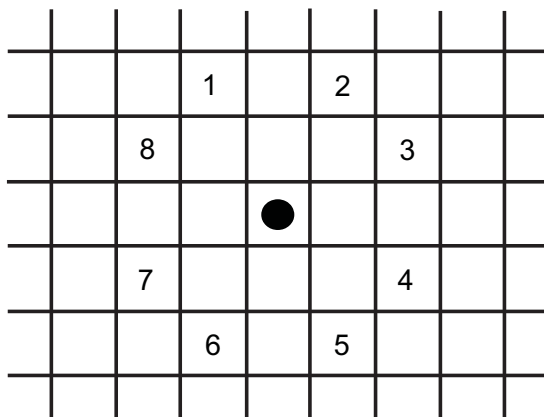


Рис. 5.4

часть ходов, возможно, придется отбросить.

Лишние ходы – это ходы за доску. Сейчас решим элементарную арифметическую задачу. Конь стоит на некотором поле с координатами X, Y . Необходимо определить координаты полей, на которые он может сделать ход. Всего возможных ходов – 8. Пусть начало системы координат в текущей позиции коня. Тогда:

Поле 1: $X' = X - 1; Y' = Y + 2$.

Поле 2: $X' = X + 1; Y' = Y + 2$.

Поле 3: $X' = X + 2; Y' = Y + 1$.

Поле 4: $X' = X + 2; Y' = Y - 1$.

Поле 5: $X' = X + 1; Y' = Y - 2$.

Поле 6: $X' = X - 1; Y' = Y - 2$.

Поле 7: $X' = X - 2; Y' = Y - 1$.

Поле 8: $X' = X - 2; Y' = Y + 1$.

Естественно, каждое из полей попадает в массив вариантов, только если его координаты не выходят за границы доски. Если доска имеет размер $N \times N$, то нижняя граница 0 и верхняя $N-1$. Запишем фрагмент программы, формирующий массив вариантов продолжений:

(*Фрагмент обработки первого поля*)

k:=-1;

IF (X-1>=0) & (Y+2<N) THEN

k:=k+1;

Ход[k].X:=X-1;

Ход[k].Y:=Y+2;

END;

(*Фрагмент обработки второго поля*)

IF (X+1<N) & (Y+2<N) THEN

k:=k+1;

```

Ход[k].X:=X+1;
Ход[k].Y:=Y+2;
END;

```

Записывать совершенно аналогичные фрагменты для всех восьми полей не будем. Отметим только, что массив вариантов в конечном итоге может состоять из меньшего количества полей (не восьми). Этот фрагмент можно «оптимизировать». Заметим, что величина изменения обеих координат пробегает строго четыре значения: -2, -1, 1, 2. Это означает, что перебор возможных изменений можно организовать в виде двух вложенных циклов. Но термин «оптимизировать» не зря взят в кавычки. Текст такой оптимизацией действительно можно сделать короче, но одновременно и сложнее для восприятия. Кроме того, очень сомнительно, что оптимизированный вариант будет работать хотя бы быстрее. В этой ситуации вопрос оптимизации сводится к выбору между краткостью текста и его читаемостью. Сейчас уже можно записать главный расчетный цикл:

```

num := 0;
WHILE num < k DO
  ПостроениеПути(Ход[num], Длина);
  num:=num + 1;
END;

```

Таким образом, как разворачивается рекурсивный процесс, уже достаточно понятно. Но в нашем варианте он никогда не завершается. Чего не хватает? Не хватает описания тупиковых ситуаций, то есть ситуаций, в которых нет продолжений. Тупики могут быть двух сортов: тупик – решение и тупик – отсутствие хода. Тупик, являющийся, решением совпадает с ситуацией **Длина=N²**. Эту ситуацию удобно обрабатывать в главном условии:

```

IF Длина< N*N THEN
  (*Развертывание рекурсии*)
ELSE
  (*Распечатка доски*)
  FOR i:=0 TO N-1 DO
    FOR j:=0 TO N-1 DO
      StdLog.Int(Доска[i,j]);
    END;
    StdLog.Ln;
  END;
END;

```

Текст, обозначенный комментарием (*Развертывание рекурсии*), уже разобран вполне подробно. Остались лишь небольшие технические детали. Заботиться специально об обработке тупика нет необходимости. Тупик означает просто отсутствие продолжений, а это по смыслу то же самое, как и завершение обработки существующих решений. Поэтому тупик – это просто завершение работы главного

цикла рекурсивных вызовов, и только. Единственное, что забыто, – это обработка самопересечения пути коня. Напомним, что конь на каждом поле может побывать только один раз.

Означенную проблему можно решить в двух вариантах. Можно при расчете допустимых продолжений, если поле находится в пределах доски, осуществлять дополнительную проверку, был конь на поле или нет (оно содержит нулевое или неотрицательное значение). Вот так:

```
IF (X-1>=0) & (Y+2<=N-1) & (Доска[X-1,Y+2]=0) THEN
  k:=k+1;
  Ход[k].X:=X-1;
  Ход[k].Y:=Y+2;
END;
```

Всего лишь восемь дополнительных логических выражений. А можно изменить главный цикл. Вот так:

```
num := 0;
WHILE num < k DO
  IF Доска[Ход[num].X,Ход[num].Y]=0 THEN
    Recurse(Ход[num], Длина);
  END;
  num:=num + 1;
END;
```

Преимуществ у этих двух методов, наверное, никаких нет, и что выбрать, вопрос вкуса. Вторая техническая особенность – это отмена хода после завершения текущей активации. Выполнение хода сводится к оператору **Доска**[X,Y]:=Число;. Следовательно, отмена хода – это оператор **Доска**[X,Y]:=0;. Единственное – необходимо уточнение величин X,Y. Рекурсивная процедура получает на вход, как уже было решено, не координаты X, Y по отдельности, а структуру данных хода. Обозначим получаемую структуру данных как **ВычисленныйХод**. Тогда операторы хода и отмены хода будут выглядеть следующим образом:

Выполнение хода: **Доска**[**ВычисленныйХод**.X, **ВычисленныйХод**.Y]:=Число;

Отмена хода: **Доска**[**ВычисленныйХод**.X, **ВычисленныйХод**.Y]:=0;

Все механизмы и технические детали разобраны полностью, соберем текст программы в единое целое:

Листинг 5.9

```
MODULE Модуль;
IMPORT StdLog,In;
TYPE
  ТипХод=RECORD
    X,Y:INTEGER;
  END;
```

VAR

Доска: ARRAY 100, 100 OF INTEGER;

N:INTEGER;

PROCEDURE **ПостроениеПути**(**ВычисленныйХод**:ТипХод; **Длина**:INTEGER);

VAR

num,k,i,j,X,Y:INTEGER;

Ход:ARRAY 8 OF ТипХод;

BEGIN

X:= **ВычисленныйХод**.X;Y:= **ВычисленныйХод**.Y;

Доска[X,Y]:=Длина;

IF **Длина**<N*N THEN

k:=-1;

(*Первое поле*)

IF (X-1>=0) & (Y+2<N) THEN

k:=k+1;

Ход[k].X:=X-1;

Ход[k].Y:=Y+2;

END;

(*Второе поле*)

IF (X+1<N) & (Y+2<N) THEN

k:=k+1;

Ход[k].X:=X+1;

Ход[k].Y:=Y+2;

END;

(*Третье поле*)

IF (X+2<N) & (Y+1<N) THEN

k:=k+1;

Ход[k].X:=X+2;

Ход[k].Y:=Y+1;

END;

(*Четвертое поле*)

IF (X+2<N) & (Y-1>=0) THEN

k:=k+1;

Ход[k].X:=X+2;

Ход[k].Y:=Y-1;

END;

(*Пятое поле*)

IF (X+1<N) & (Y-2>=0) THEN

k:=k+1;

Ход[k].X:=X+1;

Ход[k].Y:=Y-2;

END;

(*Шестое поле*)

```

IF (X-1>=0) & (Y-2>=0) THEN
  k:=k+1;
  Ход[k].X:=X-1;
  Ход[k].Y:=Y-2;
END;
(*Седьмое поле*)
IF (X-2>=0) & (Y-1>=0) THEN
  k:=k+1;
  Ход[k].X:=X-2;
  Ход[k].Y:=Y-1;
END;
(*Восьмое поле*)
IF (X-2>=0) & (Y+1<N) THEN
  k:=k+1;
  Ход[k].X:=X-2;
  Ход[k].Y:=Y+1;
END;
num := 0;
WHILE num <= k DO
  IF Доска[Ход[num].X, Ход[num].Y]=0 THEN
    ПостроениеПути(Ход[num], Длина+1);
  END;
  num:=num + 1;
END;
ELSE
  IF (X=0) & (Y=0) THEN
    StdLog.Ln;
    FOR i:=0 TO N-1 DO
      FOR j:=0 TO N-1 DO
        StdLog.Int(Доска[i,j]);
      END;
      StdLog.Ln;
    END;
    StdLog.Ln;
  END;
END;
Доска[X,Y]:=0;
END ПостроениеПути;
PROCEDURE Главная*;
VAR
  Ход: ТипХод;
BEGIN
  In.Open;
  In.Int(N);

```



```

Ход.Х:=0; Ход.У:=0;
ПостроениеПути(Ход,0);
END Главная;
END Модуль.

```

Несколько дополнительных замечаний. Оператор **Доска**[X,Y]:=0; в конце текста рекурсивной процедуры нужен для отмены хода, обработанного процедурой. Перед распечаткой решения выполняется проверка равенства нулю координат конечного хода. Это необходимо для учета того обстоятельства, что путь длины N^2 ходов не обязательно закончится на том поле, с которого конь начал свое движение.

Задание для самостоятельной работы

Цикл обработки массива ходов и сам массив не обязательны. Можно построение хода, проверку допустимости и рекурсивный вызов объединить, вот так:

```

(*Первое поле*)
IF (X-1>=0) & (Y+2<N) & (Доска[X-1,Y+2]=0) THEN
  ПостроениеПути(X-1, Y+2);
END;

```

Но тогда, конечно, вся программа нуждается в небольшой перестройке. Попробуйте этот вариант реализовать самостоятельно.

Задание для самостоятельной работы

Вопрос, каким числом отмечать поле, включенное в путь, кажется несущественным. В нашей реализации поле отмечается числом, равным длине пути. Попробуйте заменить оператор **Доска**[X,Y]:=Длина; на оператор **Доска**[X,Y]:=1;. Принципиально это ничего не значит, но программа зависнет. В чем же здесь разница?

Факторизация числа

В заключение разберем алгоритм факторизации, выполняющий свою работу не за счет поиска делителей, а путем их построения. Этот алгоритм не годится для работы в боевых условиях (для больших чисел), но как пример рекурсивно решаемой задачи он хорош.

Реализация не вполне тривиальна, а идея очень проста. Предположим, что некоторое число C является произведением двух множителей: $C = AB$. Пусть C задано массивом своих цифр и имеет длину N . Тогда, перебрав все числа A и B с длинами $0...N$ и перемножив их, мы найдем ту пару, чье произведение равно A (нулевая длина – это число из одной цифры).

Полный перебор организуем процедурой достройки множителей. Делается это так. Пусть наши множители состоят только из одной цифры. Тогда необходимо перебрать все возможные цифры для каждого множителя и выбрать из всего множества те пары, произведение которых дает первую цифру – такую же, как и первая цифра факторизируемого числа. Например, для числа 13452 для начала процесса годятся следующие пары множителей (3, 4), (1, 2), (6, 2), (6, 7) и, конечно, им сим-

метричные. Это значительно меньше количества всех возможных пар цифр. Каждая такая пара порождает продолжение процесса. А именно для каждого из множителей можно добавить один разряд, перебрать для нового разряда все цифры и выяснить, для какой цифры множителя «встает на свое место» очередная цифра произведения. Если такая цифра найдена, то процесс достройки произведения до факторизируемого числа продолжается. Таким образом, рекурсивная процедура выполняет следующую работу:

1. Удлиняет множитель на один разряд.
2. Перебирает все цифры для этого разряда.
3. Для каждой цифры пересчитывает произведение с учетом новой цифры, и если произведение совпадает с определенной частью факторизируемого числа, то продолжает достройку множителей.
4. Достраивать множители продолжает новая активация, получающая информацию об уже полученном произведении, сформированных множителях и о множителе, который необходимо удлинить на один разряд.

Это, так сказать, костяк идеи. Здесь пока много неясного. Например, что значит «пересчитать произведение с учетом новой цифры»? С какой именно частью факторизируемого числа необходимо проверять совпадение? Прежде чем заняться реализацией алгоритма, на эти вопросы надо дать ответы. Ответ на второй вопрос вам придется поискать самостоятельно. В реализации, которая будет построена ниже, отработана схема рекурсивного процесса без этого важного ограничителя.

А операцию пересчета произведения поясним. Арифметической основой для нашего алгоритма является метод умножения чисел в столбик. Но здесь ситуация не вполне обычная. Множители изменяют свои цифры попеременно, это вынуждает разработать небольшую модификацию обычного столбикового умножения. Рассмотрим, как это делается, на примере: $345 * 231$.

Шаг 1. Выполним умножение первых разрядов: $5 * 1 = 5$. Это первое произведение. Множители содержат по одной цифре.

Шаг 2. Добавим к первому множителю один разряд: $45 * 1$. Умножим новую цифру на второй множитель и прибавим к произведению со сдвигом на один разряд: $40 + 5 = 45$. То есть $45 * 1 = 45$.

Шаг 3. Добавим ко второму множителю один разряд $45 * 31$. Умножим новую цифру на первый множитель и добавим к произведению со сдвигом на один разряд: $1350 + 45 = 1395$.

Шаг 4. Добавим к первому множителю один разряд $345 * 31$. Умножим новую цифру на второй множитель и добавим к произведению со сдвигом на два разряда: $9300 + 1395 = 10695$.

Шаг 5. Добавим ко второму множителю один разряд $345 * 231$. Умножим новую цифру на первый множитель и добавим к произведению со сдвигом на два разряда: $69000 + 10695 = 79695$.

Умножение выполнено. Результат совпадает с результатом обыкновенного умножения столбиком. Но процедура заметно иная. С арифметической осно-

вой закончено, приступим к реализации. Во-первых, необходимо договориться о структуре данных. Все числа, участвующие в процессе, представлены массивами своих цифр, но необходима дополнительная информация о длине каждого числа, поэтому тип данных определим так:

```
TYPE
  Число=RECORD
    Длина:INTEGER;
    Цифра:ARRAY 100 OF INTEGER;
  END;
```

Главная процедура начинается со вспомогательной работы. Она открывает поток, вводит цифры факторизируемого числа, определяет его длину и переворачивает число так, чтобы младшие разряды находились в младших элементах массива, это удобно для программирования арифметических операций:

```
(*Ввод числа*)
In.Open;
k:=-1;
REPEAT
  k:=k+1;
  In.Int(Фактор.Цифра[k]);
UNTIL ~In.Done;
(*Переворачивание числа*)
Фактор.Длина:=k-1;
FOR k:=0 TO Фактор.Длина DIV 2 DO
  c:= Фактор.Цифра[k];
  Фактор.Цифра[k]:= Фактор.Цифра[Фактор.Длина-k];
  Фактор.Цифра[Фактор.Длина-k]:=c;
END;
```

Сейчас уже можно запускать рекурсивный процесс, но для начала необходимо определить явным, нерекурсивным образом по одной цифре для каждого множителя. Почему поиск первых цифр нельзя поручать рекурсивному процессу? Каждая рекурсивная активация будет удлинять очередной множитель на одну цифру, после чего новую цифру надо умножить описанным выше способом на второй множитель. Следовательно, второй множитель должен быть определен, хотя бы на одну цифру уже перед первой активацией. Поэтому по одной цифре и нужно определить до основного процесса.

Начинается определение множителей с определения их длины, помним, что длина числа есть компонент структуры **Число**. Затем определяем первую цифру для обоих множителей, для чего перебираем цифры от 0 до 9 для первого и для каждой цифры первого множителя перебираем цифры от 0 до 9 для второго:

```
Множ1.Длина:=0;
Множ2.Длина:=0;
```

```

flag:=TRUE;
x:=0;
WHILE flag & (x<=9) DO
    Множ1.Цифра[0]:=x;
    y:=0;
    WHILE flag & (y<=9) DO
        Множ2.Цифра[0]:=y;
        y:=y+1;
    END;
    x:=x+1;
END;

```

Поясним функцию переменной **flag**. В теле двух вложенных циклов, перебирающих первые цифры, будет активироваться рекурсивный процесс по дальнейшему построению множителей. Этот процесс для какой-то пары цифры может стать успешным. В этом случае расчеты необходимо завершить. Именно для завершения величина **flag** и потребуется. Договоримся сейчас, что рекурсивная процедура в случае успеха своей работы вернет логическое значение, прерывающее выполнение обоих циклов. В теле цикла выполняется расчет цифр произведения. Если цифры множителей невелики, то расчет даст одну цифру в произведении, но возможны две. Поэтому расчет цифр выполняется следующим фрагментом:

```

Множ2.Цифра[0]:=y;
с:=x*y;
Произв.Цифра[0]:=с MOD 10;
Произв.Цифра[1]:=с DIV 10;

```

После чего определяется длина полученного произведения:

```

IF Произв.Цифра[1]>0 THEN
    Произв.Длина:=1;
ELSE
    Произв.Длина:=0;
END;

```

Возможно, факторизуемое число мало, тогда уже две первые цифры дадут результат. Для учета такой ситуации уже сейчас необходимо сравнить полученное произведение с факторизуемым числом:

```

с:=Сравнение(Произв, Множ1, Множ2);

```

Поговорим немного о функции **Сравнение()**. Её можно построить очень просто: если факторизуемое число равно текущему произведению, то пусть она возвращает истину, иначе ложь. Но на самом деле функция из операции сравнения может выжать больше информации. Заметим, что строящиеся множители только увеличивают свое значение. Действительно, с каждой активацией длина множителя увеличивается на 1, и, следовательно, множитель новой активации, по крайней

мере, не меньше. В теле процедуры, пытающейся достроить множитель, перебираются возможные цифры, если порядок цифр взять от 0 до 9, то и в пределах одной активации множитель только возрастает (и, естественно, произведение). Поэтому результатом сравнения могут быть три вывода: произведение пока меньше факторизуемого числа (результат 0), произведение равно числу (результат 1) и произведение уже больше факторизуемого числа (результат 2). Сказанное дает неплохой ограничитель расчетному процессу. Если функция **Сравнение()** возвращает 2, то процесс необходимо завершить.

Это в общем. Применительно к подбору первой цифры ситуацию **Сравнение()=2**, наверное, можно игнорировать, так как факторизация одно- и двузначных чисел неинтересна, а если такое число и будет подано на вход, то небольшой перебор с количеством расчетов погоды не сделает.

Если **Сравнение()=1**, то расчеты однозначно необходимо прекратить, так как решение найдено. И только при **Сравнение()=0** необходимо активизировать рекурсивный процесс. Еще один способ оптимизации заключается в сравнении уже точно определенных цифр. Два однозначных множителя полностью определяют первую цифру произведения. Чего бы там в дальнейшем ни происходило, первая цифра останется неизменной. Следовательно, есть смысл активизировать рекурсивный процесс только в том случае, если первая цифра произведения совпадает с первой цифрой факторизуемого числа. Все сказанное записано в следующем листинге:

```
x:=0;
WHILE flag & (x<=9) DO
  Множ1.Цифра[0]:=x;
  y:=0;
  WHILE flag & (y<=9) DO
    Множ2.Цифра[0]:=y;
    c:=x*y;
    Произв.Цифра[0]:=c MOD 10;
    Произв.Цифра[1]:=c DIV 10;
    IF Произв.Цифра[1]>0 THEN
      Произв.Длина:=1;
    ELSE
      Произв.Длина:=0;
    END;
    c:=Сравнение(Произв, Множ1, Множ2);
    IF (c=0) & (Произв.Цифра[0]=Исходное.Цифра[0]) THEN
      flag:=-(ДостройкаМножителя (Множ1, Множ2, Произв)
        OR ДостройкаМножителя(Множ2, Множ1, Произв));
    ELSIF c=1 THEN flag:=FALSE;
  END;
  y:=y+1;
END;
```

```

x:=x+1;
END;

```

ДостройкаМножителя() и есть процедура, увеличивающая длину множителя на 1. Так как процедура удлинняет только один множитель, то пусть всегда это будет первый. Тогда необходимы две рекурсивные активации. Первая передает следующей активации в качестве первого множителя один из них, а вторая – другой. Если оба цикла (и по *x* и по *y*) отработают до конца и **flag** останется истинным – это будет означать отсутствие решения.

```

IF flag THEN StdLog.String('Решение не найдено');StdLog.Ln; END;

```

И последнее, по рассматриваемой процедуре. Факт совпадения факторизуемого числа и произведения обнаруживается в процедуре-функции **Сравнение()**. Пусть там и распечатываются множители. Возвращать их значения наверх не обязательно.

Перейдем к анализу и построению процедуры **ДостройкаМножителя()**. Во-первых, заметим, что каждая активация работает с текущим произведением и двумя множителями, уже построенными на предыдущем шаге, и делает с ними что-то свое. Поэтому, конечно, и произведение, и множители должны быть переданы процедуре, и целесообразно передать их копированием. Результат, как мы уже договорились, печатает функция, выполняющая работу сравнения. Заголовок уже можно записать:

```

PROCEDURE ДостройкаМножителя(Множ1,Множ2,Произв:Число):BOOLEAN;

```

Смысл работы процедуры – в удлинении первого множителя и попытке подобрать подходящую цифру. Поэтому первый оператор такой:

```

Множ1.Длина:= Множ1.Длина+1;

```

Далее цикл подбора новой цифры для первого множителя:

```

b:=Произв;
FOR x:=0 TO 9 DO
    Множ1.Цифра[Множ1.Длина]:=x;
    (*Пересчет произведения и следующие активации*)
END;

```

Оператор **b:=Произв**; совершенно необходим. Далее пересчитывается произведение для каждого *x*. И каждый раз оно пересчитывается от того произведения, которое было получено как фактический параметр. Поэтому изначальное значение необходимо запомнить в специальной переменной. Далее вспоминаем, чему равно полученное значение произведения, и выполняем его пересчет:

```

FOR x:=0 TO 9 DO
    Множ1.Цифра[Множ1.Длина]:=x;
    Произв:=b;
    FOR y:=0 TO Множ2.Длина DO

```

```

s:= Произв.Цифра[y+ Множ1.Длина]+ Множ2.Цифра[y]*x;
Произв.Цифра[y+ Множ1.Длина]:=s MOD 10;
Произв.Цифра[y+Множ1.Длина+1]:=Произв.Цифра[y+Множ1.Длина+1]
+(s DIV 10);

```

```
END;
```

```
END;
```

Схему пересчета пояснять не будем, этот технический прием разобран в начале параграфа. Следующим действием после пересчета необходимо определить, к чему мы пришли. Вспомним, что возможны три ситуации: факторизируемое число пока не получено (значение 0), мы достигли успеха (значение 1) или произведение уже больше факторизируемого числа (значение 2). Тип ситуации определяет процедура-функция **Сравнение()**, и её вызов сейчас и нужен, но прежде небольшая вспомогательная работа:

```

Произв.Длина:= Множ1.Длина+ Множ2.Длина+1;
WHILE (Произв.Длина>0) & (Произв.Цифра[Произв.Длина]=0) DO
  DEC(Произв.Длина);
END;
```

Сравнение() сравнивает два числа. Для этого ей необходимо знать длины. Длина текущего произведения не определяется только длинами множителей. Многое зависит еще от старших цифр множителей. А старшей цифрой вполне может быть нуль, более того, у обоих множителей может быть несколько нулей в старших цифрах. Естественно, эти нули переползут и в массив произведения. Записанный выше фрагмент делает длину произведения максимально возможной при заданных длинах множителей, а затем убирает старшие нули, после чего посчитанное произведение уже можно отдавать на обработку функции **Сравнение()**.

Функция возвращает один из трех вариантов: 0 – продолжаем процесс активаций для достройки множителей; 1 – прекращаем работу данной активации и возвращаем наверх сигнал о том, что работа завершилась успехом; 2 – прекращаем работу данной активации и сообщаем наверх, что работа ветки рекурсивных вызовов завершилась неуспехом (но это не означает полного неуспеха). Сказанное записано в следующем фрагменте:

```

c:=Сравнение(Произв, Множ1,Множ2);
IF (c=0) & (ДостройкаМножителя(Множ2, Множ1, Произв) OR
  ДостройкаМножителя(Множ1,Множ2,Произв)) THEN RETURN TRUE
END;
IF c=1 THEN RETURN TRUE END;
IF c=2 THEN RETURN FALSE END;
```

Обратите внимание на важный момент. При $c=0$ возврат истины происходит, если одна из активаций вернет истинное значение, но если обе активации вернут в этом условном операторе ложь, то ложь не вызывает дальнейшего возврата. Дело в том, что оператор возврата прерывает работу процедуры, чего делать здесь нельзя,

$c=0$ не означает неуспеха, это означает, что работу по поиску необходимо продолжить с другими цифрами. Полный провал своей деятельности активация может зафиксировать только по полному завершению цикла, перебирающего цифры, следовательно, если удалось достичь конца цикла, то тогда необходимо вернуть **ЛОЖЬ**, поэтому в конце текста процедуры записан оператор возврата

RETURN FALSE

Закончим построение реализации разбором процедуры **Сравнение()**. Её работа заключается в том, чтобы определить, какая из трех ситуаций имеет место: факторизуемое число еще не достигнуто и есть смысл продолжать работу (результат 0), число достигнуто, множители необходимо распечатать и работа завершена (результат 1), и факторизуемое число в этой ветке уже никогда не будет получено (результат 2).

Вывод о результате можно получить из разных условий. Например, если суммарная длина множителей уже больше, чем длина факторизуемого числа, то положительного результата достичь уже нельзя:

IF Множ1.Длина+Множ2.Длина> Фактор.Длина THEN RETURN 2; END;

Если длина произведения больше длины факторизуемого числа, то положительный результат опять недостижим:

IF Произв.Длина> Фактор.Длина THEN RETURN 2 END;

Если длина произведения меньше, чем длина факторизуемого числа, то о возможности результата говорить пока рано:

IF Произв.Длина< Фактор.Длина THEN RETURN 0 END;

И наконец, длины чисел (факторизуемого и произведения равны). Здесь уже пора заняться сравнением цифр. Сравнение можно выполнить так: начиная от старшего разряда пройдем по сравниваемым цифрам до тех пор, пока не встретится пара разных. Если такая пара не встретится, то, очевидно, сравниваемые числа равны, можно распечатать множители и вернуть 1. Если пара разных встретится, то больше то число, чья цифра больше. Если большая цифра принадлежит факторизуемому числу, то больше факторизуемое, в этом случае возвращаем 0, так как есть смысл продолжить построения, иначе такого смысла уже нет, и возвращаем 2. Пару разных поищем так:

k:= Фактор.Длина;

WHILE (k>=0) & (Произв.Цифра[k]= Фактор.Цифра[k]) DO k:=k-1;END;

Если по выходу из цикла $k \geq 0$ пара разных найдена, определяем тип ситуации и возвращаем соответствующее значение:

IF k>=0 THEN

IF Произв.Цифра[k]> Фактор.Цифра[k] THEN

RETURN 2;

ELSE


```

RETURN 0
END;
END;

```

Если условие $k \geq 0$ ложно, то можно было бы распечатать множители, но есть один нюанс. Существуют так называемые тривиальные множители – это единица и само число, такие пары необходимо исключить из рассмотрения. Поэтому, прежде чем принимать окончательное решение о печати множителей, надо проверить, не является ли один из множителей единицей. Если число = единице, то, во-первых, оно имеет длину нуль (один разряд) и в этом нулевом разряде находится 1. Ситуация несколько усугубляется тем, что некоторое количество старших разрядов у каждого делителя могут быть нулями, их надо отбросить и потом то, что осталось, проверить на равенство единице:

```

WHILE (Множ1.Длина>0) & (Множ1.Цифра[Множ1.Длина]=0) DO
  DEC(Множ1.Длина);
END;
WHILE (Множ2.Длина>0) & (Множ2.Цифра[Множ2.Длина]=0) DO
  DEC(Множ2.Длина);
END;
IF (Множ1.Длина=0) & (Множ1.Цифра[0]=1) THEN RETURN 0 END;
IF (Множ2.Длина=0) & (Множ2.Цифра[0]=1) THEN RETURN 0 END;

```

Если в процессе отработки этого фрагмента не было выполнено возврата, то тогда уже точно пора печатать множители:

```

StdLog.Ln;StdLog.String('Первый множитель');
FOR k:= Множ1.Длина TO 0 BY -1 DO
  StdLog.Int(Множ1.Цифра[k]);
END;
StdLog.Ln;
StdLog.Ln;StdLog.String('Второй множитель');
FOR k:= Множ2.Длина TO 0 BY -1 DO
  StdLog.Int(Множ2.Цифра[k]);
END;
StdLog.Ln;

```

И возвращать на уровень вверх сообщение об успешности выполненной работы:

```

RETURN 1;

```

И наконец, полный листинг решения:

Листинг 5.10

```

MODULE Модуль;
IMPORT In, StdLog;
TYPE

```

```

Число=RECORD
  Длина:INTEGER;
  Цифра:ARRAY 100 OF INTEGER;
END;
PROCEDURE Факторизация*;
VAR
  Фактор,Множ1,Множ2,Произв: Число;
  k,c,x,y:INTEGER;
  flag:BOOLEAN;
PROCEDURE Сравнение(Произв,Множ1,Множ2:Число):INTEGER;
VAR
  k:INTEGER;
BEGIN
  IF Множ1.Длина+Множ2.Длина> Фактор.Длина THEN RETURN 2; END;
  IF Произв.Длина> Фактор.Длина THEN RETURN 2 END;
  IF Произв.Длина< Фактор.Длина THEN RETURN 0 END;
  k:= Фактор.Длина;
  WHILE (k>=0) & (Произв.Цифра[k]= Фактор.Цифра[k]) DO k:=k-1;END;
  IF k>=0 THEN
    IF Произв.Цифра[k]> Фактор.Цифра[k] THEN
      RETURN 2;
    ELSE
      RETURN 0
    END;
  END;
END;
WHILE (Множ1.Длина>0) & (Множ1.Цифра[Множ1.Длина]=0) DO
  DEC(Множ1.Длина);
END;
WHILE (Множ2.Длина>0) & (Множ2.Цифра[Множ2.Длина]=0) DO
  DEC(Множ2.Длина);
END;
IF (Множ1.Длина=0) & (Множ1.Цифра[0]=1) THEN RETURN 0 END;
IF (Множ2.Длина=0) & (Множ2.Цифра[0]=1) THEN RETURN 0 END;
StdLog.Ln;StdLog.String('Первый множитель');
FOR k:= Множ1.Длина TO 0 BY -1 DO
  StdLog.Int(Множ1.Цифра[k]);
END;
StdLog.Ln;
StdLog.Ln;StdLog.String('Второй множитель');
FOR k:= Множ2.Длина TO 0 BY -1 DO
  StdLog.Int(Множ2.Цифра[k]);
END;
StdLog.Ln;

```

```

RETURN 1;
END Сравнение;
PROCEDURE ДостройкаМножителя(Множ1,Множ2,Произв:Число):BOOLEAN;
VAR
  x,y,s,c:INTEGER;
  b: Число;
BEGIN
  Множ1.Длина:= Множ1.Длина+1;
  b:=Произв;
  FOR x:=0 TO 9 DO
    Множ1.Цифра[Множ1.Длина]:=x;
    Произв:=b;
    FOR y:=0 TO Множ2.Длина DO
      s:= Произв.Цифра[y+ Множ1.Длина]+ Множ2.Цифра[y]*x;
      Произв.Цифра[y+ Множ1.Длина]:=s MOD 10;
      Произв.Цифра[y+Множ1.Длина+1]:=Произв.Цифра[y+Множ1.Длина+1]
        +(s DIV 10);
    END;
    Произв.Длина:= Множ1.Длина+ Множ2.Длина+1;
    WHILE (Произв.Длина>0) & (Произв.Цифра[Произв.Длина]=0) DO
      DEC(Произв.Длина);
    END;
    c:=Сравнение(Произв, Множ1,Множ2);
    IF (c=0) & (ДостройкаМножителя(Множ2, Множ1, Произв) OR
      ДостройкаМножителя(Множ1,Множ2,Произв)) THEN RETURN TRUE
    END;
    IF c=1 THEN RETURN TRUE END;
    IF c=2 THEN RETURN FALSE END;
  END;
  RETURN FALSE;
END ДостройкаМножителя;
BEGIN
  (*Инициализация массива результата*)
  FOR k:=0 TO 99 DO
    Произв.Цифра[k]:=0;
  END;
  (*Ввод числа*)
  In.Open;
  k:=-1;
  REPEAT
    k:=k+1;
    In.Int(Фактор.Цифра[k]);
  UNTIL ~In.Done;

```

```

(*Переворачивание числа*)
Фактор.Длина:=k-1;
FOR k:=0 TO Фактор.Длина DIV 2 DO
  c:= Фактор.Цифра[k];
  Фактор.Цифра[k]:= Фактор.Цифра[Фактор.Длина-k];
  Фактор.Цифра[Фактор.Длина-k]:=c;
END;
(*Расчеты*)
Множ1.Длина:=0;
Множ2.Длина:=0;
flag:=TRUE;
x:=0;
WHILE flag & (x<=9) DO
  Множ1.Цифра[0]:=x;
  y:=0;
  WHILE flag & (y<=9) DO
    Множ2.Цифра[0]:=y;
    c:=x*y;
    Произв.Цифра[0]:=c MOD 10;
    Произв.Цифра[1]:=c DIV 10;
    IF Произв.Цифра[1]>0 THEN
      Произв.Длина:=1;
    ELSE
      Произв.Длина:=0;
    END;
    c:=Сравнение(Произв, Множ1, Множ2);
    IF (c=0) & (Произв.Цифра[0]= Фактор.Цифра[0]) THEN
      flag:=~(ДостройкаМножителя (Множ1, Множ2, Произв)
        OR ДостройкаМножителя(Множ2, Множ1, Произв));
    ELSIF c=1 THEN flag:=FALSE;
    END;
    y:=y+1;
  END;
  x:=x+1;
END;
IF flag THEN StdLog.String('Решение не найдено');StdLog.Ln; END;
END Факторизация;
END Модуль.

```

Но это еще не все. Наша реализация еще очень сырая. Есть некоторые возможности для оптимизации, используя которые можно очень существенно увеличить производительность программы и серьезно обогнать прямолинейный перебор потенциальных делителей.

Задание для самостоятельной работы

Реализуйте описанные ниже возможности для увеличения производительности программы.

Первая возможность. В главной процедуре мы серьезно ограничили перебор из того соображения, что продолжать можно только те варианты построения множителей, при которых перемножение первых цифр множителей дает первую цифру факторизуемого числа. Конечно же такой критерий отсева возможен и для процедуры достройки множителя, но в процедуре этого нет. Ввод такого критерия скачком увеличит производительность программы.

Вторая возможность. В принципе, множители факторизуемого числа могут быть какими угодно, но наиболее трудно поддаются разложению числа, представляющие произведения длинных простых, максимально длинные простые в разложении получаются только в том случае, если они одинаковой или примерно одинаковой длины. Если одно простое длинное, то другое, очевидно, короткое, и это короткое будет быстро обнаружено. Это наводит на мысль, что есть смысл отсекаать варианты, в которых один из множителей существенно короче другого.

Третья возможность. В функции, сравнивающей числа, результат 2 возвращается, если суммарная длина множителей больше длины факторизуемого числа и если длина произведения больше факторизуемого числа. Понятно, что длина произведения определяется длиной множителей, и, наверное, эти два условия можно объединить в одно, уменьшив количество вычислений.

В заключение. Безусловно, рекурсия – достаточно сложный инструмент, но и предназначена рекурсия для задач с непростой логикой. На механизмы рекурсии опирается метод динамического программирования. Рекурсия позволяет перевести проблему организации ветвящихся процессов в чисто техническую задачу. Конечно, при этом возникают некоторые сложности. В рекурсивной задаче легко наскочить на ситуацию переполнения стека, не всегда легко определить условие завершения последовательности активаций, могут быть проблемы с определением передаваемых фактических параметров и механизмом возврата результата, но все же инструмент стоит того, чтобы им овладеть.

Сортировки

Общая постановка задачи	167
Обменные сортировки.	
Сортировка пузырьком.....	168
Шейкерная сортировка	170
Анализ качеств алгоритма	171
Сортировка выбором	174
Алгоритм вставки	176
Сортировка Шелла.....	178
Быстрая сортировка.....	181
Двоичная сортировка	186
Сортировка слияниями	191

Общая постановка задачи

Общая задача сортировки заключается в расстановке элементов некоторого множества в соответствии с заданным порядком. Математическое понятие порядка довольно сложно, но когда о порядке говорят программисты, то обычно имеется в виду, что элементы множества упорядочены по возрастанию или по убыванию. Обсудим значение терминов «возрастание» и «убывание».

Исходные множества могут быть в принципе различной природы, даже более того, элементам исходного множества не запрещено быть сложными структурами. Это, к примеру, паспортные данные человека, описание товара, авиационный или железнодорожный билет, идентификационные данные на книгу. В любом языке программирования есть структура данных, предназначенная для описания сложных данных. В языке Паскаль такой структурой является запись – конструкция данных вида:

```
TYPE Элемент=RECORD
    (*Описание компонентов записи*)
END;
```

Сложная структура записей создает принципиальную проблему. Пусть, например, дан массив записей следующего вида:

```
TYPE Элемент=RECORD
    Фамилия: ARRAY 20 OF CHAR;
    Возраст: BYTE;
    Номер:INTEGER; (*Какой-то порядковый номер*)
END;
```

Массив таких записей можно сортировать по любому из объявленных полей. Для поля фамилии можно определить так называемый лексикографический порядок, для числовых полей порядок определяется отношениями $>$, $<$, $=$. Но что точно никак нельзя сделать – это выполнить сортировку сразу по нескольким полям, так как возможна ситуация конфликта, для двух элементов **A** и **B** по одному полю может оказаться $A < B$, а по другому $A > B$. Поэтому сортировка выполняется по какому-то одному полю, которое называется ключом. Считается, что $A > B$, если Ключ $A >$ Ключ B . Тогда структура записи приобретает следующий вид:

```
TYPE Элемент=RECORD
    Ключ: INTEGER;
    (*Описание всех прочих компонентов записи*)
END;
```

На поле ключа не накладывается никаких ограничений по типу, но мы далее для упрощения рассуждений будем иметь в виду только числовые ключи, даже только ключи типа **INTEGER**. Более того, раз структура записи после выделения ключа уже не имеет для сортировки никакого значения, есть смысл говорить только о сортировке числовых массивов. И последнее, сортировка по возрастанию принципи-

ально ничем не отличается от сортировки по убыванию. Поэтому сформулируем нашу задачу так *«Дан массив типа INTEGER, упорядочить его элементы в порядке возрастания»*. Необходимую структуру данных можно описать так:

TYPE массив=ARRAY *ДлинаМассива* OF INTEGER

Алгоритмов сортировки существует довольно много. Каждый из них в принципе решает задачу упорядочивания, но каждый из них делает это немного по-своему. Отличаются алгоритмы скоростью работы, запросами к памяти, сложностью своего устройства, есть и некоторые другие, более тонкие особенности. По главной своей идее алгоритмы можно разделить на сортировки выбором, вставками и обменами. Но мы пока воспользуемся классификацией по сложности и начнем с простых алгоритмов, для того чтобы эта достаточно трудная тема была более понятна. Начнем с трех простых алгоритмов: простого обмена, простой вставки и простого выбора. На них же и немного проанализируем проблему оценки качества алгоритма.

Обменные сортировки. Сортировка пузырьком

Сортировка обменом предполагает операцию обмена, при которой два элемента массива меняются местами. Различные обменные сортировки отличаются друг от друга порядком прохода массива и выбором элементов для обмена. Простейшая из обменных сортировок и, наверное, сортировок вообще – это пузырьковая сортировка.

Идея пузырьковой сортировки – в определении неправильной пары. Неправильная пара – это пара рядом стоящих элементов, таких что первый элемент пары больше второго. Полностью упорядоченный массив – это массив, в котором нет ни одной неправильной пары. Отсюда идея сортировки. Необходимо организовать перебор пар массива и каждый раз, когда будет обнаружена неправильная пара, её элементы менять местами. Важен вопрос о количестве проходов массива. Ясно, что одного прохода может оказаться недостаточно. Рассмотрим пример. Пусть дан массив (4, 3, 2, 1). Проход массива заключается в переборе 3 пар элементов. Выполним проход пошагово:

- Сравниваются (4, 3). Результат (3, 4, 2, 1).
- Сравниваются (4, 2). Результат (3, 2, 4, 1).
- Сравниваются (4, 1). Результат (3, 2, 1, 4).

Заметим, что после первого прохода число 4 (наибольшее в массиве) встало на свое законное место. По всей видимости, после каждого прохода одно число будет занимать предназначенное для него место. Исключение составит последний проход, его выполнение приведет к упорядочению сразу двух чисел. Таким образом, количество проходов равно $N-1$, если N – это количество чисел. В процедуре, записанной ниже, верхняя граница цикла $n-2$ вместо $n-1$. Это следствие того, что в процедуре количество чисел не n , а $n+1$, n – номер последнего элемента массива.

Итак, нам необходимы $n-1$ операций прохода массива:


```
FOR i:=0 TO n-2 DO
  (*Проход массива и его сортировка*)
END;
```

Проход массива необходим для операции сравнения пары рядом стоящих элементов, это запишется следующим фрагментом:

```
FOR j:=0 TO n-i-2 DO
  IF a[j]>a[j+1] THEN
    (*Обмен значениями между элементами неправильной пары*)
  END;
END;
```

Обмен значениями выполняется тремя операторами присваивания:

```
c:=a[j]; a[j]:=a[j+1]; a[j+1]:=c;
```

И наконец, полный листинг решения:

Листинг 6. 1

```
PROCEDURE Пузырек(a:массив;n:INTEGER);
VAR
  i,j,c:INTEGER;
BEGIN
  FOR i:=0 TO n-2 DO
    FOR j:=0 TO n-i-2 DO
      IF a[j]>a[j+1] THEN
        c:=a[j];
        a[j]:=a[j+1];
        a[j+1]:=c;
      END;
    END;
  END;
  FOR i:=0 TO n-1 DO
    StdLog.Int(a[i]);
  END;
END Пузырек;
```

Задание для самостоятельной работы

Наш вариант программы имеет один недостаток. Рассмотрим пример неупорядоченного массива (4, 1, 2, 3). Этот массив будет упорядочен пузырьком за один проход. Но программа добросовестно отработает три прохода, из которых два лишних. Попробуйте модифицировать программу таким образом, чтобы работа внешнего цикла прекращалась по реальному завершению сортировки. Подсказка: если массив полностью упорядочен, то при очередном проходе внутренний цикл не обнаружит ни одной неправильной пары.

Почему такое название. Не так важно, можете вы или нет объяснить происхождение названия сортировки, если вы можете её реализовать, но все же поясним, откуда взялся термин «*пузырек*». Поставьте сортируемый массив вертикально, так чтобы начало оказалось вверху. Тогда окажется, что числа, большие по значению (тяжелые), опускаются вниз (тонут), а меньшие (легкие) поднимаются вверх, как пузырьки.

Шейкерная сортировка

Пузырьковая сортировка допускает несложное улучшение. Как уже было замечено, после первого прохода «пузырьком» самый большой элемент массива встанет на свое место. Выполним второй проход наоборот, от предпоследнего элемента до первого. После этого прохода встанет на свое место самый маленький элемент. Так и будем выполнять наши проходы массива: нечетные – слева направо и четные – справа налево. При этом на нечетных проходах будет занимать свое место самый большой элемент (из оставшихся), а при четных – самый маленький (также из оставшихся). Такой способ называется шейкерной сортировкой. Реализацию шейкерной сортировки не будем разбирать подробно. Запишем сразу листинг.

Листинг 6.2

```
PROCEDURE Шейкер(a:массив;n:INTEGER);
  VAR
    i,R,L,c:INTEGER;
BEGIN
  L:=0;R:=n-1;
  WHILE L<R DO
    FOR i:=L TO R-1 DO
      IF a[i]>a[i+1] THEN
        c:=a[i];a[i]:=a[i+1];a[i+1]:=c;
      END;
    END;
    R:=R-1;
    FOR i:=R TO L+1 BY -1 DO
      IF a[i]<a[i-1] THEN
        c:=a[i];a[i]:=a[i-1];a[i-1]:=c;
      END;
    END;
    L:=L+1;
  END;
  FOR i:=0 TO n-1 DO
    StdLog.Int(a[i]);
  END;
END Шейкер;
```

Анализ качеств алгоритма

Производительность

Классифицировать алгоритмы сортировки можно по разным параметрам, например по производительности (объем работы, выполняемый за единицу времени). Но для оценки производительности программ и алгоритмов такой подход не очень хорош, так как такая оценка сильно зависит от быстродействия оборудования. А интересна производительность алгоритма, так сказать в чистом виде. Производительность в чистом виде – это количество элементарных операций, которые необходимо выполнить для получения результата в зависимости от размера обрабатываемых данных. Для сортировки размер обрабатываемых данных – это длина сортируемого массива. В качестве элементарных операций разумно взять элементарные языковые операции – это операции сравнения и присваивания. Суммарное количество таких операций и будет характеризовать производительность. Конечно же время, затрачиваемое на сравнение, не обязательно равно времени присваивания, более того, время выполнения сравнений может быть разным, как и время выполнения присваиваний. Но уточнение этих величин очень сильно усложнит задачу оценки, кроме того, высокая точность расчетов в этом деле и не нужна. Это только лишь оценка, которая с ростом размера сортируемого массива постепенно превращается в более точную.

Расчет количества операций как функции длины массива – в общем случае довольно сложное и трудоемкое дело. Но для пузырьковой сортировки такой расчет почти очевиден. Число сравнений в сортировке:

$$N_1 = \frac{n^2 - n}{2}.$$

Это значение получено из следующих соображений. Если сравнивать каждый с каждым, то для n элементов количество сравнений равно n^2 , по мере упорядочивания массива его элементы выбывают из процесса, поэтому оценка должна быть меньше, чем n^2 . После каждого прохода массива один элемент становится на свое место и, следовательно, не участвует в дальнейших сравнениях. Тогда на первом шаге сравнений $n-1$ (первый элемент сам с собой не сравнивается). На втором $n-2$ и т. д. Итого:

$$N_1 = (n-1) + (n-2) \dots + 0.$$

Здесь записана сумма арифметической прогрессии. Формула суммы:

$$S_n = \frac{a_1 + a_n}{2} n, \text{ тогда имеем } N_1 = S_n = \frac{n-1+0}{2} n = \frac{n^2 - n}{2}. \text{ А оценка коли-}$$

чества присваиваний выражается формулой: $N_2 = \frac{3(n^2 - n)}{2}$. Такая оценка получается также из очень простых соображений. Предположим, что каждое сравнение

потребуется перестановки двух элементов местами. Конечно, это наихудшая ситуация из возможных (некоторые из сравнений могут быть холостыми), но оценка – это верхняя граница количества операций, поэтому нас именно наихудшая и интересует. Перестановка двух элементов A и B выполняется так: $C = A$; $A = B$; $B = C$. То есть на каждую операцию сравнения придется три присваивания. Отсюда появляется число 3 в числителе.

Расчет для шейкерной сортировки, несмотря на её кажущуюся простоту, значительно сложнее. Если читателю интересна её оценка, мы рекомендуем книгу Кнута (см. [3]). Шейкерная сортировка, впрочем, не дает существенного выигрыша по сравнению с пузырьковой, так как наибольшие временные потери приходится на операции обмена значениями между двумя элементами, а и в обычном пузырьке и в шейкерной сортировке обмену подлежат только рядом стоящие элементы, это означает, что операция обмена приводит к минимальному смещению «неправильно стоящего» элемента. Отсюда, кстати, вытекает идея ускорения сортировки – необходимо обеспечить за один шаг смещение большее, чем на одну позицию.

Если говорить в целом о проблеме оценки количества операций, то хотелось бы, чтобы та легкость, с которой мы получили оценку для пузырька, не ввела бы вас в заблуждение. На самом деле получение оценок требует довольно развитого математического аппарата. Наше изложение популярное, поэтому далее мы не будем увлекаться анализом скорости алгоритмов, а ограничимся небольшой качественной оценкой. Более детально проблема оценивания сортировок изложена в очень многих источниках, например [3] и в книге Вирта [15].

Устойчивость

Сортировка меняет порядок элементов. В этом и заключается её работа. Но массив может содержать некоторое количество элементов равной величины. Напомним, что сортируемые элементы массива – это ключи, характеризующие запись, содержащую какую-то информацию. Возможно, для некоторых целей важно, чтобы записи, имеющие одинаковый ключ, сохранили порядок следования и после выполнения сортировки. Если алгоритм сохраняет порядок равных элементов, то он считается устойчивым. Заметим, что сохранение устойчивости – не очевидное качество алгоритма. Существуют неустойчивые сортировки. Например, неустойчивой можно сделать пузырьковую сортировку, если фрагмент обмена элементов массива переписать так:

```
IF a[j]>a[j+1] THEN  
  c:=a[j];  
  a[j]:=a[j+1];  
  a[j+1]:=c;  
END;
```

Замена строгого неравенства на нестрогое превращает устойчивый пузырек в неустойчивый, так как сравнение на равенство делает возможной операцию обмена и в том случае, когда ключи равны. Но неустойчивость пузырьковой сортировки можно считать следствием ошибки. Ниже у нас будет возможность убедиться,

что устойчивость в принципе не всегда достижима. Шейкерную и пузырьковую сортировки мы отнесем к устойчивым.

Требования к памяти

Пузырек и шейкерная сортировка не требуют дополнительной памяти сверх того, что необходимо для хранения массива. Использование здесь дополнительной памяти ничего и не даст. Но в программировании, как и в технике, действует золотое правило, которое гласит, если мы готовы в чем-то проиграть, то, может быть, сможем в чем-то другом выиграть. Например, использование дополнительных структур данных (потеря памяти) может дать выигрыш в скорости. Простой и очень эффективный пример действия золотого правила разобран в 3-ей главе – это решето Эратосфена. Его эффективность достигается использованием большого вспомогательного массива. Мы теряем память и увеличиваем скорость, этот же эффект возможен и для процесса сортировки, и такие алгоритмы действительно существуют. В качестве примера можно привести очень простой алгоритм сортировки подсчетом.

Идея алгоритма заключается в простом подсчете количества вхождений каждого числа в массив. Это делается так: выполним один проход массива и для каждого числа из отрезка $[0, \text{max}]$ подсчитаем, сколько раз оно встретится. Вычисленные количества сохраним в специальном массиве счетчиков. Затем запишем в массив-результат (а это может быть и исходный массив) каждое число столько раз, чему равно значение соответствующего ему счетчика, то есть сколько раз оно встретилось.

Листинг 6.3

```
MODULE Модуль;  
  IMPORT In, StdLog;  
  PROCEDURE СортировкаПодсчетом*;  
  VAR  
    a:ARRAY 100 OF INTEGER;  
    b:ARRAY 1000 OF INTEGER;  
    N,i,k,j:INTEGER;  
BEGIN  
  N:=-1;  
  In.Open;  
  REPEAT  
    N:=N+1;  
    In.Int(a[N]);  
  UNTIL ~In.Done;  
  FOR i:=0 TO 999 DO  
    b[i]:=0;  
  END;  
  FOR i:=0 TO N-1 DO  
    b[a[i]]:=b[a[i]]+1;
```

```

END;
j:=0;
FOR i:=0 TO 999 DO
  FOR k:=1 TO b[i] DO
    a[j]:=i;
    j:=j+1;
  END;
END;
FOR i:=0 TO N-1 DO
  StdLog.Int(a[i]);
END;
END СортировкаПодсчетом;
END Модуль.

```

Здесь использование дополнительной памяти дает фантастическую скорость. Но размер памяти слишком сильно зависит от размера интервала сортируемых чисел, возможна ситуация, когда даже небольшой сортируемый массив потребует дополнительного массива очень большого размера. Как, например, массив из двух элементов: $a[0]=1$; $a[1]=1000000$.

Сортировка выбором

Идея алгоритма заключается в выборе на каждом шаге его работы наименьшего элемента из оставшихся. На первом шаге этот выбор осуществляется из всего массива, а найденное значение ставится в первую позицию. На втором шаге первый элемент исключается из рассмотрения, а найденный наименьший ставится во вторую позицию, на следующем шаге из рассмотрения исключаются уже два первых элемента и т. д. Не будем разбирать реализацию детально. Приведем сразу листинг решения:

Листинг 6.4

```

PROCEDURE СортировкаВыбором(a:массив;N:INTEGER);
VAR
  i,k,min,c:INTEGER;
BEGIN
  FOR i:=0 TO N-2 DO
    min:=i;
    (*Поиск очередного минимума*)
    FOR k:=i+1 TO N-1 DO
      IF a[min]>a[k] THEN
        min:=k;
      END;
    END;
  END;
  (*Установка найденного минимума в правильную позицию*)

```

```
c:=a[min];a[min]:=a[i];a[i]:=c;  
END;  
FOR i:=0 TO N-1 DO  
  StdLog.Int(a[i]);  
END;  
END СортировкаВыбором;
```

В каком-то смысле этот алгоритм также можно считать обменным. Здесь очередной элемент обменивается значением с найденным. Но вся соль – в механизме поиска «*правильного места*». В отличие от пузырька, здесь на каждом шагу очередной элемент смещается, возможно, больше, чем на одну позицию. Вероятность этого «возможно, больше» сильно зависит от того, насколько неупорядочен исходный массив: если он почти упорядочен, то эффект может быть не слишком велик. Если массив сильно не упорядочен, то эффект окажется значительным.

Задание для самостоятельной работы

Реализуйте сортировку выбором, для которой обмен выполняется не для очередного минимального, а для очередного максимального элемента.

Задание для самостоятельной работы

Попробуйте выяснить, является ли алгоритм простого выбора устойчивым.

Попробуем оценить производительность алгоритма. Количество сравнений, видимо, будет таким же, как и для обменного алгоритма:

$$N_1 = \frac{n^2 - n}{2}.$$

Для оценки количества присваиваний придется обратиться к вероятностной терминологии. Очередной элемент массива участвует в обмене, только если он меньше стоящих правее. Естественно, можно говорить только о вероятности такого события. Прикинем вероятности. Для второго элемента возможны только два события, он меньше первого или нет. Тогда вероятность его обмена $1/2$. Рассуждая таким же образом для третьего элемента, получаем вероятность $1/3$ и т. д. Тогда среднее число пересылок равно:

$$N_2 = 1/2 + 1/3 + \dots + 1/k.$$

Это для элемента с индексом k . Чтобы получить среднее количество присваиваний для всего массива, необходимо выполнить суммирование по k . Выполнив суммирование и проведя некоторые дополнительные выкладки, получим следующее выражение:

$$N_2 = n * (\ln(n) + g).$$

где $g = 0.5772$ – число Эйлера.

Задание для самостоятельной работы

Проведите анализ сортировки выбором на предмет её устойчивости.

Следующая общая идея сортировок – это сортировка вставками. Если в обменных сортировках решалась задача поиска двух элементов массива для обмена, то сейчас будет приниматься решение относительно правильного места для каждого очередного элемента. Правильное место – это положение в массиве, где должен стоять выбранный элемент, в полностью упорядоченном массиве. А различаются вставочные алгоритмы способом принятия такого решения.

Сортировка вставками

Идея алгоритма. Будем просматривать массив, начиная с первого элемента по последний. Скажем, что элемент стоит на своем месте, если все элементы, находящиеся левее его, меньше по значению. И скажем, что элемент не на своем месте, если слева от него есть элементы, большие по значению. Тогда если в процессе просмотра массива встретился элемент, стоящий не на своем месте, необходимо определить для него *«правильную позицию»* и выполнить вставку. Заметим, что перед началом процесса все элементы считаются стоящими не на своем месте.

Как определить *«правильную позицию»*. Вернемся к началу массива, и идя от начала, найдём элемент, такой что он больше **НЕПРАВИЛЬНО СТОЯЩЕГО** и в то же время он первый, обладающий таким свойством. Назовем его **НАЙДЕННЫМ**. Тогда место **НЕПРАВИЛЬНО СТОЯЩЕГО** – перед **НАЙДЕННЫМ**.

Для вставки необходимо выполнить следующие действия (алгоритм **ВСТАВКИ**):

- заппомним значение **НЕПРАВИЛЬНО СТОЯЩЕГО**;
- сдвинем массив от **НАЙДЕННОГО** (включая его) до **НЕПРАВИЛЬНО СТОЯЩЕГО** (включая его) на одну позицию;
- присвоим значение **НЕПРАВИЛЬНО СТОЯЩЕГО** элементу перед **НАЙДЕННЫМ**.

Начнем реализацию алгоритма. Так как для каждого элемента решение о его правильном месте принимается один раз, то главный процесс можно оформить в виде цикла, однократно пробегающего все элементы массива:

```
FOR i:=0 TO N-1 DO  
  (*Операции поиска и вставки*)  
END;
```

Решение принимаем для элемента с индексом i . Необходимо найти элемент массива первый, уже больший, чем i -й элемент. Для этого достаточно, начиная с первого (с нулевым номером) элемента, пройти все элементы, меньшие i -го.

```
FOR i:=0 TO N-1 DO (*Однократный проход массива*)  
  k:=0;  
  (*Поиск для i-го элемента правильной позиции*)  
  WHILE a[i]>a[k] DO k:=k+1;END;  
END;
```


По выходу из цикла **WHILE** переменная **k** будет содержать номер искомой позиции. Но здесь возможны две ситуации. Может оказаться, что i -й элемент массива уже стоит в правильной позиции. Если это так, то $k=i$, следовательно, вставку необходимо делать, только если $k < i$.

```
FOR i:=0 TO N-1 DO (*Однократный проход массива*)
  k:=0;
  (*Поиск для i-го элемента правильной позиции*)
  WHILE a[i]>a[k] DO k:=k+1;END;
  IF k<i THEN (*Если правильная позиция не совпадает с исходной i-й*)
    (*Сдвиг массива и вставка*)
  END;
END;
```

Выполнять сдвиг будем слева направо до элемента **a[i]**, поэтому элемент **a[i]** необходимо перед операцией сдвига запомнить во вспомогательной переменной: **c:=a[i]**. После чего можно сдвинуть массив:

```
FOR j:=i TO k+1 BY -1 DO
  a[j]:=a[j-1];
END;
```

И выполнить вставку: **a[k]:=c**. Ниже – полный листинг реализации:

Листинг 6.5

```
PROCEDURE СортировкаВставкой(a:массив;N:INTEGER);
VAR
  i,k,j,c:INTEGER;
BEGIN
  FOR i:=0 TO N-1 DO (*Однократный проход массива*)
    k:=0;
    (*Поиск для i-го элемента правильной позиции*)
    WHILE a[i]>a[k] DO k:=k+1;END;
    IF k<i THEN (*Если правильная позиция не совпадает с исходной i-й*)
      c:=a[i]; (*Запомним элемент во вспомогательной переменной*)
      (*Сдвинем массив*)
      FOR j:=i TO k+1 BY -1 DO
        a[j]:=a[j-1];
      END;
      (*Вставим элемент в правильную позицию*)
      a[k]:=c;
    END;
  END;
  FOR i:=0 TO N-1 DO
    StdLog.Int(a[i]);
  END;
END СортировкаВставкой;
```

Задание для самостоятельной работы

Алгоритм вставки допускает простое и очень эффективное улучшение. Ключевой момент вставки – это поиск «правильной позиции». Заметим, что если речь идет о поиске нового места для элемента с номером L , то, значит, массив с первого по $L-1$ элемент уже упорядочен. Поиск известной величины в линейно-упорядоченном множестве легко выполняется методом половинного деления, что должно дать заметный выигрыш по сравнению с простым перебором элементов. Имеется в виду следующий фрагмент:

```
k:=0; (*Поиск простым перебором*)  
WHILE a[i]>a[k] DO k:=k+1; END;
```

Напишите реализацию алгоритма вставки, в котором простой перебор заменен методом половинного деления.

Задание для самостоятельной работы

Ответьте на вопрос, является ли наша реализация вставки устойчивой, и если нет, то почему и можно ли сделать её таковой.

Сортировка Шелла

Цель алгоритма – заставить каждый сортируемый элемент за шаг вставки перемещаться большим скачком. Для этого исходный массив разбивается на группы специальным образом. Предположим, в массиве 16 чисел (число, кратное двойке, взято только для удобства). Выполним разбиение массива на пары следующим образом: (a_1, a_9) , (a_2, a_{10}) ... (a_8, a_{16}) . Внутри каждой группы выполним сортировку методом вставок. Затем переходим к следующему разбиению, в котором элементы массива объединяются уже по 4 следующим образом (a_1, a_5, a_9, a_{13}) , $(a_4, a_8, a_{12}, a_{16})$ и т. д. Теперь элементы отстоят друг от друга на 4 позиции. На следующем шаге – соответственно на 2 позиции, и на последнем – на 1. Последовательность расстояний внутри групп в нашем примере: 8, 4, 2, 1. Ясно, что такая последовательность невозможна для любого массива, но сортировка Шелла этого и не требует. Существенно важно только одно – на последнем шаге величина расстояния между элементами должна быть равна единице, последовательность же таких расстояний может быть любой, выбор последовательности не влияет на окончательный результат, хотя может повлиять на скорость сортировки.

Начнем работу над реализацией. Во-первых, необходимо договориться о размере групп. Вопрос выбора оптимального разбиения массива на группы на сегодня остается открытым. Возможно, потому, что ответ на этот вопрос самым существенным образом зависит от сортируемого массива и для принятия решения необходима какая-то информация о распределении значений в массиве. Мы для нашей реализации решим этот вопрос так: пусть на первой итерации массив делится на две группы, затем на 4 и т. д. Это означает, что на первом шаге расстояние между элементами группы будет $n \text{ DIV } 2$ (n – количество элементов массива), затем на каждой итерации расстояние уменьшается вдвое, и последняя итерация выполняется при расстоянии, равном единице. Запишем фрагмент:

```

k:=n DIV 2;
WHILE k>=1 DO
  k:=k DIV 2;
END;

```

Это, так сказать, каркас, на который мы сейчас начнем настраивать операции сортировки. Организуем проход массива:

```

k:=n DIV 2;
WHILE k>=1 DO
  FOR i:=k TO n-1 DO
    (*анализ и перемещение элементов массива*)
  END;
  k:=k DIV 2;
END;

```

Цикл начинается с величины k для того, чтобы идти по группе вниз, к началу. Далее вспомним, что сортировка вставками состоит из двух операций: поиска для текущего элемента правильной позиции и вставки его в найденную позицию. Эти две операции вполне можно выполнять отдельно: цикл для поиска и цикл для вставки. Однако операции поиска и вставки можно объединить. Пусть, к примеру, правильная позиция отстоит от текущего элемента на 5 позиций. Это означает, что цикл анализа должен выполнить пять операций сравнения и цикл вставки – пять операций смещения. Идея их совмещения напрашивается сама собой. Ниже – фрагмент, их совмещающий:

```

x:=a[i]; (*Это анализируемый элемент*)
j:=i-k; (*Уходим по группе на шаг назад*)
WHILE (j>=k) & (x<a[j]) DO
  (*Пока группа не пройдена и не найдена позиция*)
  a[j+k]:=a[j]; (*Выполняем смещение*)
  j:=j-k; (*Отходим еще на шаг назад к началу группы*)
END;
(*окончательное смещение*)
IF (j>=k) OR (x>=a[j]) THEN
  a[j+k]:=x;
ELSE
  a[j+k]:=a[j];
  a[j]:=x;
END

```

Запишем полный листинг реализации.

Листинг 6.6

```

MODULE Модуль;
IMPORT In, StdLog;
VAR

```

```

a:ARRAY 100 OF INTEGER;
n:INTEGER;
PROCEDURE СортировкаШелла;
VAR
  i,j,k,x:INTEGER;
BEGIN
  k:=n DIV 2;
  WHILE k>=1 DO
    FOR i:=k TO n-1 DO
      x:=a[i];j:=i-k;
      WHILE (j>=k) & (x<a[j]) DO
        a[j+k]:=a[j];j:=j-k;
      END;
      IF (j>=k) OR (x>=a[j]) THEN
        a[j+k]:=x;
      ELSE
        a[j+k]:=a[j];
        a[j]:=x;
      END;
    END;
    k:=k DIV 2;
  END;
END СортировкаШелла;
PROCEDURE Главная*;
VAR
  k:INTEGER;
BEGIN
  In.Open;
  In.Int(n);
  FOR k:=0 TO n-1 DO In.Int(a[k]);END;
  СортировкаШелла;
  FOR k:=0 TO n-1 DO StdLog.Int(a[k]); END;
END Главная;
END Модуль.

```

Замечание о производительности. Производительность алгоритма вставки можно сравнивать с производительностью алгоритма сортировки выбором, так как простые обменные сортировки конкуренции со вставкой не выдержат. Что же касается сравнения с выбором, то здесь много зависит от начального расположения элементов массива. Можно уверенно полагать, что если массив в значительной степени упорядочен, то алгоритмы вставки выполняют свою работу быстрее, в общем же случае алгоритмы выбора все же предпочтительнее.

Быстрая сортировка

Поделим исходный массив на два подмассива относительно центрального элемента (назовём его медианой). И выполним сортировку следующим образом: если элемент массива больше медианы, то перенесем его вправо за медиану, иначе – влево. По завершении этой операции массив окажется немного более упорядоченным. Продолжим упорядочение, разбив каждый из уже обработанных подмассивов ещё на два подмассива, и проведём над ними ту же самую операцию. Полностью процесс сортировки будет завершен, когда длина подмассивов окажется единичной. Сразу заметим, что сортировка имеет один недостаток. Например, если исходный массив окажется изначально полностью или почти полностью упорядоченным, это не уменьшит количества выполняемых операций. Быстрая сортировка всё равно проведет дробление исходного массива до победного конца.

Алгоритм быстрой сортировки разумно реализовать в виде рекурсивной процедуры, которая на вход получала бы отрезок исходного массива (в виде номеров элементов массива, являющихся границами отрезка) и выполняла следующие операции:

- выбор элемента – медианы (можно середину);
- для всех элементов слева от медианы:
 - если значение элемента больше значения медианы, то переносим его вправо;
- для всех элементов справа от медианы:
 - если значение элемента меньше значения медианы, то переносим его влево.

Начнем разработку реализации. Рекурсивная процедура должна некоторый очередной отрезок массива поделить на две части (для упрощения задачи будем делить отрезок пополам), для чего процедуре требуются на входе левая и правая границы сортируемого отрезка. Первый вызов выполняет свои операции над целым массивом, поэтому первый вызов должен выглядеть так:

БыстраяСортировка(0, N-1);

Для дальнейшего анализа введем некоторые обозначения: **Левая** – левая граница сортируемого отрезка; **Правая** – правая граница, **Медиана** – середина отрезка. Работа любой рекурсивной процедуры заключается в двух вещах: обработке сложного случая, для которого выполняются какие-то действия и выполняются новые активации, и в обработке тривиального случая, обходящегося без дальнейших активаций. Тривиальным случаем является отрезок, состоящий из одного числа. Такой отрезок уже не нуждается в обработке. Следовательно, сложный случай будет описан следующим фрагментом:

```
IF Правая-Левая>1 THEN
  (*Обработка сложного случая*)
END;
```

Первым действием обработки сложного случая будет поиск середины, это вы-

полнит один оператор:

Медиана:=**(Левая+Правая)** DIV 2;

Далее просмотрим весь отрезок и для каждого элемента примем решение о его переносе.

Медиана:=**(Левая+Правая)** DIV 2;

k:=**Левая**;

WHILE k<=**Правая** DO

k:=k+1;

END;

Решение принимается в двух случаях:

- число находится слева от медианы и его значение больше значения медианы;
- число находится справа от медианы и его значение меньше значения медианы.

Указанные проверки изменяют наш фрагмент следующим образом:

Медиана:=**(Левая+Правая)** DIV 2;

k:=**Левая**;

WHILE k<=**Правая** DO

flag:=TRUE;

IF (a[k]>a[**Медиана**]) & (k< **Медиана**) THEN

(*перестановка*)

flag:=FALSE;

END;

IF flag & (a[k]<a[**Медиана**]) & (k> **Медиана**) THEN

(*Перестановка*)

END;

k:=k+1;

END;

Наверное, необходимо пояснить роль переменной **flag**. Если первое условие оказалось истинным, это приведет к перестановке и, естественно, изменению массива, после чего проверка второго условия просто потеряет смысл.

Листинг 6.7

MODULE **Модуль**;

IMPORT In, StdLog;

VAR

a:ARRAY 100 OF INTEGER;

PROCEDURE **БыстраяСортировка**(**Левая**, **Правая**:INTEGER);

VAR

c, **Медиана**,k,i:INTEGER;

flag:BOOLEAN;

BEGIN

IF **Правая-Левая**>1 THEN

```

Медиана:=(Левая+Правая) DIV 2;
k:=Левая;
WHILE k<=Правая DO
  flag:=TRUE;
  IF (a[k]>a[Медиана]) & (k<Медиана) THEN
    c:=a[k];
    FOR i:=k TO Медиана-1 DO
      a[i]:=a[i+1]
    END;
    a[Медиана]:=c;
    Медиана:=Медиана-1;
    flag:=FALSE;
    k:=k-1;
  END;
  IF flag & (a[k]<a[Медиана]) & (k>Медиана) THEN
    c:=a[k];
    FOR i:=k TO Медиана+1 BY -1 DO
      a[i]:=a[i-1];
    END;
    a[Медиана]:=c;
    Медиана:= Медиана+1;
  END;
  k:=k+1;
END;
IF Медиана >Левая THEN
  БыстраяСортировка(Левая, Медиана);
END;
IF Медиана<Правая THEN
  БыстраяСортировка(Медиана,Правая);
END;
END;
END БыстраяСортировка;
PROCEDURE Главная*;
VAR
  N,i,k,j,c:INTEGER;
BEGIN
  N:=-1;
  In.Open;
  REPEAT
    N:=N+1;
    In.Int(a[N]);
  UNTIL ~In.Done;
  БыстраяСортировка(0,N-1);
  FOR i:=0 TO N-1 DO

```

```

StdLog.Int(a[i]);
END;
END Главная;
END Модуль.

```

Еще один вариант быстрой сортировки

Надеемся, что вариант реализации, построенный выше, достаточно прозрачен и читаем, но возможен немного другой подход, дающий более короткую программу. Рассмотрим новую идею.

Каждая активация быстрой сортировки имеет дело с массивом, поделенным на две части: левую и правую. В левой части мы ищем элемент больший, чем медиана, а в правой – меньший, чем медиана. Если оба элемента будут найдены, то разумно их просто поменять местами, это избавит нас от большого количества смещений.

Начинается работа активации также с вычисления медианы. И так же как и в предыдущей реализации, медиану мы определим как середину. Затем запустим изменение двух индексов: одного – от левой границы в сторону медианы, другого – от правой границы в сторону медианы. Процесс должен продолжаться до тех пор, пока индексы не сравняются. Это условие целесообразно записать циклом REPEAT:

```

i:=Левая;j:=Правая;
x:=a[(Левая+Правая) DIV 2];
REPEAT
UNTIL i>j;

```

Займемся построением тела цикла. Как сказано выше, необходимо слева найти элемент, больший медианы, а справа – меньший. Сделаем это так:

```

i:=Левая;j:=Правая;
x:=a[(Левая+Правая) DIV 2];
REPEAT
  WHILE a[i]<x DO i:=i+1; END;
  WHILE x<a[j] DO j:=j-1; END;
UNTIL i>j;

```

Каждый из циклов обязательно решит свою задачу. Как крайний вариант в качестве элемента для перестановки будет найдена медиана. Найденные элементы меняются местами, но только в том случае, если индексы не перепрыгнули друг через друга:

```

IF i<=j THEN
  w:=a[i];a[i]:=a[j];a[j]:=w;
  i:=i+1;j:=j-1;
END;

```

Это обычная процедура обмена значениями между двумя элементами. Изменения индексов выполняются синхронно с обменом. Это правильно, так как измене-

ние индексов имеет смысл только до «прыжка друг через друга».

Осталось определить условия следующих активаций. Очевидно, что активации более не нужны, если длина сортируемого отрезка становится единичной. Это возможно в двух случаях: если индекс, идущий от правой границы, достигнет левой, и второй случай обратный – если индекс, идущий от левой границы, достигнет правой. Если же этого не произойдет, то нужна еще одна активация:

```
IF Левая<j THEN БыстраяСортировка(Левая,j); END;
IF i<Правая THEN БыстраяСортировка(i,Правая);END;
```

Заметим, что активации возможны две как максимум, но одна из них может и не выполняться, это следует из различного распределения чисел в отрезке, изменения правого и левого индексов не обязательно синхронны, именно поэтому для каждого из них записан свой оператор. Осталось записать полную сборку программы:

Листинг 6.8

```
MODULE Модуль;
IMPORT In, StdLog;
VAR
  a:ARRAY 100 OF INTEGER;
  n:INTEGER;
PROCEDURE БыстраяСортировка(Левая,Правая:INTEGER);
VAR
  i,j,x,w:INTEGER;
BEGIN
  i:=Левая;j:=Правая;
  x:=a[(Левая+Правая) DIV 2];
  REPEAT
    WHILE a[i]<x DO i:=i+1; END;
    WHILE x<a[j] DO j:=j-1; END;
    IF i<=j THEN
      w:=a[i];a[i]:=a[j];a[j]:=w;
      i:=i+1;j:=j-1;
    END;
  UNTIL i>j;
  IF Левая<j THEN БыстраяСортировка(Левая,j); END;
  IF i<Правая THEN БыстраяСортировка(i,Правая);END;
END БыстраяСортировка;
PROCEDURE Главная*;
VAR
  k:INTEGER;
BEGIN
  In.Open;
  In.Int(n);
  FOR k:=0 TO n-1 DO In.Int(a[k]);END;
```

```
БыстраяСортировка(0,n-1);  
FOR k:=0 TO n-1 DO StdLog.Int(a[k]); END;  
END Главная;  
END Модуль.
```

Еще одно важное замечание. Первая реализация стремится найденный неправильный элемент вставлять сразу после (или сразу перед) медианой. Таким образом, первая реализация, как кажется, более оптимальна, так как каждая активация стремится оставить после себя по возможности более упорядоченную структуру. Но, к сожалению, это не имеет большого смысла. Быстрая сортировка все равно честно будет делить массив на две части, пока не дойдет до минимальных подмассивов. И если на некотором шаге, например, правый отрезок окажется очень мал, то это будет означать лишь большую величину левого отрезка. Поэтому данное преимущество первой реализации не дает оптимизации, а вот необходимость большого количества дополнительных сдвигов безусловно затормозит работу. Поэтому вторая реализация не только более короткая, но и более оптимальна по скорости выполнения.

Замечание о быстродействии алгоритма. Быстрая сортировка вполне оправдывает свое название, она требует достаточно мало сравнений и обменов, но, к сожалению, не в любом случае. Оказывается, на быстродействие самое существенное влияние оказывает выбор медианы. В нашей реализации в качестве медианы всегда выбирается средний элемент. Это выгодно, если значения массива перемешаны максимально хаотично. Если перед быстрой сортировкой поставить легкую задачу, например упорядочить почти упорядоченный массив, то её преимущества быстро сойдут на нет. В легких задачах с быстрой сортировкой вполне сможет конкурировать даже пузырьковая. Поэтому выбор медианы очень важен. Мы вопрос выбора медианы опустим, а почитать о проблеме медианы можно в классической книге Н. Вирта [15].

Задание для самостоятельной работы

Выясните, является ли быстрая сортировка устойчивой.

Двоичная сортировка

Идея двоичной сортировки заключается в построении двоичного дерева и специальном размещении элементов массива в его вершинах. Пусть порядок обхода дерева определен следующим образом:

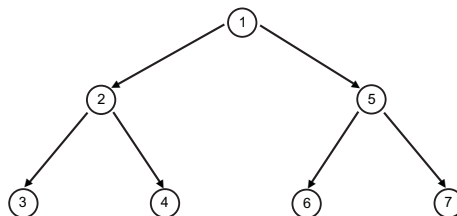


Рис. 6.1

Номерами помечены вершины в порядке их обхода. Это обход графа в глубину, и такой порядок обхода назовем для определенности правилом левой руки. Если элементы сортируемого массива расставить по дереву в таком порядке, что для любой пары чисел большее окажется правее меньшего, при обходе дерева по правилу левой руки, то сортировку можно будет считать выполненной, так как для восстановления массива будет достаточно пройти дерево и переписать значения вершин обратно в массив.

Осталось сформулировать правило построения нужного дерева. Предположим, что часть дерева уже построена. Первый шаг построения тривиален. Записываем в корень первый элемент массива, и так как дерево состоит из одной вершины, то условие очевидно выполняется. Далее для каждого последующего элемента массива проходим дерево и ищем ему подходящее место по следующему правилу: если значение очередной вершины больше числа из массива, то пытаемся уйти вправо по дереву. Если же ветви вправо нет, то создаем её и записываем в новую вершину число из массива, если же ветка есть, то переходим в существующую вершину и повторяем операцию. Если же значение вершины не больше числа из массива, то такие же действия выполняются с веткой влево. А сейчас займемся построением реализации. Главную процедуру организуем следующим образом:

```
PROCEDURE Главная*;  
VAR  
  k:INTEGER;  
BEGIN  
  In.Open;  
  In.Int(n);  
  FOR k:=0 TO n-1 DO In.Int(a[k]);END;  
  ДвоичнаяСортировка;  
  FOR k:=0 TO n-1 DO StdLog.Int(a[k]); END;  
END Главная;
```

Из текста ясно, что сортируемый массив в нашей реализации будет глобальным. Вся работа по построению дерева и переброске значений вершин обратно в массив возлагается на процедуру **ДвоичнаяСортировка**. Её мы построим так:

```
PROCEDURE ДвоичнаяСортировка;  
VAR  
  uk1:uk;  
  k,N:INTEGER;  
BEGIN  
  NEW(uk1);  
  uk1.a:=a[0];  
  FOR k:=1 TO n-1 DO  
    ДобавитьВершину(a[k],uk1);  
  END;  
  N:=0;
```

```
ВернутьМассив(uk1);
END ДвоичнаяСортировка;
```

Здесь до начала работы цикла первая вершина заполняется значением первого элемента массива, затем уже в цикле для каждого очередного элемента массива организуются проход дерева и поиск подходящего места (процедура **ДобавитьВершину**). Проход начинается всегда с корня. По завершении цикла выполняется процедура **ВернутьМассив**, перебрасывающая значения вершин обратно в массив. Так как массив глобальный, то переменная, обслуживающая его нумерацию *N*, инициализируется до входа в процедуру **ВернутьМассив**. Структура тип-дерево выглядит так:

```
Указатель=POINTER TO Вершина;
Вершина=RECORD
  a:INTEGER;
  Левая,Правая:Указатель;
END;
```

Следующий шаг – это разработка процедуры **ДобавитьВершину**. Её основная работа – это обход дерева, поэтому разумно построить рекурсивное решение. Рекурсивный процесс заключается в спуске по дереву, для чего на каждом шаге определяется направление спуска. Указатель, связанный с деревом, определим так:

```
VAR Дерево: Указатель;
```

А так выглядит процедура обхода:

```
IF a>Дерево.a THEN
  (*Попытаемся уйти вправо*)
ELSE
  (*Попытаемся уйти влево*)
END;
```

Попытка уйти вправо будет успешна только в том случае, если справа есть вершина, если же её нет, то её надо создать, и эта вершина является правильным местом для элемента массива, иначе вершина есть и необходимо выполнить переход. Запишем сказанное:

```
IF Дерево.Правая=NIL THEN
  NEW(Дерево.Правая); Дерево:=Дерево.Правая; Дерево.a:=a;
ELSE
  ДобавитьВершину(a,Дерево.Правая);
END;
```

Попытка переместиться влево выполняется аналогично, поэтому дадим полную запись текста процедуры:

```
PROCEDURE ДобавитьВершину(a:INTEGER; Дерево:Указатель);
BEGIN
```

```

IF a>Дерево.а THEN
  IF Дерево.Правая=NIL THEN
    NEW(Дерево.Правая); Дерево:=Дерево.Правая; Дерево.а:=а;
  ELSE
    ДобавитьВершину(а, Дерево.Правая);
  END;
ELSE
  IF Дерево.Левая=NIL THEN
    NEW(Дерево.Левая); Дерево:=Дерево.Левая; Дерево.а:=а;
  ELSE
    ДобавитьВершину(а, Дерево.Левая);
  END;
END;
END ДобавитьВершину;

```

Заметим, что процедура тащит за собой значение элемента массива через все свои активации до тех пор, пока не найдет правильного места. После этого все активации последовательно завершают свою работу и сворачиваются. Процедура **ВернутьМассив** также имеет хорошее рекурсивное решение. Проход по дереву – задача почти тривиальная, и её решение выглядит так:

```

IF Дерево.Левая#NIL THEN ВернутьМассив(Дерево.Левая); END;
IF Дерево.Правая#NIL THEN ВернутьМассив(Дерево.Правая);END;

```

При записи массива в дерево элементы массива записываются по ходу вглубь дерева. Очевидно, переписываться в массив они должны на обратном пути по дереву. Примерно вот так:

```

IF Дерево.Левая#NIL THEN ВернутьМассив(Дерево.Левая); END;
а[N]:=Дерево.а;N:=N+1;
IF Дерево.Правая#NIL THEN ВернутьМассив(Дерево.Правая);END;
а[N]:=Дерево.а;N:=N+1;

```

Слово «примерно» здесь употреблено, так как мы в общем-то серьезно ошиблись. При такой логике вершина дерева будет записана в массив дважды, при возврате по левой ветви и при возврате по правой. Исправимся:

```

PROCEDURE ВернутьМассив(Дерево:Указатель);
BEGIN
  IF Дерево.Левая#NIL THEN ВернутьМассив(Дерево.Левая); END;
  а[N]:=Дерево.а;N:=N+1;
  IF Дерево.Правая#NIL THEN ВернутьМассив(Дерево.Правая);END;
END ВернутьМассив;

```

А теперь полная сборка программы:

Листинг 6.9

MODULE Модуль;

```

IMPORT In, StdLog;
TYPE
  Указатель=POINTER TO Вершина;
  Вершина=RECORD
    a:INTEGER;
    Левая,Правая:Указатель;
  END;
VAR
  a:ARRAY 100 OF INTEGER;
  n:INTEGER;
PROCEDURE ДвоичнаяСортировка;
VAR
  Дерево:Указатель;
  k,N:INTEGER;
PROCEDURE ДобавитьВершину(a:INTEGER;Дерево:Указатель);
BEGIN
  IF a>Дерево.a THEN
    IF Дерево.Правая=NIL THEN
      NEW(Дерево.Правая); Дерево:= Дерево.Правая; Дерево.a:=a;
    ELSE
      ДобавитьВершину(a, Дерево.Правая);
    END;
  ELSE
    IF Дерево.Левая=NIL THEN
      NEW(Дерево.Левая); Дерево:=Дерево.Левая; Дерево.a:=a;
    ELSE
      ДобавитьВершину(a,Дерево.Левая);
    END;
  END;
END ДобавитьВершину;
PROCEDURE ВернутьМассив(Дерево:Указатель);
BEGIN
  IF Дерево.Левая#NIL THEN ВернутьМассив(Дерево.Левая); END;
  a[N]:=Дерево.a;N:=N+1;
  IF Дерево.Правая#NIL THEN ВернутьМассив(Дерево.Правая);END;
END ВернутьМассив;
BEGIN
  NEW(Дерево);
  Дерево.a:=a[0];
  FOR k:=1 TO n-1 DO
    ДобавитьВершину(a[k],Дерево);
  END;
  N:=0;
  ВернутьМассив(Дерево);

```

```
END ДвоичнаяСортировка;  
PROCEDURE Главная*;  
VAR  
    k:INTEGER;  
BEGIN  
    In.Open;  
    In.Int(n);  
    FOR k:=0 TO n-1 DO In.Int(a[k]);END;  
    ДвоичнаяСортировка;  
    FOR k:=0 TO n-1 DO StdLog.Int(a[k]); END;  
END Главная;  
END Модуль.
```

Замечания по оценке производительности. Двоичная сортировка требует не слишком много присвоений. Если массив состоит из n элементов, то потребуется $2n$ присвоений, n присвоений – для переброски массива в дерево и n – для обратной операции. Несложно оценить и количество сравнений. Предположим, что на некотором этапе дерево имеет максимальную глубину L . Для очередного элемента массива каждое сравнение спускает элемент на одну позицию вглубь дерева. Это означает, что при максимальной текущей глубине L возможно не более чем L сравнений. Отсюда следует, что двоичная сортировка тем выгоднее, чем меньше будет глубина строящегося дерева. А это, в свою очередь, возможно, если дерево будет иметь максимально возможное количество ветвлений. Что, в свою очередь, требует одинаковой длины всех поддеревьев одного уровня. В отношении массива последнее требование означает его максимальную хаотичность, если в массиве появляются частично упорядоченные участки, то это приводит к появлению длинных веток.

Задание для самостоятельной работы

Как обычно, решите самостоятельно вопрос об устойчивости сортировки.

Сортировка слияниями

Сортировка слияниями – это метод, позволяющий работать с очень большими массивами данных, настолько большими, что для их размещения недостаточно оперативной памяти. Очевидно, что при этом придется работать с магнитными носителями информации, скорость доступа к которым не слишком высока. Действительно, рассмотрим для примера сортировку Шелла. Её главная идея – обеспечить обмен данными, отстоящими друг от друга максимально далеко. Прямая индексация позволяет не задумываться о расстоянии между элементами массива. Необходимость чтения массы данных, для того чтобы добраться до далеко стоящего элемента, эту очевидную идею ускорения обращает в недостаток. Работа с файлами требует минимального «хождения» по файлу взад-вперед.

Очевидно, ограничение в возможности многократного «хождения» по массиву

ву данных потребует изменения стратегии поведения при однократном проходе. Здесь возможна, например, идея накопления какой-либо дополнительной информации. Однако, даже не зная, о какой информации может идти речь, разумно ожидать, что объем этой информации должен расти в зависимости от объема сортируемых данных. Это естественно приведет к серьезной потере памяти, а с быстрым ростом дополнительной памяти можно ожидать и потерь в скорости. Это тем более становится ясным, если учесть, что дополнительная информация также будет храниться в файлах.

Итак, идея генерирования вспомогательной информации несет в себе органический недостаток. Иная идея заключается в ограничении объема сортируемых данных. Чем исходный массив меньше, тем меньше объем работы по переброске элементов массива. Минимальный, нетривиальный массив – это массив из двух элементов. Его сортировка не требует никаких специальных методов, реализуется простым сравнением и простой перестановкой:

```
IF a[1]<a[2] THEN  
  c:=a[1]; a[1]:=a[2]; a[2]:=c;  
END;
```

Вместо индексов 1 и 2 можно подставить индексы k и $k+1$. Сказанное, конечно, не означает необходимости ограничения размера исходных массивов данных. Сказанное дает идею минимизации «хождений» по массиву. Поделим исходный массив на пары (это условная операция, не требующая никаких специальных усилий), затем в первом проходе выполним упорядочение в пределах пар. Если массив содержит нечетное количество элементов, то последний элемент составит тривиальный одноэлементный массив, который можно считать уже упорядоченным. Вторым проходом полученного массива упорядоченные пары специальной операцией слияния превращаются в упорядоченные четверки, третьим проходом четверки превращаются в упорядоченные подпоследовательности из 8 элементов и т. д. И на последнем шаге два больших подмассива сливаются в один. Существенно важна в этом процессе так называемая операция слияния. Рассмотрим её детально.

Операция слияния

Для удобства иллюстрации будем полагать, что сливаемые последовательности находятся в двух разных массивах, каждый из которых расположен на отдельной ленте. От начала каждой ленты с исходными данными пустим бегунок, движущийся пошагово от данного к данному. На каждом шаге слияния два бегунка показывают на два числа. Из этой пары в ячейку результирующей ленты записывается меньшее число. Если числа пары одинаковы, то число берется из ленты, взятой в качестве приоритетной по умолчанию. На следующих рисунках показаны три шага бегунка.

На первом шаге число взято с первой ленты, и её бегунок, соответственно, уходит на шаг вправо (рис. 6.2).

На втором шаге (рис. 6.3) меньшее число опять на первой ленте. Поэтому вто-

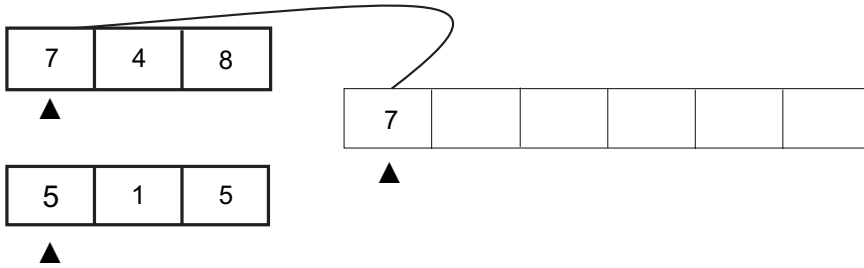


Рис. 6.2

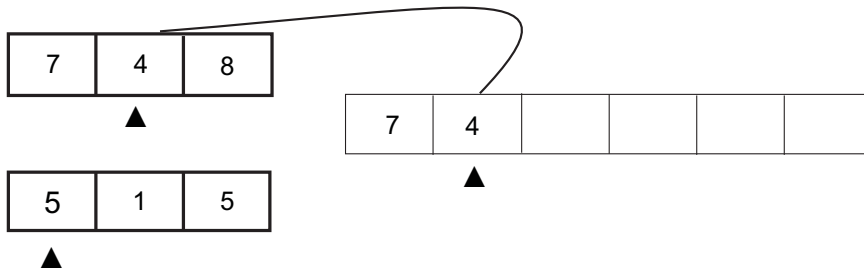


Рис. 6.3

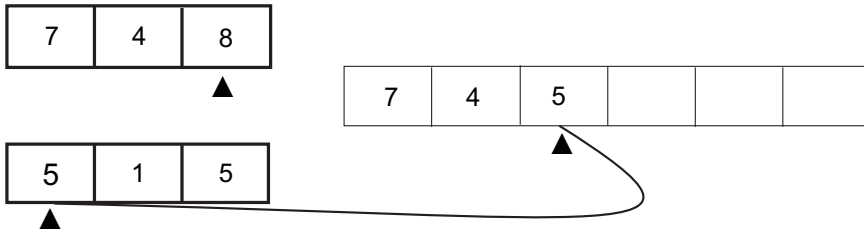


Рис. 6.4

рое число опять уходит с неё в результирующую ленту. И наконец, на третьем шаге (рис. 6.4) нижний бегунок показывает на меньшее число из двух возможных (8, 5). Поэтому на третьем шаге в ячейку ленты результата будет записано число 5.

Пример полной сортировки слиянием

Пусть исходный массив такой (5, 3, 2, 7, 8, 9, 1, 4, 4). Разобьем его на пары и пары упорядочим: (3, 5), (2, 7), (8, 9), (1, 4, 4).

Первый шаг слияния:

- (3, 5) + (2, 7) = (2, 3, 5, 7);
- (8, 9) + (1, 4, 4) = (1, 4, 4, 8, 9).

Второй шаг слияния:

- (2, 3, 5, 7) + (1, 4, 4, 8, 9) = (1, 2, 3, 4, 4, 5, 7, 8, 9).

В качестве носителя сортируемых чисел будем использовать массив, этого до-

статочно для демонстрации метода. Вместо двух массивов для лент источников и массива ленты результата используем два массива, которые будут меняться ролями: источник – результат. Начнем построение реализации.

Из первого массива во второй элементы должны сливаться с соответствующих отрезков. Соответствующие отрезки – это рядом расположенные отрезки. Длины сливаемых отрезков одинаковы. Длина сливаемого отрезка есть главная характеристика процесса. На каждом шаге длина должна удваиваться, и процесс завершается, когда длина сливаемого отрезка станет не меньше длины первого массива. Сказанное реализовано следующим фрагментом:

```
ДлинаОтрезка:=2;
WHILE ДлинаОтрезка<N DO
  (*Основной процесс*)
  ДлинаОтрезка:=ДлинаОтрезка*2;
END;
```

Основной процесс – это слияние пар отрезков. Операцию слияния оформим отдельной процедурой, здесь же опишем процесс прохода по отрезкам заданной длины.

```
WHILE ДлинаОтрезка<N DO
  k:=0;
  FOR i:=1 TO (N DIV ДлинаОтрезка)-1 BY 2 DO
    Слияние(ДлинаОтрезка*(i-1),ДлинаОтрезка*i,ДлинаОтрезка*i,
      ДлинаОтрезка*i+ ДлинаОтрезка);
  END;
END;
```

Параметр i определяет номер сливаемых отрезков. Левую и правую границы отрезков можно было бы рассчитать и в самой процедуре, но можно и так, как в нашей реализации, в момент вызова.

Достаточно нетривиальный смысл имеет величина k – инициализируемая до начала цикла **FOR**. Второй массив растет естественным образом, начиная от нулевого элемента, с шагом единица. И это так независимо от выбора элемента из первого массива. Это во-первых. Во-вторых, после слияния очередной пары отрезков из первого массива заполнение второго элементами следующей пары продолжается с той позиции, на которой процесс был завершён для предыдущей пары. Таким образом, индекс второго массива – это «сквозная» величина, проходящая через все вызовы слияния. Следовательно, индекс второго (а роль этого индекса как раз и играет величина k) должен быть глобальной величиной по отношению к процедуре слияния и инициализироваться до начала процесса очередной сессии слияний.

В цикле **FOR** выполняется слияние одинаковых по длине отрезков, но, возможно, массив не допускает точного разбиения, тогда по завершении цикла останется небольшой хвостик, о слиянии которого необходимо позаботиться особо. И по за-

вершении всех слияний длину отрезка можно увеличить вдвое. Полный фрагмент приобретает следующий вид:

```

ДлинаОтрезка:=2;
WHILE ДлинаОтрезка<N DO
  k:=0;
  FOR i:=1 TO (N DIV ДлинаОтрезка)-1 BY 2 DO
    Слияние(ДлинаОтрезка*(i-1),ДлинаОтрезка*i,ДлинаОтрезка*i,
      ДлинаОтрезка*i+ ДлинаОтрезка);
  END; (*FOR по i*)
  Слияние(ДлинаОтрезка*(i-1), ДлинаОтрезка*i, ДлинаОтрезка*i,N);
  Первый:=Второй;
  ДлинаОтрезка:=ДлинаОтрезка*2;
END;

```

Оператор **Первый:=Второй;** перебрасывает Второй массив обратно в Первый для повторения процесса на уже больших отрезках. Конечно, такая переброска требует серьёзных временных затрат. Дальше при построении алгоритма естественных слияний мы избавимся от этого недостатка.

Внешняя сторона организации процесса слияний ясна, можно заняться описанием процедуры слияния двух отрезков, при условии что их границы известны. Определим две величины – бегунки, чьи роли будут заключаться в обходе своих отрезков: **Бегунок1** пусть пробегает первый отрезок пары и **Бегунок2** – второй отрезок пары. Бегунки начинают свое движение от левых границ своих отрезков:

```

Бегунок1:=Левая1;Бегунок2:=Левая2;

```

В конечном итоге активация процедуры слияний должна обработать два отрезка, или, что то же самое, один отрезок двойной длины:

```

Бегунок1:=Левая1;Бегунок2:=Левая2;
FOR j:=1 TO ДлинаОтрезка*2 DO
  k:=k+1;
END;

```

На каждом шаге записанного выше цикла во второй массив попадает один элемент либо из первого, либо из второго отрезка. Мы, однако, не пользуемся для построения процедуры терминами отрезков. Мы используем **Бегунки**, следовательно, можно сказать так: во второй массив попадает либо элемент первого массива, на который указывает **Бегунок1**, либо элемент, на который указывает **Бегунок2**. Сортировка требует взятия меньшего из двух, поэтому появляется следующий условный оператор:

```

FOR j:=1 TO ДлинаОтрезка*2 DO
  IF Первый[Бегунок1]<Первый[Бегунок2] THEN
    (*Переносится элемент первого бегунка*)
  ELSE

```

```

        (*Переносится элемент второго бегунка*)
    END;
    k:=k+1;
END;

```

Перед тем как переносить элемент массива, на который указывает выбранный **Бегунок**, необходимо проверить, не достигнута ли правая граница отрезка. Если граница достигнута, то элементов в отрезке уже просто нет, и тогда элемент для переноса необходимо взять от другого **Бегунка**. Ниже – полный фрагмент такого сложного условия:

```

IF Первый[Бегунок1]<Первый[Бегунок2] THEN
    IF Бегунок1<Правая1 THEN
        Второй[k]:=Первый[Бегунок1];
        Бегунок1:= Бегунок1+1;
    ELSE
        Второй[k]:=Первый[Бегунок2];
        Бегунок2:=Бегунок2+1;
    END;
ELSE
    IF Бегунок2<Правая2 THEN
        Второй[k]:=Первый[Бегунок2];
        Бегунок2:= Бегунок2+1;
    ELSE
        Второй[k]:=Первый[Бегунок1];
        Бегунок1:= Бегунок1+1;
    END;
END;

```

Осталось собрать программу полностью.

Листинг 6. 10

```

MODULE Модуль;
    IMPORT In, StdLog;
    PROCEDURE Сортировка*;
    VAR
        Первый,Второй:ARRAY 100 OF INTEGER;
        N,i,c,ДлинаОтрезка,k,p:INTEGER;
    PROCEDURE Слияние(Левая1,Правая1,Левая2,Правая2:INTEGER);
    VAR
        Бегунок1, Бегунок2,j:INTEGER;
    BEGIN
        Бегунок1:=Левая1;Бегунок2:=Левая2;
        FOR j:=1 TO ДлинаОтрезка*2 DO
            IF Первый[Бегунок1]<Первый[Бегунок2] THEN
                IF Бегунок1<Правая1 THEN

```

```

        Второй[k]:=Первый[Бегунок1];
        Бегунок1:= Бегунок1+1;
    ELSE
        Второй[k]:=Первый[Бегунок2];
        Бегунок2:=Бегунок2+1;
    END;
ELSE
    IF Бегунок2<Правая2 THEN
        Второй[k]:=Первый[Бегунок2];
        Бегунок2:= Бегунок2+1;
    ELSE
        Второй[k]:=Первый[Бегунок1];
        Бегунок1:= Бегунок1+1;
    END;
END; (*Главное сравнение*)
k:=k+1;
END; (*FOR по j*)
END Слияние;
BEGIN
    In.Open;
    N:=-1;
    REPEAT
        N:=N+1;
        In.Int(Первый[N]);
        Второй[N]:=0;
    UNTIL ~In.Done;
    FOR i:=0 TO N-2 BY 2 DO
        IF Первый[i]>Первый[i+1] THEN
            c:=Первый[i];Первый[i]:=Первый[i+1];Первый[i+1]:=c;
        END;
    END;
    (*Начало главного цикла*)
    ДлинаОтрезка:=2;
    WHILE ДлинаОтрезка<N DO
        k:=0;
        FOR i:=1 TO (N DIV ДлинаОтрезка)-1 BY 2 DO
            Слияние(ДлинаОтрезка*(i-1),ДлинаОтрезка*i,ДлинаОтрезка*i,
                    ДлинаОтрезка*i+ ДлинаОтрезка);
        END; (*FOR по i*)
        Слияние(ДлинаОтрезка*(i-1), ДлинаОтрезка*i, ДлинаОтрезка*i,N);
        Первый:=Второй;
        ДлинаОтрезка:=ДлинаОтрезка*2;
    END;
    FOR i:=0 TO N-1 DO

```

```
StdLog.Int(Первый[i]);  
END;  
END Сортировка;  
END Модуль.
```

Рассмотренный алгоритм можно существенно улучшить. Заметим, что длина сливаемых подпоследовательностей выражается константами (степени двойки). Это так, быть может, за исключением последнего отрезка – хвостика массива.

Заметим, что исходная последовательность уже состоит из некоторого количества отсортированных подпоследовательностей. Это, очевидно, всегда так. В худшем случае, для последовательности убывающих чисел, можно считать, что каждая отсортированная подпоследовательность первого шага состоит из одного элемента. Но такая ситуация столь же маловероятна, как и последовательность, изначально отсортированная. Скорее всего, в исходной последовательности будут присутствовать правильно отсортированные подпоследовательности разной длины, и чем исходная длиннее, тем вероятней наличие длинных «готовых» подпоследовательностей. Такие подпоследовательности будем называть естественными. Например, в массиве 3, 6, 7, 8, 4, 3, 2, 5 можно выделить следующие естественные подпоследовательности: (3, 6, 7, 8), (4), (3), (2, 5). Далее вместо неуклюжего термина «подпоследовательность» будем использовать термин «серия».

Интуитивно понятно, что сливать естественные серии выгоднее. Выгоду легко обосновать и математически. Действительно, пусть сливаются две последовательности, каждая из которых состоит из N серий. Тогда общее количество сливаемых серий $2N$. А последовательность-результат будет состоять только из N серий, то есть на каждом шаге слияния количество серий будет уменьшаться минимум вдвое. А скорее всего, еще быстрее. Во всяком случае, количество операций можно оценить величиной $N \log(N)$.

Но выигрыш в скорости достигается за счет увеличения сложности процесса. Сложность процесса обусловлена тем, что мы не можем вычислить конец естественной серии какой-либо точной формулой. Поэтому необходимы дополнительные проходы, для того чтобы поймать конец очередной серии. Необходимость дополнительных проходов выливается в необходимость дополнительных сравнений, это конечно же влечет за собой потери производительности. Но потери не должны быть слишком велики. Необходимо помнить, конечно же, что сортировка слияниями работает не с массивами, а с файлами, а операции пересылки между файлами требуют значительно больше времени, чем операции пересылки между элементами массива.

Начнем построение механизма естественных слияний. Главный пункт анализа – это границы естественных серий. Учет границ можно выполнить двумя способами. Во-первых, для каждого слияния добавить еще один проход специально с целью построения серий. Во-вторых, возможно создать еще один массив специально для хранения информации о границах естественных серий. Это избавит от необходимости дополнительных проходов, но добавит проблем с обработкой этого дополнительного массива, и кроме того, существенно усложнит логику. В нашем реше-

нии для этой цели будет использован дополнительный проход.

Задание для самостоятельной работы

После того как разберете реализацию с дополнительным проходом, постройте реализацию, в которой для границ серий использовался бы дополнительный массив.

Операции слияния выполняются до тех пор, пока в исходном массиве можно выделить хотя бы две естественные серии. Сигналом к завершению будет ситуация, в которой не удалось выделить вторую серию. Ниже записан главный цикл:

```
flag:=TRUE;
WHILE flag DO
  flag:=FALSE;
  (*Процесс построения серий*)
END;
```

Если в процессе построения серий будет обнаружена хотя бы одна пара, то это будет означать, что, возможно, процесс сортировки не завершен, тогда значению флага будет присвоено значение «Истина» и главный цикл начнет следующую итерацию. На каждой итерации может быть обнаружено несколько серий, поэтому для поиска серий необходимо пройти весь массив. Получаем следующее усложнение:

```
flag:=TRUE;
WHILE flag DO
  flag:=FALSE;
  k:=0;

  WHILE k<=N-1 DO
    (*Построение очередной пары серий и их слияние*)
  END;
END;
```

N – это количество элементов сортируемого массива, массив нумеруется с нуля. Шаг итерации внутреннего цикла заключается в построении именно пары серий. Построение серии заключается в определении её левой и правой границ. Введем следующие обозначения: **Левая1**, **Правая1** – левая и правая границы первой серии, **Левая2**, **Правая2** – левая и правая границы второй серии. Сейчас мы исходим из предположения, что для каждой серии найдется пара, но в общем-то это не обязательно. На некоторой итерации внутреннего цикла может оказаться, что **Правая1**= $N-1$ и для второй серии элементов в массиве уже нет. Еще один важный момент – это завершение внутреннего цикла. Условие завершения очевидно, но нужно грамотно распорядиться изменением величины k . После каждой итерации если:

- удалось построить обе серии, тогда $k=\text{Правая2}+1$;
- удалось построить только одну серию, тогда $k=\text{Правая1}+1$.

Кстати, и процедура слияния самым серьезным образом зависит от успешности построения парной серии. Поэтому наши действия будут следующими: строим первую серию. Эта операция будет всегда успешна, в крайнем случае, первой серией будет весь массив. Затем проверяем, совпадает ли правая граница первой серии с номером последнего элемента массива, если нет, то в массиве есть элементы, из которых гарантированно можно составить вторую серию. Запишем фрагмент программы:

```

WHILE flag DO
  flag:=FALSE;
  k:=0;
  WHILE k<=N-1 DO
    (*Поиск двух отрезков*)
    Левая1:=k;Правая1:=Левая1;
    WHILE (Правая1<N-1) & (a[Правая1]<=a[Правая1+1]) DO
      Правая1:=Правая1+1;
    END;
    IF Правая1<N-1 THEN (*Поиск второго отрезка*)
      Левая2:=Правая1+1;Правая2:=Левая2;
      WHILE (Правая2<N-1) & (a[Правая2]<=a[Правая2+1]) DO
        Правая2:=Правая2+1;
      END;
      k:=Правая2+1;
      (*Их слияние*)
    ELSE (*Второй отрезок не найден*)
      k:=Правая1+1;
    END;
  END;
END;

```

Поясним процедуру построения серий. На каждой итерации внутреннего цикла величина **k** есть номер элемента, с которого можно строить новые серии. Поэтому левая граница первой серии **Левая1:=k**, затем выполняется движение по массиву до тех пор, пока очередное число меньше следующего, или пока не будет достигнут конец массива. Это движение выполняет следующий код:

```

Левая1:=k;Правая1:=Левая1;
WHILE (Правая1<N-1) & (a[Правая1]<=a[Правая1+1]) DO
  Правая1:=Правая1+1;
END;

```

Если выход из цикла произошел не из-за достижения границы массива, то совершенно аналогичный фрагмент выполняет построение второй серии. Построение серий мы пояснили, сейчас разберем процедуру слияния. Заметим, что их две. Но прежде выясним, что произойдет после всех слияний. Слияния выполняются из массива **a[]** в массив **b[]**. После чего, естественно, новые данные необходимо

вернуть в массив $a[]$ и потерять на этом очень много времени. Для программы, иллюстрирующей метод, это, быть может, и не очень большая беда. Но если с проблемой можно справиться малыми силами, то лучше это сделать. А проблема действительно решается очень легко, если массивы объявить динамическими:

```
a,b,c:POINTER TO ARRAY 100 OF INTEGER;
```

В тексте массивы $a[]$ и $b[]$ созданы для хранения данных, указатель $c[]$ играет вспомогательную роль, и к нему процедуру **NEW** не надо применять. При таком выборе структур данных последовательность операторов присваивания

```
c:=a; a:=b;b:=c;
```

выполняет переброску указателей с одной области на другую, сами данные остаются на месте. А сейчас вернемся к проблеме слияния.

Была построена только одна серия. Это более простой случай, поэтому начнем с него. Единственность серии означает, что часть элементов массива $a[]$ уже переброшена в массив $b[]$ и в массиве $a[]$ остались неслитыми только элементы этой последней серии, пронумерованные с **Левая1** по **N-1**. Эти элементы можно перекинуть в $b[]$ простым циклом:

```
FOR t:=Левая1 TO N-1 DO
  b[t]:=a[t];
END;
```

Было построено две серии. Это более сложный случай. Элементы в массив $b[]$ перебрасываются из двух серий, и решение о том, из какой серии на данном шаге брать элемент, принимается в зависимости от целого ряда условий. Перечислим их.

1. Если элементы обеих серий выбраны, то работа цикла слияния должна быть завершена.
2. Если элементы первой серии полностью выбраны, то необходимо взять очередной элемент из второй серии.
3. Если элементы второй серии полностью выбраны, то необходимо взять очередной элемент из первой серии.
4. Если очередной элемент первой серии меньше очередного элемента второй серии, то необходимо взять элемент первой серии.
5. Если очередной элемент второй серии меньше очередного элемента первой серии, то необходимо взять очередной элемент второй серии.

Если вы внимательно вчитаетесь в описанные условия, то сможете увидеть, что эти условия взаимоисключающие. Это дает возможность легко описать их как набор альтернатив в цикле **LOOP**:

```
t:=Левая1;
LOOP
  IF t>Правая2 THEN EXIT
  ELSIF Левая1>Правая1 THEN b[t]:=a[Левая2];Левая2:=Левая2+1;
  ELSIF Левая2>Правая2 THEN b[t]:=a[Левая1];Левая1:=Левая1+1;
```

```

    ELSIF a[Левая1]>a[Левая2] THEN b[t]:=a[Левая2];Левая2:=Левая2+1;
    ELSE b[t]:=a[Левая1];Левая1:=Левая1+1;
  END;
  t:=t+1;
END;

```

Цикл LOOP здесь очень полезен, так как позволяет проверку всех условий возложить на одну конструкцию IF. Полученный условный оператор несколько велик, но его логика проста и прозрачна. На этом мы завершаем разбор задачи естественного слияния и запишем полный листинг.

Листинг 6.11

```

MODULE Модуль;
  IMPORT In, StdLog;
  PROCEDURE ЕстественныеСлияния*;
  VAR
    a,b,c:POINTER TO ARRAY 100 OF INTEGER;
    k,N,i,Левая1,Правая1,Левая2,Правая2,t1,t2,t:INTEGER;
    flag:BOOLEAN;
  BEGIN
    NEW(a);NEW(b);
    In.Open;
    N:=-1;
    REPEAT
      N:=N+1;
      In.Int(a[N]);
    UNTIL ~In.Done;
    flag:=TRUE;
    WHILE flag DO
      flag:=FALSE;
      k:=0;
      WHILE k<=N-1 DO
        (*Поиск двух отрезков*)
        Левая1:=k;Правая1:=Левая1;
        WHILE (Правая1<N-1) & (a[Правая1]<=a[Правая1+1]) DO
          Правая1:=Правая1+1;
        END;
        IF Правая1<N-1 THEN (*Поиск второго отрезка*)
          Левая2:=Правая1+1;Правая2:=Левая2;
          WHILE (Правая2<N-1) & (a[Правая2]<=a[Правая2+1]) DO
            Правая2:=Правая2+1;
          END;
          k:=Правая2+1;
          (*Их слияние*)
          flag:=TRUE;
        END IF;
      END WHILE;
    END WHILE;
  END PROCEDURE;

```

```

t:=Левая1;
LOOP
  IF t>Правая2 THEN EXIT
  ELSIF Левая1>Правая1 THEN b[t]:=a[Левая2];Левая2:=Левая2+1;
  ELSIF Левая2>Правая2 THEN b[t]:=a[Левая1]; Левая1:=Левая1+1;
  ELSIF a[Левая1]>a[Левая2] THEN b[t]:=a[Левая2];Левая2:=Левая2+1;
  ELSE b[t]:=a[Левая1];Левая1:=Левая1+1;
  END;
  t:=t+1;
END;
ELSE (*Второй отрезок не найден*)
  k:=Правая1+1;
  FOR t:=Левая1 TO N-1 DO
    b[t]:=a[t];
  END;
END; (*Второй отрезок*)
END;
(*Обмен*)
c:=a; a:=b;b:=c;
END;
FOR i:=0 TO N-1 DO
  StdLog.Int(a[i]);
END;
StdLog.Ln;
END ЕстественныеСлияния;
END Модуль.

```

Замечание о производительности. Главное преимущество алгоритма – отнюдь не в высокой производительности. Область его применения – там, где другие алгоритмы не работают из-за слишком больших объемов данных. Конечно же, речь не идет о слиянии массивов. Если сливаемые массивы можно разместить в памяти компьютера, то, следовательно, эти массивы можно объединить в один массив и применить к нему, например, быструю сортировку или сортировку Шелла, в общем любую, имеющую высокую производительность. Безусловное преимущество нашего последнего метода проявится, если данные хранятся в очень большом файле, настолько большом, что использование оперативной памяти становится невозможным.

Подведем итог. В тексте главы описаны только несколько вариантов сортировки, в действительности их существенно больше. Но все разнообразие сортировок может быть оценено двумя свойствами: это быстродействие и устойчивость. И кроме того, иногда бывает необходимо учитывать природу сортируемого массива. Например, способ его хранения и размер. Какие проблемы создает такая зависимость, мы видели при анализе сортировки слиянием. Поэтому при выборе метода необходимо тщательно проанализировать характер сортируемых данных и взять подходящий метод именно для вашего случая.

Комбинаторные задачи

Общая постановка задачи	205
Оптимизация перебора	207
Связь комбинаторики с алгоритмами на графах	209
Основные комбинаторные задачи	210

Общая постановка задачи

В общем виде комбинаторную задачу можно сформулировать так: дано некоторое множество. Необходимо найти комбинацию элементов этого множества, удовлетворяющую заданным условиям. Правила построения «комбинаций» могут сильно отличаться. Слово комбинация взято в кавычки вот по какой причине – интуитивно понятно, что это такое, но в математике такой термин не используется, в математике вместо одного термина используются несколько, точным значением которых вы можете поинтересоваться в соответствующем приложении. Способы построения комбинаций могут быть разные. Например, перестановки – комбинации, в каждой из которых участвуют все элементы множества. Это могут быть сочетания без повторений – комбинации, в которых участвует часть элементов множества, причем их порядок не играет значения. Если правило построения известно, то все, что нужно, – это алгоритм построения комбинаций и проверка каждой из них на предмет соответствия заданным условиям.

Задач комбинаторного типа очень много, вот несколько общеизвестных примеров:

Разбиение кучи камней. Дана куча камней, необходимо раскидать её на две по возможности равного веса. Решается задача перебором всех возможных разбиений на две кучи и поиском среди них искомого.

Размещение восьми ферзей. Как расставить 8 ферзей на шахматной доске, так чтобы ни один из них не был под боем. Задача имеет решение, и даже не одно. Решение можно найти довольно простой организацией перебора. Например, так: поставим первого ферзя на первую горизонталь, затем на второй горизонтали поищем небитую позицию для второго ферзя, затем на третьей – для третьего и т. д. Если для очередного ферзя небитой позиции не найдется, то вернемся к предыдущему ферзю и изменим его расположение.

Построение магического квадрата. Построить числовой квадрат, такой что сумма его элементов на всех горизонталях и всех вертикалях и главных диагоналях одинакова. Если сторона квадрата N , то квадрат строится на числах от 1 до N^2 перебором всех возможных вариантов размещения чисел в клетках квадрата.

Прежде чем двигаться дальше, заметим, что иногда задача, кажущаяся переборной, таковой на самом деле не является. Например, задачу поиска кратчайших путей на графе можно решать полным перебором всех путей и сравнением их длин. Но для такого рода задач существуют довольно мощные алгоритмы, позволяющие находить решение быстрее, чем перебором. Эти методы рассмотрены в главе, посвященной алгоритмам на графах.

Бывает и наоборот. Может показаться, что задача имеет простой алгоритм решения. Но простой алгоритм оказывается не более чем частным случаем, а задачу все же приходится решать полным перебором. В качестве примера рассмотрим задачу о разбиении кучи камней. Для некоторых частных случаев возможен следующий алгоритм:

Упорядочим камни исходной кучи в порядке возрастания

Пока в исходной куче есть хотя бы один камень
 Возьмем из исходной кучи самый тяжелый камень
 Если левая куча тяжелее правой
 То положим взятый камень в правую кучу
 Иначе положим взятый камень в левую кучу

Логика алгоритма проста и прозрачна. Отработаем следующий тест. Исходная куча: (1, 2, 3, 4, 5, 6). Ниже в таблице записана последовательность операций:

Таблица 7.1

Номер шага	Исходная куча	Левая куча	Правая куча
0	1, 2, 3, 4, 5, 6		
1	1, 2, 3, 4, 5	6	
2	1, 2, 3, 4	6	5
3	1, 2, 3	6	4, 5
4	1, 2	3, 6	4, 5
5	1	2, 3, 6	4, 5
6		2, 3, 6	1, 4, 5

Легко видеть, что полученное решение оптимально. Можно придумать еще много тестов, для которых данный алгоритм получит наилучшее решение. Алгоритм необыкновенно быстр и кажется очень разумным. Но тем не менее он не годится для общего случая. Легко придумать тест, для которого алгоритм решения не обнаружит.

Задания для самостоятельной работы

1. Попробуйте придумать такой тест.
2. Попробуйте дать точное описание класса тестов, для которого данный алгоритм действительно дает решение.

Чем отличаются задачи полного перебора от задач, в которых от перебора все же можно избавиться?

Ответ будет таков. На множестве данных переборной задачи нельзя найти никакой закономерности. Такова ситуация в задаче о двух кучах камней. Про веса камней в исходной куче нельзя сказать ничего определенного. Они могут быть какими угодно. Искусственный прием по упорядочению, как кажется, создает такую закономерность, на множестве возникает порядок, но надо понимать, что такой порядок возможен на любом множестве и, следовательно, этот порядок не несет в себе никакой информации. Почему возможен волновой алгоритм и алгоритм Дейкстры на графах. Граф уже является упорядоченной структурой. Вершины графа не являются равнозначными по отношению друг к другу. Некоторые из них соединены ребрами, а некоторые нет. И эти ребра (дуги) несут дополнительную информацию, которая и делает возможными непереборные алгоритмы.

Проблема комбинаторного взрыва. В задачах комбинаторной природы есть одно качество, создающее большие проблемы. Это слишком большое количество исследуемых вариантов. Рассмотрим примеры.

Задача о ферзях. Всего ферзей 8. Каждого из них можно разместить на доске 8 способами (при условии, что для каждого ферзя будем считать допустимым расположение только на одной горизонтали). Итого общее количество вариантов $8^8 = 16777216$. Для ручного анализа многовато. Для машинного, впрочем, реально. Но если мы перейдем от доски 8×8 к доске большего размера, то количество вариантов очень быстро станет велико и для очень мощного компьютера.

Задача о разбиении кучи камней на две. Процедура перебора всех возможных разбиений можно построить как процесс построения сочетаний, сначала по одному элементу, затем по два и т. д. Одна из куч – это получаемое сочетание, вторая – это оставшиеся камни. Отсюда понятно, что количество всех возможных разбиений одной кучи на две равно количеству всех возможных сочетаний без повторений из N элементов, если N – это количество камней в исходной куче. А количество сочетаний без повторений для множества из N элементов равно 2^N . При довольно небольшом значении N , например 100, количество сочетаний быстро становится астрономическим числом.

Таким образом, решение комбинаторной задачи может существовать и даже быть записано достаточно простой программой, но количество возможных комбинаций исходных данных делает запуск такой программы делом совершенно бессмысленным. Поэтому в решении комбинаторных задач часто используют так называемые эвристические алгоритмы. Это алгоритмы, дающие неплохое решение за приемлемое время. Но эвристические алгоритмы, к сожалению, ничего не гарантируют, поэтому поиск эвристического решения всегда означает готовность пожертвовать качеством решения для того, чтобы получить хотя бы что-нибудь. В нашей книге есть глава, посвященная принятию решений, там детально проанализирована ситуация, в которой отказаться от решения нельзя, а получить точное решение принципиально невозможно. и там же можно посмотреть определение эвристического алгоритма.

Оптимизация перебора

Иногда ситуация не так печальна. Например, в задаче о ферзях установка ферзя на любую клетку доски делает часть полей доски битыми. Рассматривать варианты установки последующих ферзей на битые поля не имеет смысла. Поэтому если программист придумает механизм установки метки на битые поля, это избавит его от рассмотрения большого количества вариантов. Это же верно и в задаче о разбиении кучи камней. Пусть дана следующая куча (10000, 1, 2, 3, 4, 5), ясно, что очень большой камень, положенный, например, в правую кучу, делает бессмысленным все варианты, кроме одного (понятно, какого). Эту ситуацию можно обобщить следующим правилом: если вес одной из куч больше, чем суммарный вес оставшихся камней и другой кучи, то все оставшиеся камни можно, не задумываясь, класть в ту другую, более легкую кучу.

Таким образом, возможны приемы и методы, позволяющие выделить специфические ситуации, из которых дальнейший ход перебора можно ограничить. В подтверждение – еще одна классическая задача.

Обход конем шахматной доски. Шахматный конь, начиная свой путь из одного угла шахматной доски, должен обойти всю доску, побывав при этом на каждом поле только по одному разу, и завершить свой путь на том же поле, с которого он начал движение. Задача уже детально проанализирована в главе о рекурсии. Сейчас мы её используем только как пример задачи перебора.

Перебор можно организовать, отмечая поля, на которых конь уже побывал. Тогда если конь находится на конкретном поле, то все возможные продолжения – это поля, на которых нет метки и на которые конь может выполнить ход с данного поля. Алгоритм, построенный так, учитывает условие, в котором сказано, что на каждом поле конь может побывать только один раз. Здесь уже заложено ограничение перебора так установка коня на определенное поле ограничивает выбор полей для следующего хода.

Кроме того, заметим, что поля доски представляют собой строго упорядоченное множество – квадрат. Это наводит на мысль о дальнейшей оптимизации. Поля доски в отношении хода коня не равнозначны. С некоторых полей (центральных) можно сделать много ходов, с некоторых (боковых и угловых) – меньше. Поэтому для задачи обхода доски конем работает правило, называемое правилом Варнсдорфа. Оно заключается в следующем: если с текущего поля можно выполнить несколько ходов, то из них выбирается ход, имеющий наименьшее количество продолжений. Правило выделяет из общего дерева вариантов слабо ветвящееся поддерево в надежде на то, что в этом поддереве можно найти решение. Вероятность такого исхода действительно ненулевая. Кроме того, если надежда себя не оправдывает, мы ничего не теряем, так как правило не отсекает ни одной ветви большого дерева, оно только переопределяет порядок их обхода.

Задание для самостоятельной работы

В нашей реализации обхода коня правило Варнсдорфа не использовалось. Попробуйте модифицировать программу с учетом этого правила и сравните эффективность работы реализации с правилом и без правила.

Задача о подборе суммы. Пусть даны произвольное множество чисел и число A . Необходимо найти подмножество такое, что сумма его элементов меньше всего отличается от числа A и при этом не превосходит самого числа A .

Решение этой задачи заключается в сравнении уже найденной суммы и полученной из очередного подмножества, для чего, естественно, необходимо организовать процесс перебора всех возможных подмножеств. Процесс перебора можно строить по-разному. Пусть мы смогли найти метод со следующим свойством: каждое новое подмножество получается из уже известного добавлением еще одного числа. Если такой метод удастся получить, то, несмотря на полную неопределенность с исходными данными, отсеечение вариантов все же возможно. А именно сработает следующее простое правило: назовем каждое подмножество, из которого было получено большее подмножество (возможно, за несколько шагов), порождающим.

Получение порождающего подмножества есть повод для работы следующего шага алгоритма. А если очередное подмножество окажется уже больше, чем число A , то назовем такое подмножество тупиковым. Естественно, что все подмножества, для которых тупиковое могло бы быть порождающим, выпадают из рассмотрения. Тупиковое множество не есть порождающее, хотя для полного перебора необходимо было бы рассмотреть и возможные тупики.

Задание для самостоятельной работы

Найдите для задачи о подборе суммы метод со следующими свойствами:

- каждое подмножество получено из некоторого другого добавлением одного числа;
- метод обеспечивает полный перебор всех возможных вариантов.

Связь комбинаторики с алгоритмами на графах

Между алгоритмами на графах и комбинаторными существует глубокая внутренняя связь. Любой перебор можно представить как дерево вариантов. Это не всегда бывает эффективно, но иногда полезно для понимания происходящего. Например, в задаче об обходе шахматной доски конем все пути естественным образом выстраиваются в виде дерева путей, некоторые из которых заканчиваются тупиковыми позициями, то есть позициями, из которых ход невозможен, а некоторые ветви имеют длину 64, что означает полный обход доски с соблюдением условий задачи.

В виде дерева можно выстроить переборное множество из задачи о подборе суммы, при условии что вы найдете метод, описанный в задании для самостоятельной работы. Связь между комбинаторикой и графами видна и на некоторых числовых соотношениях. Например, количество конечных вершин двоичного дерева глубины N равно 2^N , и точно этой величине равно количество всех возможных сочетаний без повторов из множества в N элементов.

Связь между графами и комбинаторикой полезна не только для более полного понимания какой-либо задачи. Представление переборного множества в виде графа дает возможность использования мощного аппарата теории графов и множества разработанных в этой теории алгоритмов, не имеющих переборной природы. Простой пример. Пусть требуется найти кратчайший путь в городе между двумя пунктами. Кратчайшим путем будем считать путь с наименьшим количеством пересадок на общественном транспорте. Участки пешего пути будем считать фактором несущественным. Если множество остановок общественного транспорта представить как граф, вершины которого – остановки, а ребра – маршруты, то мы получим задачу, решаемую волновым алгоритмом, а если стоимость проезда на разных видах транспорта разная и нас интересует не количество пересадок, а стоимость пути, то алгоритмом Дейкстры.

Задание для самостоятельной работы

Дано множество из некоторого количества символов (можно небольшого). Постройте все возможные сочетания и постройте на этом множестве дерево, вершины которого – сочетания.

Основные комбинаторные задачи

Множество комбинаторных задач практически необъятно, но существует несколько задач чистого перебора, к которым в конечном итоге очень многое сводится. Мы из этого набора рассмотрим задачи построения сочетаний с повторениями и без, задачу построения перестановок и задачу построения размещений.

Задача получения перестановок на множестве из N элементов

Ниже приведены два решения: рекурсивное и нерекурсивное. У обоих решений одна идея. Определим для массива понятие циклического сдвига как последовательного перемещения элементов, при котором последний элемент встает на место первого, а все остальные смещаются на место своего соседа справа. Такой сдвиг можно описать следующим программным фрагментом:

```
Значение:=Элемент[M];  
FOR k:=M TO 1 BY -1 DO  
    Элемент[k]:=Элемент[k-1];  
END;  
Элемент[0]:=Значение;
```

Здесь M – номер последнего элемента массива. Ясно, что для массива из N элементов можно выполнить $N-1$ сдвиг, каждый из которых дает новую перестановку. Исходное положение также является перестановкой. Поэтому для N элементов имеем N перестановок. Из каждой перестановки длины N можно выделить подмассив длины $N-1$, для которого также возможны сдвиги, и, соответственно, массив длины $N-1$ порождает $N-1$ перестановку. Каждая перестановка длины $N-1$ порождает... В общем и т. д. В итоге получаем: $N*(N-1)*(N-2)*...*2=N!$ перестановок. Процесс получения перестановок, очевидно, будет заключаться в выполнении двух действий:

1. Определение подмассива, на котором необходимо выполнить циклический сдвиг.
2. Выполнение сдвига.

Вторая операция уже построена выше. В отношении первого действия заметим следующее: для подмассива, состоящего из k элементов, можно выполнить k циклических сдвигов. Отсюда следует, что для каждого подмассива необходимо завести счетчик сдвигов. В нерекурсивном варианте это может быть массив, элементы которого увеличиваются на 1 при каждом сдвиге, в рекурсивном варианте это может быть локальная переменная рекурсивной процедуры.

Рекурсивное решение. Как известно, простая форма организации рекурсии содержит условие продолжения рекурсивных активаций и выполнение некоторых действий для очевидных ситуаций. В описанном выше алгоритме каждая рекурсивная активация строит перестановки для «своего подмножества». «Свое

подмножество» можно определить как подмножество длины **ДлинаПерестановки** для активации с номером **ДлинаПерестановки**. Здесь сразу уточним, что **ДлинаПерестановки** это номер последнего элемента, поэтому если **ДлинаПерестановки=1**, это означает, что в перестановке участвуют два элемента, памятуя об этом уточнении, мы все же будем продолжать пользоваться термином **ДлинаПерестановки**.

Для подмножества длины **ДлинаПерестановки** можно выполнить **ДлинаПерестановки** циклических сдвигов. Поэтому тело рекурсивной процедуры с номером **ДлинаПерестановки** – это тело цикла в **ДлинаПерестановки** шагов.

```
i:=0;
WHILE i< ДлинаПерестановки DO
  (*Построение очередной перестановки*)
  i:=i+1;
END;
```

В теле этого цикла строится очередная перестановка:

```
i:=0;
WHILE i< ДлинаПерестановки DO
  (*Запоминаем последний элемент подмножества*)
  c:=Элемент[ДлинаПерестановки - 1];
  j:= ДлинаПерестановки - 1;
  (*Выполняем сдвиг подмножества к концу, замещая последний элемент*)
  WHILE j>0 DO
    Элемент[j]:= Элемент[j-1];
    j:=j-1;
  END;
  (*Запомненный последний записываем в нулевой*)
  Элемент[0]:=c;
  i:=i+1;
END;
```

После построения перестановки её требуется распечатать, но здесь появляется технический нюанс. Перестановки повторяются. Ниже, пример работы алгоритма на множестве из трех элементов (a, b, c): abc, acb, **abc**, cab, cba, **cab**, bca, bac, **bca**, **abc**. Жирным шрифтом отмечены повторы. На таком множестве работают только два рекурсивных вызова. Повторы **abc**, **cab**, **bca** имеют место на втором вызове, видно, что повтор – это всегда последняя перестановка для своего подмножества. Следовательно, распечатывать можно только перестановки с номером $i < \text{ДлинаПерестановки} - 1$. В тексте печать реализована в процедуре **Печать**, и отвечает за печать фрагмент, записанный в листинге ниже:

```
IF i<ДлинаПерестановки-1 THEN
  Печать;
END;
```

И наконец, последний момент – это выполнение следующей активации. Оче-

редная активация возможна только для подмножества с номером **ДлинаПерестановки**>0. Вспомним, что **ДлинаПерестановки** – это не длина подмножества, а номер последнего элемента. Поэтому условие **ДлинаПерестановки**>0 означает, что построение перестановок имеет смысл только для множества, в котором не менее двух элементов. Фрагмент, отвечающий за новую активацию, записан в листинге:

```
IF ДлинаПерестановки >0 THEN
Перестановка(ДлинаПерестановки -1);
END;
```

Ниже – листинг полного решения:

Листинг 7.1

```
MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE ПостроениеПерестановок*;
VAR
Элемент:ARRAY 10 OF INTEGER;
ДлинаПерестановки:INTEGER;
PROCEDURE Печать;
VAR
k:INTEGER;
BEGIN
  FOR k:=0 TO ДлинаПерестановки-1 DO
    StdLog.Int(Элемент[k]);
  END;
  StdLog.Ln;
END Печать;
PROCEDURE Перестановка(ДлинаПерестановки:INTEGER);
VAR
i,j,c:INTEGER;
BEGIN
  i:=0;
  WHILE i< ДлинаПерестановки DO
    c:= Элемент[ДлинаПерестановки-1];
    j:=ДлинаПерестановки-1;
    WHILE j>0 DO
      Элемент[j]:= Элемент[j-1];
      j:=j-1;
    END;
    Элемент[0]:=c;
    IF i< ДлинаПерестановки-1 THEN
      Печать;
    END;
    IF ДлинаПерестановки>0 THEN
```

```
Перестановка(ДлинаПерестановки-1);
END;
i:=i+1;
END;
END Перестановка;
BEGIN
  In.Open;
  ДлинаПерестановки:=-1;
  WHILE In.Done DO
    ДлинаПерестановки:=ДлинаПерестановки+1;
    In.Int(Элемент[ДлинаПерестановки]);
  END;
  Печать; (*Распечатка начальной перестановки*)
  Перестановка(ДлинаПерестановки);
END ПостроениеПерестановок;
END Модуль.
```

Нерекурсивное решение. Попробуем нерекурсивное решение получить, смоделировав рекурсивный процесс. Рекурсивная процедура **Перестановка()** выполняет сдвиги подмассива своей длины и для каждого сдвига активирует новую копию себя для подмассива меньшей длины. Рекурсивный механизм берет на себя важную задачу определения длины подмассива, для которого в данный момент необходимо выполнить сдвиг. Вот эту задачу прежде всего нам и предстоит реализовать.

Как определить длину подмассива для очередного сдвига? Будем отталкиваться от очевидного факта: подмассив большей длины сдвигается только после того, как выполнены все возможные сдвиги подмассива меньшей длины. Например, подмассив длины 4 можно обрабатывать только после того, как выполнено 3 операции сдвига подмассива длины 3. Отсюда следует, что поиск кандидата на циклический сдвиг необходимо вести от минимального подмассива (длина 2). Кандидат на сдвиг – это минимальный подмассив, для которого количество сдвигов меньше его собственной длины.

Таким образом, прояснена еще одна важная вещь. Для успешного поиска кандидата на сдвиг необходимо хранить информацию о количестве сдвигов для каждого подмассива. В рекурсивном варианте такая информация хранится в стеке, здесь же придется объявить специальный массив счетчиков сдвигов.

Главный цикл, описывающий процесс, завершается тогда, когда выполнены все сдвиги для полного массива. Поэтому главный цикл запишется так:

```
WHILE Счетчик[N]<N DO
  (*Вся обработка перестановок*)
END;
```

Массив счетчиков устроен следующим образом: индекс счетчика – это длина подмассива, значение счетчика – это количество выполненных над подмассивом

сдвигов. Нулевой и первый элементы массива счетчиков выпадают из рассмотрения, это можно рассматривать как бессмысленную потерю памяти, но такая потеря несколько упрощает нумерацию счетчиков, что довольно-таки полезно.

Задание для самостоятельной работы

После того как завершите разбор нашей реализации, модифицируйте её таким образом, чтобы в массиве счетчиков не было лишних элементов.

Продолжим наш анализ. Начнем поиск кандидата на сдвиг. Для чего найдем счетчик, чье значение меньше его номера, поиск начнем со второго элемента:

```
WHILE Счетчик[N]<N DO
  i:=2;
  WHILE Счетчик[i]=i DO i:=i+1;END;
END;
```

Счетчик найден, следовательно, обнаружен и подмассив. Теперь можно выполнить сдвиг и, соответственно, увеличить значение счетчика:

```
WHILE Счетчик[N]<N DO
  i:=2;
  WHILE Счетчик[i]=i DO i:=i+1;END;
  Счетчик[i]:= Счетчик[i]+1;
  d:= Элемент[1];
  FOR k:=1 TO i-1 DO Элемент[k]:= Элемент[k+1]; END;
  Элемент[i]:=d;
END;
```

Последний момент – печать. Каждый сдвиг дает перестановку. Как мы уже знаем, перестановки могут повторяться. Повторяющиеся перестановки – это первая и последняя. Последняя перестановка – это перестановка, полученная сдвигом, чей номер равен длине подмассива и, соответственно, равен номеру своего счетчика. Следовательно, печатаем только ту перестановку, для которой значение счетчика меньше его номера.

Из факта печати перестановки следует возможность сдвигов подмассивов меньшей длины, но их счетчики уже переполнены. Поэтому в блоке печати перестановки необходимо обнулить все младшие счетчики, чтобы опять сделать возможным младшие сдвиги.

```
WHILE Счетчик[N]<N DO
  i:=2;
  WHILE Счетчик[i]=i DO i:=i+1;END;
  Счетчик[i]:= Счетчик[i]+1;
  d:= Элемент[1];
  FOR k:=1 TO i-1 DO Элемент[k]:= Элемент[k+1]; END;
  Элемент[i]:=d;
  IF Счетчик[i]<i THEN
    Печать;
```

```

    FOR k:=2 TO i-1 DO Счетчик[k]:=0; END;
  END;
END;

```

Ниже, как обычно, – полное решение.

Листинг 7.2

```

MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE Перестановка*;
VAR
  Элемент,Счетчик:ARRAY 10 OF INTEGER;
  d,N,i,k:INTEGER;
PROCEDURE Печать;
VAR
  k:INTEGER;
BEGIN
  FOR k:=1 TO N DO StdLog.Int(Элемент[k]);END;
  StdLog.Ln;
END Печать;
BEGIN
  In.Open;
  In.Int(N);
  FOR i:=1 TO N DO
    In.Int(Элемент[i]); Счетчик[i]:=0;
  END;
  Печать; (*Исходное состояние массива – это первая перестановка*)
  WHILE Счетчик[N]<N DO
    i:=2;
    WHILE Счетчик[i]=i DO i:=i+1;END;
    Счетчик[i]:= Счетчик[i]+1;
    d:= Элемент[1];
    FOR k:=1 TO i-1 DO Элемент[k]:= Элемент[k+1]; END;
    Элемент[i]:=d;
    IF Счетчик[i]<i THEN
      Печать;
      FOR k:=2 TO i-1 DO Счетчик[k]:=0; END;
    END;
  END;
END Перестановка;
END Модуль.

```

Задание для самостоятельной работы

Подсчитайте, сколько «лишних» перестановок будет порождено при обработке множества, состоящего из N элементов.

Рассмотрим еще одну идею получения перестановок без программной реализации. Перестановки пусть содержатся в массиве **Элемент**[], и очередную перестановку будем получать из предыдущей. Определим для перестановок понятие лексикографического порядка. Сравним две перестановки A и B . Пусть при проходе перестановки от первого элемента некоторое количество (возможно, нулевое) элементов перестановки A совпадает с элементами перестановки B . И пусть в i -й позиции элементы не совпадают. Тогда если $A_i > B_i$, то $A > B$ иначе $B > A$. Сравнение на больше/меньше для чисел определяется естественным образом. Для литер отношение больше/меньше можно определить по позиции литеры в алфавите. Например, большей литерой можно считать литеру с меньшим порядковым номером в алфавите.

Алгоритм в процессе своей работы, начиная с лексикографически наименьшей перестановки, строит большие до тех пор, пока не получит максимальную. Например, (1, 2, 3) – наименьшая, а (3, 2, 1) – наибольшая перестановка. Лексикографическое увеличение перестановки должно быть минимальным. Что это означает? Если перестановка B получена из перестановки A , то не должно быть такой перестановки C , что $A < C < B$.

Как это можно обеспечить?

Или иначе, в каком случае i -й член перестановки можно увеличить, не меняя предыдущих? Ответ: только в том случае, если он меньше какого-либо из следующих членов (членов с номерами больше i). Необходимо наибольшее i , при котором **Элемент**[i] < **Элемент**[$i+1$] > ... > **Элемент**[N]. После этого **Элемент**[i] увеличивается на наименьшую величину, для чего среди **Элемент**[$i+1$], ..., **Элемент**[n] ищется наименьший элемент-число, больший **Элемент**[i]. Поменяв **Элемент**[i] с найденным, сортируем элементы правее **Элемент**[i] в порядке возрастания.

Задание для самостоятельной работы

Напишите программу по описанному выше методу.

Построение сочетаний без повторений на множестве элементов

Вторая базовая задача комбинаторного программирования – это построение сочетаний без повторений. Для её решения также приведем два различных алгоритма и оба доведем до завершённой программы. Первый вариант заключается в сведении проблемы получения сочетаний к построению двоичных чисел.

Пронумеруем множество (будем далее называть его базовым), из элементов которого должны быть составлены сочетания. Нумерация выполняется естественным образом, если множество представлено массивом. Составим еще один массив (назовем его двоичным числом) целых чисел, значениями которого могут быть только 0 и 1.

Правило построения сочетания таково: если k -ая цифра двоичного числа равна единице, то k -ый элемент базового множества входит в сочетание, иначе – нет.

Таким образом, задача получения всех сочетаний сводится к задаче получения

всех k -разрядных двоичных чисел. Получить же все двоичные числа можно так:

- в начале процесса проведем инициализацию двоичного числа нулями;
- очередное двоичное число получается прибавлением к предыдущему числу единицы по правилам двоичной арифметики.

Реализацию начнем с уточнения условия завершения процесса. Ясно, что последнее сочетание – это сочетание, соответствующее двоичному числу без нулей. Предположим, что каким-то, пока неясным образом в ходе процесса мы сможем установить факт отсутствия нулей. Это можно сделать многими способами, например просто просмотреть массив двоичного числа, но пока оставим этот вопрос. Главный цикл построим так:

```
Флаг:=TRUE;  
WHILE Флаг DO  
  (*Главный процесс*)  
  (*Построение двоичного числа*)  
  (*Распечатка очередной выборки*)  
END;
```

Ключевой пункт здесь – прибавление к двоичному числу единицы по правилам двоичной арифметики. Для прибавления единицы достаточно найти первый (от начала числа) ноль, заменить его на единицу и все младшие единицы обнулить.

```
i:=0;  
WHILE (i<n) & (Число[i]=1) DO i:=i+1;END;
```

Этот цикл обнаружит первый значащий ноль. Но может и не обнаружить. В случае удачи необходимо прибавить единицу и распечатать очередную выборку. Случай неудачи как раз и означает достижение максимально возможного числа:

```
Флаг:=TRUE;  
WHILE Флаг DO  
  i:=0;  
  WHILE (i<n) & (Число[i]=1) DO i:=i+1;END;  
  IF i<n THEN  
    (*Операции*)  
  ELSE  
    Флаг:=FALSE;  
  END;  
END;
```

Фрагмент из листинга показывает решение проблемы с завершением главного цикла. В случае удачи выполняется следующий фрагмент:

```
Число[i]:=1; (*Найденный ноль заменяется на единицу*)  
(*Обнуляются младшие разряды*)  
FOR k:=0 TO i-1 DO Число[k]:=0;END;  
(*Печать очередной выборки*)  
FOR k:=0 TO n-1 DO
```

```
IF Число[k]=1 THEN StdLog.Int(Элемент[k]);END;
END;
```

И наконец, полная реализация:

Листинг 7.3

```
MODULE Модуль;
  IMPORT In, StdLog;
  PROCEDURE СочетанияБезПовторений*;
    VAR
      Элемент, Число:ARRAY 20 OF INTEGER;
      i,n,k:INTEGER;
      Флаг:BOOLEAN;
    BEGIN
      In.Open;
      In.Int(n);
      FOR i:=0 TO n-1 DO
        In.Int(Элемент[i]);
        Число[i]:=0;
      END;
      Флаг:=TRUE;
      WHILE Флаг DO
        i:=0;
        WHILE (i<n) & (Число[i]=1) DO i:=i+1;END;
        IF i<n THEN
          Число[i]:=1;
          FOR k:=0 TO i-1 DO Число[k]:=0;END;
          FOR k:=0 TO n-1 DO
            IF Число[k]=1 THEN StdLog.Int(Элемент[k]);END;
          END;
          StdLog.Ln;
        ELSE
          Флаг:=FALSE;
        END;
      END;
    END СочетанияБезПовторений;
  END Модуль.
```

Второй вариант решения строится на несколько иной идее. Заметим, что каждый элемент базового множества может либо входить в сочетание, либо не входить. Если для сочетания выделить отдельный массив с длиной, равной длине массива базового множества, то для каждого элемента массива сочетания возможны два состояния: либо этот элемент пуст, либо он содержит соответствующий элемент базового множества.

Для первого элемента сочетания возможны два состояния (содержит, не содер-

жит), для каждого состояния первого элемента возможны два состояния второго элемента и т. д. Введем понятие последовательности сочетания. Последовательностью сочетания длины k будем называть часть массива длины k , начиная с первого элемента. Множество последовательностей длины k возможно определить через множество последовательностей длины $k-1$ следующим образом: каждая последовательность длины k получается из некоторой последовательности $k-1$ добавлением в позицию k либо пробела, либо k -го элемента базового множества.

Тогда множество сочетаний – это множество последовательностей длины N . Полученное определение имеет рекуррентный вид, из чего следует возможность получения рекурсивного решения. Строится рекурсивное решение на следующих очевидных соображениях:

- цель искомой процедуры – за N вызовов построить очередное сочетание, следовательно, на N -ом вызове можно завершить построение очередного сочетания и выполнить распечатку;
- каждый вызов удлиняет уже построенную последовательность на один элемент;
- очередной элемент последовательности можно определить двумя способами (пустой или элемент базового множества), следовательно, каждый вызов процедуры содержит еще два вызова.

Начинается рекурсивный процесс с первого элемента. Он может быть включен в сочетание, а может быть и нет. Поэтому в главной процедуре две активации:

Сочетания(0,1);

Сочетания(1,1);

Первая активация включает в сочетание первый элемент, вторая – нет. Второй параметр указывает на длину сочетания. При обработке первого элемента построенная длина сочетания равна 1. Этот параметр нужен для определения, построено сочетание или нет. Каждая активация процедуры выполняет операцию достройки сочетания на один элемент. И это первое, что делает процедура в процессе своей работы.

IF t=0 THEN

ЭлементСочетания[k]:=0;

ELSE

ЭлементСочетания[k]:=ЭлементМножества[k];

END;

Если параметр t нулевой, то элемент в сочетание не входит, иначе элемент в сочетании участвует. После достройки сочетания определяем, построено оно полностью или нет. Если нет, то отправляем на обработку следующий элемент, который также может быть включен или нет.

IF k<n THEN

Сочетание(0,k+1);

Печать;

```

Сочетание(1,k+1);
Печать;
END;

```

Сворачивание цепочки активаций означает завершение достройки сочетания, следовательно, после каждого рекурсивного вызова необходимо попытаться распечатать полученное сочетание.

Задание для самостоятельной работы

Выше было сказано о процедуре печати «попытаться». И процедура **print** действительно именно пытается печатать, о чем говорит оператор:

```

IF k+1=n THEN
  (*Цикл печати*)
END;

```

Если убрать это условие, то программа некоторые сочетания будет выдавать дважды. Проанализируйте тестовые примеры и ответьте на вопрос: откуда возникает необходимость в этом дополнительном условии и можно ли от него избавиться?

Ниже приведено полное решение задачи:

Листинг 7.4

```

MODULE Модуль;
  IMPORT In, StdLog;
  PROCEDURE СочетанияБезПовторений*;
    VAR
      ЭлементСочетания,ЭлементМножества:ARRAY 20 OF INTEGER;
      i,n:INTEGER;
    PROCEDURE Сочетания(t,k:INTEGER);
    PROCEDURE Печать;
      VAR
        i:INTEGER;
    BEGIN
      IF k+1=n THEN
        FOR i:=1 TO n DO
          IF ЭлементСочетания[i]#0 THEN
            StdLog.Int(ЭлементСочетания[i]);
          END;
        END;
        StdLog.Ln;
      END;
    END Печать;
    BEGIN
      IF t=0 THEN
        ЭлементСочетания[k]:=0;
      ELSE

```

```

ЭлементСочетания[k]:=ЭлементМножества[k];
END;
IF k<n THEN
  Сочетания(0,k+1);
  Печать;
  Сочетания(1,k+1);
  Печать;
END;
END Сочетания;
BEGIN
  In.Open;
  In.Int(n);
  FOR i:=1 TO n DO
    In.Int(ЭлементМножества[i]);
  END;
  Сочетания(0,1);
  Сочетания(1,1);
END СочетанияБезПовторений;
END Модуль.

```

Задания для самостоятельной работы

- Второй алгоритм (листинг 7.4) индексирует массивы с единицы. Внесите исправления в программу, так чтобы индексация выполнялась с нуля.
- Выборка, даже короткая, сопоставляется с максимально длинной последовательностью длины N . Это означает, что часть последовательности (хвост), состоящая из одних пробелов, не несет в себе информации о сочетании, но тем не менее этот хвост будет добросовестно построен. Это означает существенную потерю эффективности. Попробуйте скорректировать алгоритм.
- Второй вариант решения построен рекурсивно. Напишите программу, реализующую ту же идею, но без рекурсии.

Сочетания с повторениями

Немного ограничим задачу. Сочетания без повторений были получены все. Сочетания с повторениями будем строить только определенной длины. И ограничимся только одним решением – рекурсивным. Напомним, что сочетания с повторениями отличаются от сочетаний без повторений тем, что элемент, уже принявший участие в построении сочетания, не исключается и может быть представленным в сочетании сколько угодно раз. Это, кстати, означает, что сочетания с повторениями могут иметь любую длину, в отличие от сочетаний без повторений, длина которых ограничена размером исходного множества.

Для построения рекурсивного решения необходимо рекуррентное определение сочетания. Пусть сочетание длины N уже построено. И пусть исходное множество содержит M элементов. Тогда сочетание длины N определяет M сочетаний длины

$N+1$, каждое из которых может быть получено добавлением к сочетанию длины N еще одного символа из исходного множества. Аргументом такой рекурсивной процедуры может быть позиция сочетания, в которую добавляется очередной элемент. Вызов процедуры можно выполнить в цикле, который M раз запускает процедуру для нулевой позиции:

```
FOR k:=0 TO ДлинаМножества -1 DO
  РекурсивнаяПроцедура(k,0);
END;
```

В теле процедуры выполним операцию добавления очередного элемента и определим условие завершения активаций:

```
ЭлементСочетания[НомерАктивации] := ЭлементМножества[k];
IF НомерАктивации = ДлинаСочетания THEN
  ELSE
  END;
```

Если номер активации равен длине сочетания, то, очевидно, сочетание заданной длины построено, так как каждая активация удлиняет сочетание на единицу. Поэтому в случае истинности условия необходимо выполнить распечатку очередного сочетания:

```
FOR j:=0 TO ДлинаСочетания DO
  StdLog.Int(ЭлементСочетания[j]);
END;
StdLog.Ln;
```

Если условие неверно, то, очевидно, длина сочетания меньше требуемой и необходимо продолжить построение, организовав цикл активаций:

```
FOR j:=0 TO ДлинаМножества -1 DO
  РекурсивнаяПроцедура(j,НомерАктивации+1);
END;
```

Собственно, и все. Ниже – листинг полного решения.

Листинг 7.5

```
MODULE Модуль;
  IMPORT StdLog, In;
  PROCEDURE СочетанияС_Повторениями*;
  VAR
    ЭлементМножества, ЭлементСочетания: ARRAY 10 OF INTEGER;
    k, ДлинаМножества, ДлинаСочетания: INTEGER;
  PROCEDURE РекурсивнаяПроцедура(k, НомерАктивации: INTEGER);
  VAR
    j: INTEGER;
  BEGIN
    ЭлементСочетания[НомерАктивации] := ЭлементМножества[k];
```

```

IF НомерАктивации= ДлинаСочетания THEN
  FOR j:=0 TO ДлинаСочетания DO
    StdLog.Int(ЭлементСочетания[j]);
  END;
  StdLog.Ln;
ELSE
  FOR j:=0 TO ДлинаМножества -1 DO
    РекурсивнаяПроцедура(j,НомерАктивации+1);
  END;
END;
END РекурсивнаяПроцедура;
BEGIN
  In.Open;
  In.Int(ДлинаСочетания);
  In.Int(ДлинаМножества);
  FOR k:=0 TO ДлинаМножества -1 DO In.Int(ЭлементМножества[k]); END;
  FOR k:=0 TO ДлинаМножества -1 DO
    РекурсивнаяПроцедура(k,0);
  END;
END СочетанияС_Повторениями;
END Модуль.

```

Задание для самостоятельной работы

Напишите программу, получающую сочетания с повторениями всех длин от 1 до L , где L – вводимое число.

Задача получения размещений

Писать реализацию размещений мы не будем. Размещения от сочетаний отличаются только тем, что для них существенно важен порядок. Это означает, что, имея решения для перестановок и сочетаний, вполне можно получить и реализацию размещений. Действительно, получив некое сочетание и построив из него все возможные перестановки, мы получим размещения. Следовательно, задача построения размещений сводится к задаче построения сочетаний, которые как аргумент передаются процедуре построения перестановок. Если есть желание, можете написать такую программу самостоятельно.

В заключение. Конечно же, приведенные задачи не исчерпывают всего многообразия комбинаторных проблем. Конечно же, это только базовые задачи, в реальной задаче необходимость построения $n!$ -элементной или перестановок еще надо увидеть, и алгоритм построения базовой комбинаторной структуры надо еще уметь интегрировать с реальной комбинаторной проблемой. А главный и, к сожалению, зачастую не решаемый вопрос – это космическая скорость роста количества комбинаций, поэтому если вы уверенно определили задачу как комбинаторную, то вам надо быть готовым к необходимости поиска эвристического решения.

Динамические структуры данных

Понятие о динамической величине	225
Линейный связный список	226
Использование рекурсивных определений для создания деревьев данных	233

Понятие о динамической величине

Любая программа состоит из операторов и данных. Операторы нужны для записи операций, выполняемых над числами, литерами, строками и т. д. Отделить операторы от данных нельзя, одно без другого не существует. Однако во всех главах этой книги данные и действия над ними неравноправны. Рассматривая тот или иной алгоритм, мы всегда разбираем необходимые операции, структуры же данных являются чем-то вторичным, не представляющим специальной проблемы. Это, конечно, не всегда так. Серьезное отступление от такого представления видно в главе, посвященной алгоритмам на графах. Представление графа матрицей смежности самым существенным образом влияет на разработку алгоритмов, можно даже с уверенностью сказать, что другое представление изменило бы если не алгоритмы, то их реализацию до неузнаваемости.

Этот пример говорит о том, что возможны ситуации, в которых структуры данных могут стать если не ключевой проблемой, то во всяком случае встать вровень с чисто алгоритмическими построениями. В сущности, в этом вопросе все зависит от сложности построений. Если, например, мы исследуем числовые алгоритмы, в которых нет даже массивов, но сами алгоритмы не тривиальны, то понятно, что структуры данных будут играть второстепенную или третьестепенную роль. В алгоритмах на графах используется двумерный массив – матрица смежности, достаточно нетривиальный объект, поэтому там структуры данных нуждаются в специально на них затраченной умственной энергии. Сейчас же мы обсудим структуры, сложность которых такова, что даже для их построения необходимы алгоритмические усилия.

В заголовке главы сказано, о чем пойдет речь, – о динамических данных, но термин «динамические» включает в себя достаточно разные вещи, поэтому придется потратить еще немного вашего внимания для уточнения цели обсуждения.

Статические и динамические величины. Чтобы хорошо вникнуть в понятие динамической величины, поговорим немного о статических переменных, опираясь на различие – хороший прием для понимания. Статические данные – это такие величины, память под которые выделяется компилятором, естественно, на этапе компиляции. Выделяемая память жестко фиксирована по объему. Эти величины нельзя удалить, и если программа в них перестала нуждаться, то это ничего не меняет, они продолжают занимать память. Полезность статических величин огромна. Если они создаются компилятором, то на него можно возложить ответственность за корректность выделения памяти. Это, кстати, очень важный момент. Распределение памяти – достаточно не простая процедура. Кроме того, для статических переменных четко определяется набор операций, который можно над ними выполнять. Оба этих момента позволяют обнаруживать большое количество ошибок на этапе компиляции, что чрезвычайно экономит время программиста, и не только начинающего.

Можно встретиться с мнением, что жесткий контроль за процессом создания величин ограничивает программиста в возможностях. Но это очень тонкий вопрос. Может быть и так, но всегда можно усомниться в необходимости тех или иных

хитроумных возможностей. Хорошее решение – это все же решение простое, но главное – то, что хитроумные возможности создают возможности и для хитроумных ошибок, а мнение, что опытный программист не ошибается, глубоко не верное. Кроме того, существует достаточно много ситуаций, в которых программистская ошибка может стоить достаточно дорого.

Таким образом, принципы статического определения создают хорошую защиту от ошибок, но необходимость точного определения объема требуемой памяти – все же достаточно сильное ограничение. Нетрудно придумать задачу, в которой будет обрабатываться некий массив переменной длины. Например, результаты измерений некоторой величины в течение заданного отрезка времени. Если на заданном отрезке измерения проводятся каждую секунду, то это один объем данных, а если каждую сотую секунду, то это объем, в 100 раз больший, но статическое определение требует на все случаи жизни определить массив одной заданной длины, следовательно, придется тратить памяти столько, сколько её не всегда понадобится. И это еще половина беды. Длина структуры данных может изменяться в процессе работы. Пусть, например, перед программой поставлена задача обслуживания очереди (не важно, чего). Очередь имеет обыкновение изменять свою длину как в сторону увеличения, так и в сторону уменьшения, но массив имеет фиксированную длину!

Поэтому статическое определение данных – действительно сильное ограничение, и поэтому любой язык программирования предоставляет программисту возможность объявлять структуры данных, созданием и уничтожением которых можно управлять по ходу выполнения программы. Такие величины и называются динамическими. Здесь программист получает большую свободу. В каждый отдельный момент можно выделить столько памяти, сколько нужно, и вернуть её обратно, когда необходимость отпадет.

Примеры динамической структуры:

A: POINTER TO ARRAY 100 OF INTEGER; – указатель на массив фиксированной длины.

A: POINTER TO ARRAY OF INTEGER; – указатель на открытый массив. Его длина определяется в момент создания.

В момент такого определения выделяется память не под массив, а под указатель на массив, память под собственно массив будет выделена в процессе выполнения программы. Мы немного вспомнили, что такое динамические величины, но далее объектом исследования будут не динамические величины вообще, а так называемые рекурсивно определяемые.

Линейный связный список

Что такое рекурсивно определяемая величина? Рассмотрим следующую запись:

A: ARRAY OF INTEGER;

Здесь термин A определяется тремя терминами: ARRAY, OF, INTEGER. Термина A среди них нет. Теперь следующее определение:

A: ARRAY OF A;

Это определение уже рекурсивно, здесь **A** определяется через **A**. Это рекурсия, но рекурсия несодержательная. Здесь допущена грубая ошибка, идентификатор **A** используется в двух смыслах: слева от двоеточия – это идентификатор переменной, а справа – это идентификатор типа. Два совершенно различных смысла. Это во-первых. Во-вторых, выходит так, что переменная **A** вполне определена через тип **A**, но что такое тип **A**, осталось неизвестным. Рассмотрим другое определение:

```
ТипУказатель = POINTER TO ТипЗапись;
ТипЗапись = RECORD
    Число:INTEGER;
    Следующий: ТипУказатель;
END;
```

Здесь тип **ТипУказатель** определяется через тип **ТипЗапись**. А тип **ТипЗапись**, в свою очередь, содержит в определении тип **ТипУказатель**. Тип содержит в своем определении тип, это нормальная ситуация, это правилами языка вполне допустимо. Более того, это содержательное определение, так как тип **ТипЗапись** не сводится к чистому типу **ТипУказатель**. Он есть запись, содержащая данные, отличные от **ТипУказатель**. Данное определение можно проиллюстрировать следующим рисунком:



Рис. 8.1

Каждый прямоугольник изображенный на рис. 8.1, – это запись типа **ТипЗапись**. Она содержит значение компонента **Число** (**a**), и её компонент **Следующий** (**next**) указывает на следующую запись. Каждая запись знает о своем следующем соседе, и от записи до записи можно пройти, пользуясь указателем **Следующий**. Путешествие по списку записей выполняется присвоением адреса, хранящегося в **Следующий**, некоторой переменной – указателю, которая для совместимости обязана быть типом **ТипУказатель**. Такая структура данных называется связным списком. Ниже – пример, создающий связный список и выполняющий его проход.

Листинг 8.1

```
MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE ПростойСвязныйСписок*;
TYPE
    ТипУказатель=POINTER TO ТипЗапись;
    ТипЗапись=RECORD
        Число:INTEGER;
        Следующий: ТипУказатель;
    END;
VAR
```

```

Указатель,Запасной: ТипУказатель;
k:INTEGER;
BEGIN
  NEW(Указатель); Запасной:=Указатель;
  Указатель^.Число:=1;
  k:=2;
  WHILE k<=10 DO
    NEW(Указатель^.Следующий);
    Указатель:= Указатель^.Следующий;
    Указатель^.Число:=k;
    k:=k+1;
  END;
  k:=1;
  Указатель:=Запасной;
  WHILE k<=10 DO
    StdLog.Int(Указатель^.Число);
    Указатель:= Указатель^.Следующий;
    k:=k+1;
  END;
END ПростойСвязныйСписок;
END Модуль.

```

Разберем существенные моменты. Первый цикл **WHILE** выполняет работу по созданию связного списка. До начала его работы создается первая запись оператором **NEW(Указатель)**, и полученный адрес запоминается в дополнительном указателе (зачем чуть позже). Затем в цикле на каждом шаге:

1. Создается новая запись **NEW(Указатель^.Следующий)**.
2. Выполняется переход на новую запись **Указатель:=Указатель^.Следующий**.
3. Компоненте **Число** присваивается некоторое значение **Указатель^.Число:=k**.

После создания связного списка выполняется повторный его проход. И здесь возникает проблема. Обращение к элементу связного списка в программе возможно только двумя способами: либо через указатель **Следующий** предыдущей записи, либо через указатель **Указатель**. Но на первом шаге цикла **Указатель** уходит на вторую запись, забывая местонахождение первой, и если учесть, что нет записи, содержащей информацию об адресе первой, то можно понять, что первая запись окажется потерянной. В памяти она остается, но получить доступ к ней уже невозможно. Такая ситуация называется утечкой памяти. Используемый нами Компонентный Паскаль решает эту проблему следующим образом: если до структуры добраться уже все равно невозможно, то разумно структуру уничтожить, а кусок памяти вернуть в свободную область. Эту работу автоматически выполняет сборщик мусора. Таким образом, гарантируется, что в памяти компьютера всегда будут храниться только реально используемые данные. Но это приведет к дальнейшей потере элементов списка. А именно на втором шаге **Указатель** уйдет на третью

запись и забудет о второй. Первая запись будет уничтожена сборщиком мусора, следовательно, вторая запись окажется в положении ранее уничтоженной первой и тоже попадет в поле зрения сборщика мусора и т. д.

В результате, когда процедура создания связанного списка дойдет до конца, в памяти компьютера будет сохранена только одна последняя запись (скорее всего, это произойдет не так быстро, но произойдет). Поэтому перед началом работы первого цикла **WHILE** адрес первой записи сохраняется в дополнительном указателе. Это позволяет избежать санкций со стороны сборщика мусора и дает возможность повторно пройти список, выполнив присвоение **Указатель:=Запасной**. Так сказать, вспомнить, где находится первая запись. Далее начнем усложнять ситуацию, но сначала – еще раз о сути рекурсивного определения.

- Рекурсивное определение – это всегда определение типа. Его рекурсивная природа обеспечивается указательным типом.
- Рекурсивная структура – это запись, содержащая сколь угодно сложные и объемные структуры данных + указатель на структуру такого же типа.
- Применив к указателю, входящему в запись, процедуру **NEW**, получаем еще одну запись такого типа, связанную с данной через указатель.
- Немаловажно то, что указатель на запись может быть не один.

Важное примечание. Для любого языка существует правило: любая структура должна быть определена до момента своего первого использования. В нашем определении это правило нарушается. Тип **ТипУказатель** определяется через тип **ТипЗапись**, описанный ниже. Если определения поменять местами:

```
ТипЗапись = RECORD
    Число:INTEGER;
    Следующий:ТипУказатель;
END;
ТипУказатель = POINTER TO ТипЗапись;
```

то это ничего не изменит, теперь тип **ТипЗапись** использует идентификатор **ТипУказатель**, определяемый ниже. Для Компонентного Паскаля это не представляет проблемы, так как КП считает определения известными друг другу, если они находятся на одном уровне, например если таковой тип описан в блоке описаний модуля. В некоторых языках для такой структуры определяется допустимый порядок.

Зачем рекурсивные структуры нужны?

Массив, определенный так:

```
A: POINTER TO ARRAY 100 OF INTEGER
```

является динамическим. Это означает, что его можно создать и от его услуг можно отказаться в процессе исполнения программы. Но его размер фиксирован на этапе компиляции. Массив, объявленный так:

A: POINTER TO ARRAY OF INTEGER

в некотором смысле более динамический. Его размер определяется в ходе выполнения программы, но будучи определенным, он уже не может измениться. Можно, конечно, придумать операции, изменяющие объем памяти, выделенной под динамические структуры, прецеденты таких решений есть, но для этого необходимо сначала отказаться от строгой типизации, что означает очень высокую цену решения. И только рекурсивно определенные структуры полностью свободны от ограничений в управлении со стороны программиста. В объявлении величины рекурсивного типа нет никаких указаний на количество записей, которые можно создать. Даже для линейного списка создание записи дает возможность выделения памяти еще под одну запись. Появление новой записи дает возможность для создания новой и т. д. То есть размер линейного связного списка – величина переменная и зависит только от объема памяти, выделяемого под задачу. В этом первый смысл связного списка.

Второй смысл заключается в возможности создания структур данных почти произвольной сложности. Говорить о структурах почти произвольной сложности мы сможем после определения такой структуры как дерево. Но нам еще необходимо потренироваться в выполнении операций над линейным списком. Выделим следующие операции:

- добавление записи к хвосту списка;
- добавление записи к голове списка;
- удаление группы записей;
- склеивание двух списков;
- вставка списка в список.

Проблема добавления записи к хвосту списка уже решена в *задаче о проходе списка*. Заметим только, что для добавления записи необходимо указатель довести до последней записи. Остальные проблемы представляют интерес.

Задача 2. Добавление записи к голове списка

На этой операции хорошо видно преимущество связного списка перед массивом. Для того чтобы добавить элемент в начало массива, необходимо выполнить смещение всех элементов массива, поэтому чем массив длиннее, тем больше указанная операция будет занимать времени. Линейный связный список отличается от массива тем, что его элементы, будучи упорядоченными в списке, не обязаны быть упорядочены в адресном пространстве. Поэтому для добавления элемента к голове списка элемент достаточно создать и его соответствующей компоненте сообщить адрес бывшего первого элемента. Эту задачу решим частично, запишем только фрагмент программы, добавляющий одну запись. Пусть указатель **Начало** указывает на начало связного списка. Тогда искомый фрагмент можно записать так:

```
NEW(Указатель);  
Указатель.Следующий:=Начало;  
Начало:=Указатель;
```

Задание для самостоятельной работы

Напишите программу, создающую линейный связный список из некоторого количества записей, после чего добавьте некоторое количество записей в голову этого списка.

Задача 3. Удаление группы записей

Вспомним, что фактически удалением динамических переменных занимается сборщик мусора. Сигналом для удаления переменной является её фактическая ненужность. Ненужность величины означает, что нет указателя, хранящего её адрес. Пусть наш список называется *A*. И пусть перед нами стоит задача уничтожить записи с номера *N1* по номер *N2* включительно. Это означает, что необходимо разорвать связь между записью *N1* и записью с номером *N1+1* и установить связь между записями с номерами *N1* и *N2+1* так, как это показано на рис. 8.2.

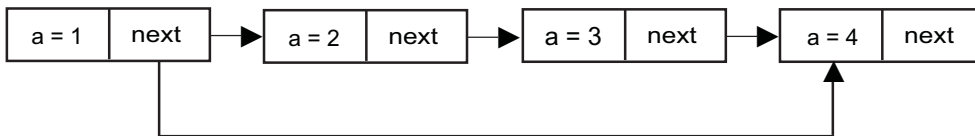


Рис. 8.2

Листинг 8.2

Указатель:=Запасной; (*переход на начало списка*)

(*переход на запись с номером *N1**)

FOR k:=1 TO *N1*-1 DO

Указатель:=Указатель.Следующий;

END;

Вспомогательный:=Указатель;

(*Переход на запись с номером *N2+1* *)

FOR k:=1 TO *N2*-*N1*+1 DO

Указатель:=Указатель.Следующий;

END;

Вспомогательный.Следующий:=Указатель;

Задача 4. Склеивание двух списков

Имеется в виду следующее: есть два списка. Необходимо начало одного из них подклеить к концу другого. Операция – бессмысленная в терминологии массивов. Возможность такой операции еще раз подчеркивает мощь и гибкость конструкции списка в сравнении с массивом. Действительно, для операции подклейки достаточно пройти на последнюю запись списка *A* и записать в неё адрес первой записи списка *B*. Собственно, и все.

Задача 5. Вставка списка в список

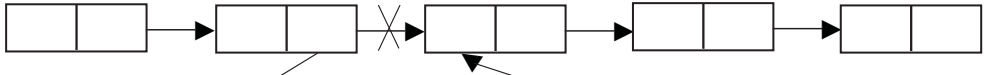
Решение. Выполним следующие действия:

1. Пройдем по списку *B* до позиции *L* (начало вставки), запоминая при этом адрес предшествующей записи.
2. Предшествующую запись свяжем с началом списка *B*.

3. Конец списка *B* свяжем с продолжением списка *A* (запись *L*).

Сказанное проиллюстрировано на следующем рис. 8.3.

Список А



Список В



Рис. 8.3

Листинг 8.3

```

PROCEDURE ВставкаСписка*;
TYPE
  ТипУказатель=POINTER TO ТипЗапись;
  ТипЗапись=RECORD
    Число:INTEGER;
    Следующий: ТипУказатель;
  END;
VAR
  Указатель1,Указатель2:ТипУказатель;
  Запасной1,Запасной2,Вспомогательный:ТипУказатель;
  k,L:INTEGER;
BEGIN
  In.Open;
  In.Int(L);
  NEW(Указатель1); Запасной1:=Указатель1;
  NEW(Указатель2); Запасной2:= Указатель2;
  Указатель1.Число:=1;
  Указатель2.Число:=1;
  FOR k:=2 TO 10 DO
    NEW(Указатель1.Следующий);
    NEW(Указатель2.Следующий);
    Указатель1:= Указатель1.Следующий;
    Указатель2:= Указатель2.Следующий;
    Указатель1.Число:=k;
    Указатель2.Число:=k*k;
  END;
  Указатель1:=Запасной1;
  FOR k:=1 TO L-1 DO
    Вспомогательный:=Указатель1;

```



```

Указатель1:=Указатель1.Следующий;
END;
Вспомогательный. Следующий:= Запасной2;
Указатель2.Следующий:=Указатель1;
Указатель1:=Запасной1;
FOR k:=1 TO 20 DO
  StdLog.Int(Указатель1.Число);
  Указатель1:=Указатель1.Следующий;
END;
END ВставкаСписка;

```

Использование рекурсивных определений для создания деревьев данных

Если запись содержит более одного указателя, то последовательность записей начинает ветвиться (рис. 8.4):

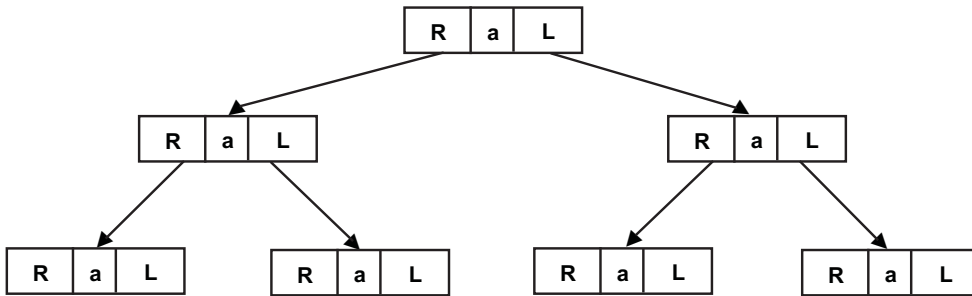


Рис. 8.4

Это пример простейшего дерева – двоичного. Ниже – листинг программы, создающей дерево и выполняющей его последующий проход.

Листинг 8.4

```

MODULE Модуль;
IMPORT In, StdLog;
TYPE
ТипУказатель=POINTER TO ТипЗапись;
ТипЗапись=RECORD
  Число:INTEGER;
  Левый,Правый: ТипУказатель;
END;
PROCEDURE ПостроениеДерева(Дерево:ТипУказатель;n:INTEGER);
BEGIN
  Дерево.Число:=n;
  IF n<4 THEN
    NEW(Дерево.Левый);

```

```

ПостроениеДерева(Дерево.Левый,n+1);
NEW(Дерево.Правый);
ПостроениеДерева(Дерево.Правый,n+1);
END;
END ПостроениеДерева;
PROCEDURE ПроходДерева(Дерево:ТипУказатель;n:INTEGER);
BEGIN
  StdLog.Int(Дерево.Число);
  IF n<4 THEN
    ПроходДерева(Дерево.Левый,n+1);
    ПроходДерева(Дерево.Правый,n+1);
  END;
END ПроходДерева;
PROCEDURE Главная*;
VAR
  Дерево:ТипУказатель;
BEGIN
  NEW(Дерево);
  ПостроениеДерева(Дерево,1);
  ПроходДерева(Дерево,1);
END Главная;
END Модуль.

```

Процедура **ПостроениеДерева** выполняет построение дерева. Заметим, кстати, что длины всех ветвей дерева одинаковы. Такое дерево называется сбалансированным и играет большую роль в алгоритмах поиска. Процедуры построения дерева и его обхода совершенно идентичны, отличие только в том, что процедура обхода переходит к новой вершине, полагая её уже существующей, в то время как процедура построения должна вершину создать.

Конечно же, эта программа не имеет большой прикладной ценности. Действительно, полезное дерево, возникающее из прикладной задачи, скорее всего, будет иметь различное количество вершин-потомков и различную глубину, то есть быть несбалансированным. Предположим, построенное дерево состоит из вершин, имеющих не более двух потомков (могут существовать вершины с одним потомком и вообще тупиковые). Не вдаваясь в вопрос, как такое дерево может быть построено, запишем процедуру его обхода:

```

PROCEDURE ПроходДерева(Дерево:ТипУказатель);
BEGIN
  StdLog.Int(Дерево.Число);
  IF Дерево.Левый#NIL THEN ПроходДерева(Дерево.Левый); END;
  IF Дерево.Правый#NIL THEN ПроходДерева(Дерево.Правый); END;
END ПроходДерева;

```

Различие очевидно. Путешествие вглубь дерева по каждому направлению про-

должается до тех пор, пока это возможно. Такая процедура способна пройти любое двоичное дерево, даже его вырожденный случай – линейный список. Заметим только, что, как и в первом примере, левое направление является предпочтительным, путешествие вправо начинается только тогда, когда продолжать ветвь влево уже невозможно.

В отношении дерева также можно выделить некоторые базовые операции. К вершине можно подклеить целое дерево или вершину. Это тривиальная операция. Достаточно вершине-предку сообщить адрес нового потомка. Такая же тривиальная операция – уничтожение вершины или поддерева. Достаточно дойти до вершины-источника уничтожаемого поддерева и присвоить соответствующему указателю значение `NIL`. Если речь идет о поддереве, то после присвоения `NIL` будет уничтожен корень поддерева и затем, спустя некоторое время, сборщиком мусора будет уничтожено и все поддерево.

В заключение. В качестве нетривиальной операции, не сводимой к операциям над линейными списками, приведем пример удаления узла без удаления привязанного к нему поддерева. Рассмотрим простое дерево (рис. 8.5):

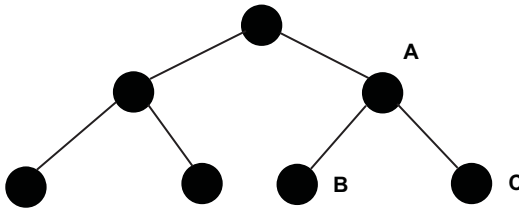


Рис. 8.5

В чем здесь проблема? Предположим, стоит задача удаления узла *A*. Но узлы *B* и *C* должны быть сохранены. Это означает, что один из узлов необходимо подклеить к корню, а второй сделать его дочерним. Например, вот так:

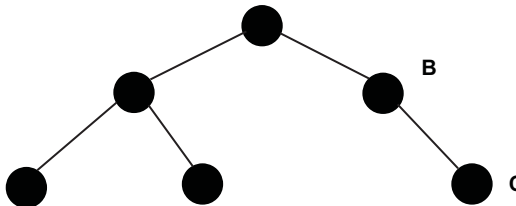


Рис. 8.6

Рисунок 8.6 показывает выход из положения при условии, что вершина *B* тупиковая, из неё не выходит ни одной ветви. Такая процедура удаления может быть применена и в том случае, если каждая вершина содержит не фиксирован-

ное количество указателей, а связный список, в этом случае процедура удаления немного усложняется технически, но принципиально та же. Если же количество указателей фиксировано, то процедура удаления существенно усложняется.

Операции вставки дерева в дерево, удаления ветвей тесно связаны с проблемами поиска данных на дереве. Выше был упомянут термин «сбалансированное дерево». Этот термин означает, что все ветви дерева имеют одинаковую длину. Реально на практике под сбалансированным деревом понимают дерево, в котором длина любых двух ветвей отличается не более чем на единицу. Поиск данных проще всего осуществлять именно на сбалансированных деревьях, так как для таких деревьев математическое ожидание (суть термина посмотрите в приложении) длины пройденного пути минимально. Операции вставки и удаления могут нарушить сбалансированность дерева. Вопросы восстановления баланса пока не входят в поле зрения данной книги, но если вам это интересно, то можете обратиться к книге Вирта [15].

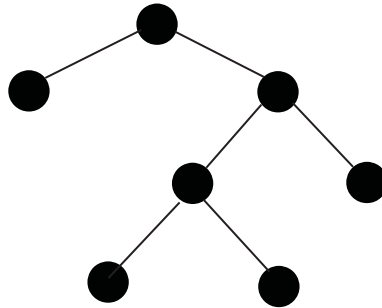


Рис. 8.7

Алгоритмы принятия решений

Постановка задачи. Понятие эвристического алгоритма	238
Оценочная функция	240
Метод минимакса	241
Альфа-бета алгоритм	245

Постановка задачи. Понятие эвристического алгоритма

Первые компьютеры и первые программы были предназначены для решения вычислительных задач. Понятие вычислительной задачи, правда, очень широко. Можно численно считать интеграл, искать сумму числового ряда, можно рассчитывать форму корпуса корабля или самолета, можно считать заработную плату, все это будут вычислительные задачи. Общее в них – то, что существует исходный набор данных, строго определенная последовательность алгоритмических действий и вполне определенный числовой результат.

Существует другой сорт задач. Назовем такие задачи условно играми. Их главная особенность – наличие соперников с трудно предсказуемым поведением. Такого рода задач довольно много. И соперником может быть не обязательно человек. Приходится вести игру руководителям коммерческих предприятий, здесь против них играет не человек и не отдельное предприятие-конкурент, а вся рыночная среда. Наверное, с задачей такого рода пришлось иметь дело разработчикам программного обеспечения луноходов. Соперник лунохода – окружающая среда спутника Земли.

Программу, ведущую игру, нельзя представить в виде некоторого количества операторов выбора, жестко реагирующих на строго определенный набор ситуаций, скорее, такая программа должна уметь принимать решение в условиях острой нехватки информации. Оказывается, это возможно, и более того, основной принцип принятия решений удивительно прост. Однако прежде ограничим задачу, для того чтобы последующие рассуждения не были слишком размыты.

Более точная постановка задачи. Во-первых, положим, что игра регулируется небольшим количеством простых правил, во-вторых, согласимся, что каждый из игроков обладает всей полнотой информации об игровой ситуации. Такие игры называются играми с полной информацией. Примеры игр с полной информацией: шахматы, шашки, в принципе, наверное, почти все известные игры на досках можно отнести к играм с полной информацией. Пример игры, не удовлетворяющей поставленным условиям, – любая карточная игра. В карточных играх нарушается второй принцип. Игроки не владеют полной информацией о картах, находящихся на руках соперников.

Начнем анализ игры с полной информацией. Задача стоит так: дана некоторая игровая позиция (возможно, исходная), необходимо найти ход (принять решение), либо ведущий к победе, либо в худшем случае к ничьей. Принципиально задача разрешима. Необходимо построить полное дерево вариантов, просмотреть все конечные позиции и определить ход, заканчивающийся нужным образом при любых действиях соперника. Это принципиально. Посмотрим, что происходит в действительности.

Для этого представим себе некую абстрактную игру, которую назовем двоичной. Её двоичность заключается в том, что из каждой позиции возможно ровно 2 продолжения. Пусть двоичная игра длится 40 ходов (достаточно короткая шах-

матная партия). 40 ходов – это пара: один ход первого игрока и один ход второго игрока. То есть 80 ходов. Каждый ход создает два варианта продолжения. Следовательно, конечных позиций для анализа 2^{80} . Число астрономического масштаба. Если учесть, что в реальных играх, в тех же шахматах игра может длиться существенно дольше и количество продолжений из каждой позиции существенно больше, чем два, то вариантов продолжений настолько много, что принципиальная разрешимость игровой задачи так принципиальной и остается.

Спасает положение тот факт, что невозможность найти точный ответ не означает невозможности найти *достаточно хороший ответ*. Если игрок готов пожертвовать качеством игры, то взамен он получает возможность не слишком углубляться в анализ ситуации. Поясним сказанное на примере шахмат. Предположим, черный король остался один против двух белых ладей. Ясно, что это проигрыш, но ясно также, что это еще не конец игры. Игра еще продолжится некоторое количество ходов, но проводить анализ до конечной позиции уже нет необходимости. В общем виде идея звучит так: если игрок через N ходов получает хорошую позицию, то есть сильная уверенность, что качество позиции сохранится некоторое время и после N -го хода. Это не обязательно, но вполне вероятно.

Таким образом, игрок может поставить перед собой задачу поиска достаточно хорошего решения вместо наилучшего, такое решение называют *эвристическим*, а метод, позволяющий его получить, – *эвристикой*.

Эвристика – это основанное на опыте правило, существенно ограничивающее поиск решения в сложной задаче. Эвристика не гарантирует оптимальность полученного решения, полезная эвристика предлагает решения, которые с высокой степенью вероятности оказываются хорошими или точнее достаточно хорошими. Соответственно эвристический алгоритм – это алгоритм, пользующийся при решении задач эвристикой.

В первой главе, посвященной парадигме структурного программирования, мы в качестве примера как раз использовали эвристический алгоритм. Можете вернуться к первой главе и еще раз пробежать описание алгоритма построения расписания. На каждом своем шаге этот алгоритм выбирает для распределения такое занятие, для которого риск конфликта наиболее высок. Это выглядит разумно, но совершенно не гарантирует, что принятое решение не создаст конфликта в некотором будущем. Такое, «вероятно, правильное» решение есть решение эвристическое. Эвристическое поведение демонстрирует и шахматист, просчитывая варианты игры на некоторое количество ходов вперед. Возможно, его анализ на N ходов и дает хороший прогноз, но не гарантирует успеха на $N+1$ ходе. Эвристическое поведение демонстрирует менеджер, планирующий деятельность компании на некоторый временной срок. В общем, необходимо заметить, что ситуаций, в которых невозможно найти точное решение, но требующих разумного эвристического поведения, очень и очень много. Попробуем выполнить анализ составляющих эвристического игрового поведения.

Оценочная функция

Принятие решения опирается на представление игрока о том «что такое хорошо и что такое плохо». Он стремится к хорошим ситуациям и старается избежать плохих. Чтобы такое стремление можно было алгоритмизировать, качество позиции должно быть оценено числом. Игрок-программа должен иметь формулу, посредством которой для каждой ситуации можно вычислить число, которое тем больше, чем лучше ситуация. Тогда игрок, перебрав некоторое количество ситуаций, может выбрать ту, для которой вычисленное значение имеет наибольшее значение. Такая формула называется оценочной функцией. Для того чтобы понять, как строится оценочная функция, обратимся к шахматам, в надежде, что вы имеете некоторую игровую практику.

Введем понятие игрового фактора. Игровой фактор – это некоторое качество, влияющее на оценку позиции. Например, наличие фигур и пешек – безусловно, фактор, влияющий на оценку позиции даже безотносительно от их положения. Также ясно, что пешка или ферзь – это совершенно разные факторы. Также неодинаково качество коня, или ладьи, или слона. Это порождает естественную идею присвоить каждому фактору некоторое число. Например: пешка = 1; конь = 2; слон = 3; ладья = 4; ферзь = 5. Тогда наличие на доске двух пешек даст 2 балла, наличие одной ладьи – 4 балла и т. д. И, подсчитав количество фигур, мы сможем дать ситуации количественную оценку.

Естественно, то, как мы присвоили фигурам числовые значения, не даст никакой осмысленной оценки. Эти значения взяты, что называется с потолка. К сожалению, реально нет математических методов, позволяющих оценить роль того или иного игрового фактора. В действительности для оценки необходимо привлекать высококвалифицированных экспертов, владеющих данной игрой и способных оценить относительную силу выделенных факторов. Кроме того, мы для примера взяли очень примитивные факторы. Любой, даже не слишком опытный шахматист скажет, что две ладьи сильнее одной не в два раза, а более. Сдвоенных слонов можно выделить в отдельный фактор. Рядом стоящие пешки сильнее изолированных, даже при равном их количестве. Это так называемые позиционные факторы. Значимость фактора может зависеть от игровой фазы. Ценность ладьи растет по мере того, как освобождается доска. Конь сильнее при ограниченном игровом пространстве, так как может скакать через фигуры, а сила слона резко возрастает при необходимости игры на два фланга. Все эти примеры приведены только для того, чтобы показать, что даже игра ограниченным количеством фигур на ограниченной доске порождает большое количество взаимозависимых факторов. Точно так же обстоит дело и в любой другой игре.

Но независимо от сложности игры и её природы метод построения оценочной функции один. Нужен эксперт как можно более высокой квалификации, и желательно не один. Комиссия экспертов должна выделить факторы, влияющие на игру, присвоить им определенные веса, после чего программисту для оценки ситуации будет достаточно вычислить оценочную функцию следующего вида:

$$F = \sum_k n_k f_k ,$$

где индекс k пробегает по всему множеству факторов, n_k – количество единиц данного фактора (например, количество пешек для фактора «пешка»), f_k – вес фактора. Нетрудно заметить из формулы, что мы упростили ситуацию, предположив все используемые факторы взаимонезависимыми. Но для пояснения базовых идей этого достаточно. Итак, первая базовая идея – это оценочная функция. Вторая базовая идея – это специальная организация обхода дерева перебора игровых вариантов.

Метод минимакса

Множество вариантов, возникающее при анализе игры на некоторое количество ходов, можно выстроить в виде дерева перебора. Вернемся для упрощения рассуждений к двоичной игре. Пусть выполняется анализ из некоторой позиции на два хода вглубь. Первый ход – за игроком, второй – за оппонентом. Дерево перебора в этом случае заканчивается четырьмя возможными позициями. Предположим, эти позиции удалось оценить так, как это показано на рис. 9.1.

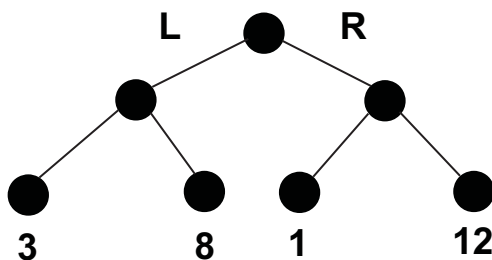


Рис. 9.1

У игрока есть два варианта продолжения. Ход влево обозначен буквой **L**, и ход вправо обозначен буквой **R**. Какое ему стоит выбрать продолжение? Жадная стратегия подсказывает, что предпочтителен ход **R**. В этом случае возможно получение позиции с максимально возможной оценкой 12. Однако "возможно" не означает, что это получится. Необходимо учесть, что следующий ход – за противником, и надо полагать, разумный противник в ситуации **R** выберет ход, ведущий к позиции с оценкой 1, а не 12.

Целесообразно предположить, что противник будет играть наилучшим образом, на языке оценок это означает следующее: противник предложит игроку выбор из наихудших оценок, и уже из этих наихудших игрок имеет право выбрать лучшую. Поэтому метод выбора и называется методом минимакса (выбор максимальной оценки из минимальных). Техника расчета варианта такова: оцениваются конечные позиции дерева перебора. Далее выполняется подъем оценок по следующему правилу:

- если текущий ход за игроком, то из имеющегося множества оценок выбираются лучшие, и они формируют новое множество;
- если текущий ход за противником, то из множества выбираются худшие, и они формируют новое множество.

Чтобы лучше понять механизм, рассмотрим еще один пример (рис. 9.2):

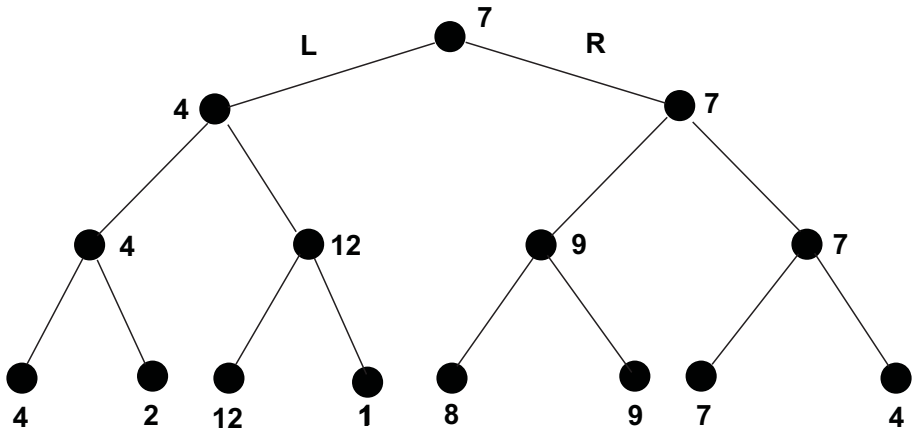


Рис. 9.2

Анализируемый набор пар – это пары нижнего уровня. На уровне выше решение о ходе принимается игроком, поэтому на родительские вершины поднимаются наибольшие оценки. На следующем уровне решения принимает противник, поэтому дальше уходят наихудшие оценки. Самый верхний уровень – это опять уровень игрока, и он принимает решение о ходе вправо, так как этот ход дает наилучшую оценку из оставшихся, то есть оценку 7.

Пример реализации минимакса. Реализуем процедуру минимакса для двоичной игры глубины N . Значения оценочной функции пусть задаются случайной функцией.

Реализация, очевидно, должна состоять из двух процедур: во-первых, дерево двоичной игры необходимо построить, во-вторых, к построенному дереву должна быть применена процедура минимакса.

Построение дерева. Построение дерева сводится к построению поддеревьев от каждой вершины. Это рекурсивное определение, реализуем его рекурсивной процедурой – **ПостроениеДерева**. На вход подадим текущую глубину, которая с каждой активацией уменьшается на 1, и адрес уже созданной вершины. И это происходит до тех пор, пока переменная, имеющая смысл глубины, не станет равна нулю. Если глубина нуль (достигли основания дерева), то вычисляем вес вершины как случайную величину. Если глубина не нуль (не достигли основания), то создаем две вершины и уходим дальше вглубь по дереву:

```
IF N=0 THEN
  (*основание достигнуто, вычисляем вес*)
```

```

Указатель.Вес:=R.Int(10);
  (*Распечатка веса для контроля*)
  StdLog.Int(Указатель.Вес);
ELSE
  (*Создаем правую вершину и выполняем рекурсивную активацию для её обработки*)
  NEW(Указатель.Правый);
  ПостроениеДерева(Указатель.Правый,N-1);
  (*Создаем левую вершину и выполняем рекурсивную активацию для её обработки*)
  NEW(Указатель.Левый);
  ПостроениеДерева(Указатель.Левый,N-1);
END;

```

Процедура минимакса. Каждая активация минимаксной процедуры должна получать от более глубоких слоев дерева два значения, из которых она выберет либо наибольшее, либо наименьшее значение. За исключением последней активации, работающей на самом глубоком уровне. Там в конце ветки дерева есть только одно значение, которое в любом случае необходимо передать на верхний уровень:

```

IF N>0 THEN
  (*Обработка двух значений*)
ELSE
  (*Возврат единственного значения с нижнего уровня*)
  RETURN Указатель.Вес;
END;

```

Для обработки двух значений их необходимо получить от последующих активаций:

```

Правый:=Минимакс(Указатель.Правый,N-1);
Левый:=Минимакс(Указатель.Левый,N-1);

```

Из этих двух необходимо выбрать одно. Либо наименьшее, либо наибольшее. Вопрос выбора зависит от номера вызова. Если на некоторой активации выбиралось наименьшее из двух, то далее должно выбираться наибольшее. Этот выбор можно привязать к четности номера вызова:

```

IF N>0 THEN
  Правый:=Минимакс(Указатель.Правый,N-1);
  Левый:=Минимакс(Указатель.Левый,N-1);
  IF N MOD 2=0 THEN
    (*Скажем, здесь очередь первого игрока*)
    (*Выбираем наибольшее из двух*)
    IF Правый>Левый THEN RETURN Правый ELSE RETURN Левый; END;
  ELSE
    (*Тогда здесь очередь второго*)
    (*Выбираем наименьшее из двух*)
    IF Правый>Левый THEN RETURN Левый ELSE RETURN Правый; END;

```

```

    END;
ELSE
    RETURN Указатель.Вес;
END;

```

Осталось привести полный текст реализации.

Листинг 9.1

```

MODULE Модуль;
    IMPORT In, StdLog, R := Info21sysRandom;
    TYPE
        ТипУказатель=POINTER TO ТипЗапись;
        ТипЗапись=RECORD
            Вес:INTEGER;
            Левый,Правый:ТипУказатель;
        END;
    PROCEDURE Минимакс(Указатель:ТипУказатель;N:INTEGER):INTEGER;
    VAR
        Правый,Левый:INTEGER;
    BEGIN
        IF N>0 THEN
            Правый:=Минимакс(Указатель.Правый,N-1);
            Левый:=Минимакс(Указатель.Левый,N-1);
            IF N MOD 2=0 THEN
                IF Правый>Левый THEN RETURN Правый ELSE RETURN Левый; END;
            ELSE
                IF Правый>Левый THEN RETURN Левый ELSE RETURN Правый; END;
            END;
        ELSE
            RETURN Указатель.Вес;
        END;
    END Минимакс;
    PROCEDURE ПостроениеДерева(Указатель:ТипУказатель;N:INTEGER);
    BEGIN
        IF N=0 THEN
            Указатель.Вес:=R.Int(10);
            StdLog.Int(Указатель.Вес);
        ELSE
            NEW(Указатель.Правый);
            ПостроениеДерева(Указатель.Правый,N-1);
            NEW(Указатель.Левый);
            ПостроениеДерева(Указатель.Левый,N-1);
        END;
    END ПостроениеДерева;
    PROCEDURE Главная*;

```

```
VAR
  N:INTEGER;
  Указатель:ТипУказатель;
BEGIN
  NEW(Указатель);
  In.Open;
  In.Int(N);
  ПостроениеДерева(Указатель,N);
  StdLog.Ln;StdLog.String('Результат');
  StdLog.Int(Минимакс(Указатель,N));
END Главная;
END Модуль.
```

Итак, минимакс позволяет найти хорошее эвристическое решение. Его качество зависит от двух вещей: от качества оценочной функции и от глубины дерева перебора. Какой из этих двух факторов важнее, трудно сказать. Разработчики игр на досках (шахматы, го, шашки и т. д.) стремятся больше нагрузки возложить на оценочную функцию.

Возможно, существуют игры, для которых реально построить оценочную функцию, исчерпывающе описывающую игру, но это, видимо, дело исключительной сложности, и полностью избавиться от необходимости анализа дерева перебора практически нельзя. А дерево перебора даже простой двоичной игры плодит варианты с космической скоростью, и рассчитывать на большую глубину перебора, даже имея мощный компьютер, – не вполне серьезное занятие. Частичное решение проблемы заключается в том, что, оказывается, ветви дерева не равнозначны. Некоторые из них вполне вероятны, а некоторые можно отрезать от дерева как маловероятные.

Кстати, большие мастера могут строить очень эффективные эвристические оценки. Однажды Х. Р. Капабланку (чемпиона мира по шахматам) спросили, насколько глубоко он анализирует позицию. О нем в то время ходили легенды, утверждавшие, что он продумывает позицию на десятки ходов. Ответ был неожиданным: «На один», – ответил Капабланка. Возможно, это было сказано для красного словца, но доля правды в том есть. Последние годы вошли в моду поединки между суперкомпьютерами и лучшими шахматистами планеты. Люди против суперкомпьютеров выступают вполне успешно, хотя ясно, что ни один шахматист-человек и близко не стоит рядом с компьютером в возможностях счета вариантов. Видимо, здесь дело в умении оценивать ситуацию, владеть качественной эвристикой. А сейчас вернемся к проблеме отсеечения маловероятных вариантов.

Альфа-бета алгоритм

Альфа-бета алгоритм позволяет отсекаать ветви дерева из некоторых естественных предположений о силе игроков. Мы начали свой анализ из соображения, что противник будет играть наилучшим образом. Естественно, то же самое разумно

предположить и о том участнике, которого мы называем игроком. Оба участника игры выкладываются по максимуму своих возможностей. Следующее разумное предположение: силы игроков равны. Обоснуем целесообразность этого допущения. Вы игрок. Предположим, вы ошиблись в своем предположении и на самом деле противник слабее вас, тогда, ожидая сильной игры, вы будете играть сильно и выиграете. И наоборот, предположим, вы ошиблись в противоположную сторону и ваш противник сильнее вас, тогда ваша игра по максимуму все равно приведет к поражению. Следовательно, предположив равенство сил, вы по крайней мере не ухудшите своего положения. Кстати, из этого следует еще один любопытный вывод. Если игра предоставляет игрокам равные шансы и игроки равны по силе, то единственное чем может закончиться игра, – это ничья. *Следовательно, при равных шансах победа возможна только в случае расчетной ошибки.* Это, кстати, дает большие преимущества компьютерной программе. Если сила игры программы равна силе её оппонента-человека, то вероятность победы программы все же выше, так как она в рамках своей оценочной функции не ошибается.

Предположение равенства силы игроков и безошибочность их игры означают, что игрок не может сделать хода, который даст ему серьезный перевес в виде большого значения оценочной функции, его противник этого не позволит. С другой стороны, его противник не сможет нанести ему слишком большого ущерба, иначе говоря, не должно быть хода дающего, слишком маленькое или слишком большое значение оценочной функции. Неправдоподобно маленькое значение оценки обозначим буквой α , а неправдоподобно большое – буквой β . α и β – это соответственно нижняя и верхняя границы оценочной функции. Вершины дерева (позиции игры), чьи оценки выходят за эти рамки, объявляются неправдоподобными и отрезаются вместе с поддеревом вариантов, построенным на этой вершине.

Наверное, ясно, что речь идет не о конечных оценках, а о промежуточных. Для использования альфа-бета интервала необходимо считать оценку в каждой позиции при построении дерева перебора. Если посчитанная оценка попадает в альфа-бета интервал, то дерево строится дальше, иначе вершина и все построенное на ней дочернее дерево выпадают из минимаксного анализа. Выравнивание оценок, стремление рассматривать в идеале только равные ситуации означает доведение до крайности идеи игры на ничью. Но это не означает отказа от выигрыша при ошибке противника. Альфа-бета интервал не обязательно одинаково выгоден для обоих игроков. Верхняя и нижняя границы – всего лишь наиболее вероятные ограничения оценочной функции. Они ничего не говорят о выгоде или невыгоде позиции для того или иного игрока. Можно лишь сказать, что в равной ситуации эти границы имеют значения, не дающие игроку преимущества, а в случае ошибок противника альфа-бета значения могут изменяться в сторону увеличения, в случае ошибок игрока они могут изменяться в сторону уменьшения.

Задание для самостоятельной работы

Дополните реализацию алгоритма минимакса учетом альфа-бета границ.

Комбинационный удар. Альфа-бета алгоритм имеет один существенный недостаток. Он безусловно минимизирует риски, но вместе с тем минимизирует

еще одну важную вещь. Вернемся к шахматам. В шахматах, да и в других играх на досках, существует понятие комбинации. Комбинацией называется последовательность ходов, ведущая, возможно, к потерям в течение некоторого времени (уменьшению оценочной функции), а затем форсированно (термин, означающий невозможность противостоять этому со стороны противника) выполняется последовательность ходов, резко увеличивающая оценочную функцию и в итоге обеспечивающая серьезный перевес.

Вывод о минимаксе. Ясно, что и минимаксная процедура, и альфа-бета оценка направлены на ничейный результат как единственно возможный, при равных начальных условиях и правильной игре обоих партнеров. Тогда вследствие чего возможен выигрыш? Источника два. Во-первых, для существующих игр, даже строго определенных, таких как шахматы и шашки, вопрос о равенстве возможностей надо признать открытым. Строгой теории игры не существует ни для игры вообще, ни для отдельных игр. Есть некоторые гипотезы, более или менее достоверные. Например, считается, что белые фигуры в шахматах (белые ходят первыми) имеют больше возможностей для атаки, первый игрок го имеет значительный перевес, настолько значительный, что второму игроку полагается фора в несколько камней. Преимущество первого хода в рэндзю настолько велико, что для компенсации этого преимущества на первого игрока накладывается ряд ограничений, называемых правилами фолла (запретные ходы).

Но это все гипотезы, полученные эмпирически, как результат накопленной игровой практики. Совершенно не ясно, как такие гипотезы можно было бы доказать или опровергнуть. Во всяком случае, фактор первого хода влияет на игру только в случае встречи двух мастеров. Для подавляющего большинства игроков этот фактор не имеет ровным счетом никакого значения.

Вторая возможность для выигрыша заключается в неизбежности ошибки. Ошибка расчетов действительно неизбежна, и не в силу невнимательности игрока, а в силу недостаточной глубины дерева перебора. Минимакс обеспечивает более-менее равные шансы на заданную глубину, далее все непонятно.

В чем тогда роль минимакса? Ответ следующий. Метод минимакса направлен на максимально выгодное позиционное развитие, на так сказать накопление атакующего потенциала. При неудачных действиях противника хорошая позиционная игра приведет к увеличению атакующего потенциала настолько, что станет возможен тактический, комбинационный удар, скачком увеличивающий оценочную функцию.

Дополнительно о тактическом ударе. Выше говорилось о комбинации, но только о её частном случае – комбинации с жертвой. Комбинация (далее будем говорить: тактический удар) не сводится к жертве и последующему форсированному мщению. Возможна, например, двойная угроза, когда игрок одними и теми же фигурами создает несколько угроз, отбить которые его противник не может одновременно. Или, например, возможна ситуация, когда игрок, атакуя, связывает силы противника на одном участке поля, а затем быстро перегруппировывается и организует атаку на другом, при условии что противник не располагает такими же возможностями для переброски сил.

Мы не будем сейчас давать какую-то классификацию тактических ударов, заметим, только, что если стратегическая борьба направлена на накопление атакующего потенциала, то тактический удар нужен для его реализации. Процедура построения атаки не укладывается в процедуру минимакса, следовательно, игра должна быть поделена на две фазы: стратегическая (минимаксный поиск лучшей оценки) и тактическая – поиск завершающего удара. Это, конечно, упрощение, чаще всего игра состоит из нескольких периодов стратегической борьбы, прерывающихся тактическими ударами различной значимости. А сейчас два важных вопроса:

- критерий возможности тактического удара;
- поиск последовательности ходов для тактического удара.

Второй вопрос более простой. Если есть предположение, что возможна комбинация, то для её обнаружения необходимо один раз честно все посчитать, то есть просмотреть дерево перебора на большую, чем обычно, глубину без учета альфа-бета границ. Впрочем, и при расчете тактического удара возможно отбросить часть дерева. Комбинация всегда локализована на доске, направлена на определенный объект, преследует конкретную цель, ограниченную определенной территорией. Если определить фигуры, участвующие в будущей комбинации, или территорию доски, на которой будут разыгрываться события, то все, что происходит за пределами этой территории или не с этими фигурами, можно игнорировать. В позиционных играх, в которых фигуры не перемещаются по доске, таких как го и рэндзю, тактический удар можно привязать к территории. В играх, разрешающих движение фигур, расчет комбинации лучше привязать к фигурам.

Вопрос о том, какие фигуры принять в расчет или на какой территории будут развиваться события, тесно увязан с определением самой возможности тактического удара. Определение такой возможности – вопрос действительно сложный. Ни один мастер комбинационной игры не расскажет вам строгой красивой теории, скорее всего, вы услышите в ответ что-то мало понятное, замешанное на личных ассоциациях. Видимо, можно утверждать, что видение красивого тактического удара связано с развитой интуицией. Конечно, есть и некоторые грубые критерии, например превосходство в силах на некоторой территории. Кстати, успех военных действий в значительной степени определяется способностью военачальника создать на одном важном направлении численный перевес, однако из истории достаточно много известно примеров успешных полководцев, одерживавших победы над превосходящими силами противника. Видимо, дело обстоит существенно сложнее, чем может показаться на первый взгляд.

Тема организации тактического удара, комбинации очень сложна и интересна, но, пожалуй, это тема отдельной книги, а не небольшой главы, поэтому мы наш рассказ завершим, несмотря на то что рассказ выглядит весьма незаконченным.

В заключение. Изложенные в главе идеи необходимо рассматривать как идеи общего характера. Приступая к анализу, вы знаете, что необходимо составить оценочную функцию и организовать минимаксный обход дерева вариантов. Но как именно решить эти вопросы, изложенная здесь теория не поясняет. В действительности все сколько-нибудь сложные вопросы упираются в природу игры и

только ей присущие внутренние законы. Однако надо заметить, что написать программу, демонстрирующую вполне разумное поведение, несложно. На некотором уровне понимания игры легко выделить небольшое количество факторов, представляющих собой логически цельное описание победной стратегии, и достаточно легко организовать минимакс. Проблемы начнутся в тот момент, когда вполне разумное поведение вашей программы перестанет вас устраивать и возникнет желание написать хорошо играющую программу. Трудности реализации возрастут многократно. Но с этим уже ничего не поделаешь.

Кроме того, необходимо помнить, что множество игр не укладываются в рамки игр с полной информацией. Большинство игр, в которые мы играем в своей жизни, и не только на досках, ведутся без полной информации и иногда даже без жестко определенных правил. И даже если игра все-таки есть игра с полной информацией, не факт, что описанная технология с ней справится. Например, есть широко распространенное мнение, что хорошее решение для игры го в рамках минимаксной технологии невозможно. Поэтому если проблемы программирования игр вас заинтересовали, то рассматривайте данную главу только как начало большого пути.

Алгоритмы на графах

Стратегии обхода.....	251
Построение остовного дерева	253
Алгоритм поиска компонент связности.....	263
Волновой алгоритм	265
Алгоритм Дейкстры.....	269
Алгоритм Флойда.....	276
Нахождение максимального потока	280

Стратегии обхода

Реализованные ниже алгоритмы посвящены следующим темам: построение компоненты связности, построение остовного дерева, поиск кратчайших путей, расчет максимального потока. Все алгоритмы реализованы для неориентированных графов. А так как разница между ориентированным и неориентированным графом выражается в симметрии их матриц смежности, такое различие можно рассматривать как непринципиальное, поэтому вы можете попробовать самостоятельно изменить наши реализации под ориентированные графы. Все алгоритмы главы опираются на представление графа в виде матрицы смежности. Запомните этот технический момент.

Если не все, то очень большое количество задач, сформулированных в терминах графов, требуют полного прохода всех вершин (ребер, дуг) графа. При обходе графа можно вычислять разные величины и решать различные содержательные задачи, но и сам обход – не вполне тривиальная проблема. В алгоритмах, рассмотренных ниже, вопросы обхода будут так или иначе решаться, будут продемонстрированы конкретные технические приемы. Но все многообразие алгоритмической техники в этом вопросе сводится к двум принципиально различным методам: обход графа в ширину и обход графа в глубину. И прежде чем переходить к алгоритмам, выполняющим содержательную работу, обсудим эти два общих подхода.

Обход графа в ширину

Такого типа обход похож на движение волны. На примере ниже (рис. 10.1) волна начинает движение от вершины, залитой черным цветом. Этой вершине смежны две вершины графа, и их волна заливает на следующем шаге, после чего образуется новый фронт волны, состоящий из уже двух вершин. На втором шаге волна заливает все остальные вершины.

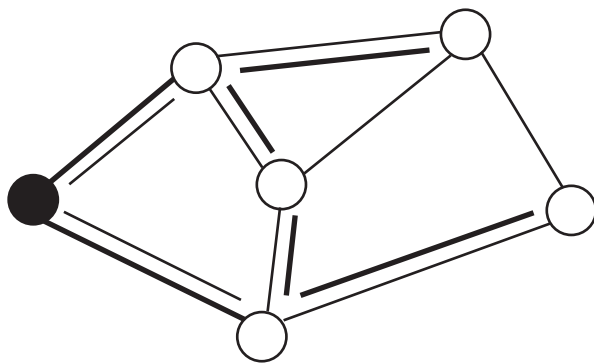


Рис. 10.1

Более тонкой линией обозначена первая итерация, которая включает во фронт волны две вершины, и уже вторая итерация, действие которой обозначено более

толстой линией, завершает обход вершин графа. Заметим еще одну особенность. Существуют вершины, в которые можно попасть на некотором шаге из двух и более вершин, но обход в ширину делает это единожды из какой-нибудь одной. Важный вопрос о выборе пути в вершину, достижимую из нескольких вершин фронта, не есть вопрос общего метода. Это содержательная проблема, решается она, исходя из потребности конкретной задачи.

Немного уточним процедуру обхода. Ключевой технический момент – это наличие фронта волны. До завершения процесса множество вершин графа можно разделить на три подмножества: подмножество, через которое волна уже прошла, подмножество вершин, достигнутых на последнем шаге (волна пришла в вершину, но еще не ушла из неё), и подмножество вершин, еще не достигнутых волной. Для построения пути волны важно именно множество фронта. Это множество может быть представлено обычным массивом, перестраиваемым по следующим правилам:

- вершина, в которую пришла волна, включается в массив фронта;
- вершина, из которой волна ушла, исключается из массива.

Итерационный шаг заключается в следующем: для каждой очередной вершины массива фронта определяются вершины, смежные с ней и еще не достигнутые волной. Найденные смежные вершины включаются в массив фронта, после чего очередная вершина из массива убирается.

Такой массив удобно организовать как двустороннюю очередь, тогда, например, вершину, находящуюся в одном конце очереди можно считать очередной для обработки, а добавлять вершины в другой конец очереди. Организация специальной очереди не обязательна. Как выше уже было замечено, все вершины графа делятся на три подмножества. Введем обозначения: 1 – вершина не достигнута волной; 2 – вершина находится во фронте волны; 3 – волна ушла из вершины. Очередную итерацию можно записать на псевдокоде следующим образом:

```
Для всех вершин графа
  Если статус Очередной вершины = 2 То
    Для всех вершин смежных Очередной
      Если статус смежной = 1 То
        Статус смежной = 2
    Статус Очередной = 3
```

Задание для самостоятельной работы

Перепишите данный псевдокод на КП, с учетом того, что граф задан матрицей смежности.

Если есть фрагмент, описывающий очередную итерацию, то весь процесс можно построить как цикл, условием завершения которого будет отсутствие вершин со статусом ниже 3, а начинается процесс с некоторой вершины-источника, которой первой присваивается статус 2.

Обход графа в глубину

Обход графа в глубину – это попытка уйти по графу как можно дальше, насколько это возможно, и если на каком-то шаге дальнейший путь оказывается невозможным (тупик), то выполняется возврат до тех пор, пока не будет обнаружено еще не пройденное ребро. *Тупик – это вершина, из которой либо не выходит ни одно ребро, либо все они уже пройдены.* Процесс обхода в глубину можно организовать рекурсивной процедурой. Попад в очередную вершину, обходчик графа обнаруживает перед собой ту же картину, которую он видел в предыдущей вершине, – некоторое количество еще не пройденных ребер. Таким образом, переход от вершины к новой вершине – это переход от задачи прохода графа к задаче прохода графа, но от нового источника, а это рекурсивная постановка. Ниже приведен пример (рис. 10.2):

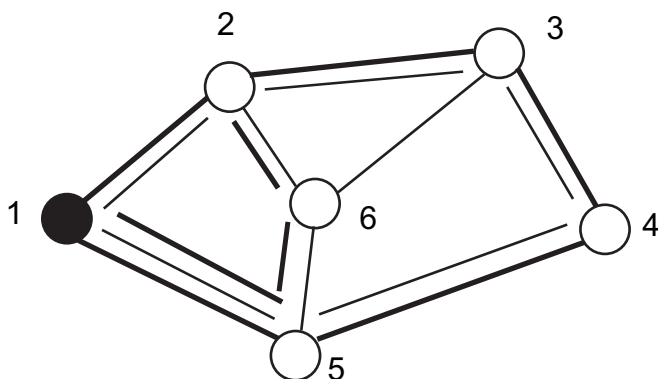


Рис. 10.2

Здесь не показан весь процесс. Обход вглубь начинается от вершины с номером 1 и идет до тупика, то есть до вершины с номером 1, в которую обходчик попадает из пятой. Обнаружив тупик, обходчик возвращается в пятую и продолжает свой путь через шестую вершину до второй, на которой он опять встречает тупик. На этом шаге граф можно считать пройденным, так как обходчик побывал во всех вершинах. Но для обхода в глубину может быть поставлена задача не просто обхода всех вершин или ребер, а полный перебор всех путей по графу, в этом случае полный путь есть некоторое дерево.

Задание для самостоятельной работы

Постройте дерево перебора всех путей в глубину для графа с рис. 10.2.

Построение остовного дерева

Остовное дерево – это подграф взвешенного графа, содержащий все вершины исходного графа, имеющий наименьший суммарный вес из всех таких подграфов и не содержащий циклов, то есть являющийся деревом. Примеры построения остовного дерева можно посмотреть в приложении.

Ниже мы рассмотрим два алгоритма построения такого дерева, алгоритм Прима и алгоритм Краскала. Оба алгоритма разбирают граф на составные части и собирают из них новый граф, являющийся остовным деревом исходного. Алгоритм Прима собирает остовное дерево, добавляя к уже имеющемуся еще одну вершину, а алгоритм Краскала наращивает компоненты связности, добавляя на каждой итерации еще одно ребро, соединяя две уже построенные компоненты, пока все компоненты не сольются в одну. Оба алгоритма работают со связными графами.

Алгоритм Прима

Формулировка задачи. Дан взвешенный граф, в котором веса присвоены ребрам. Необходимо найти минимальное остовное дерево, имеющее, своим источником одну из вершин графа.

Идея алгоритма. Пусть часть остовного дерева уже построена. Это утверждение всегда верно, так как в начале процесса вершина, с которой начинается построение уже входит в дерево. А если часть остовного дерева уже есть, то множество вершин графа можно разделить на два подмножества: подмножество, состоящее из вершин уже построенного остовного дерева и оставшихся вершин графа.

Очевидно, что среди ребер, соединяющих эти два множества, существует ребро наименьшего веса. Можно доказать (но мы здесь этого делать не будем), что минимальное дерево содержит это ребро. На рис. 10.3 показаны этапы построения остовного дерева алгоритмом Прима.

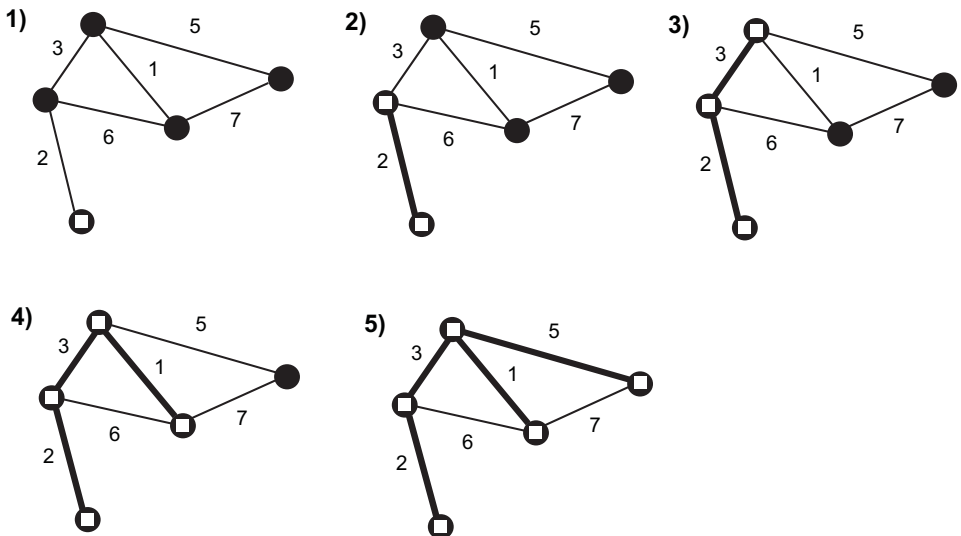


Рис. 10.3.

Числа около ребер — это веса ребер. Белыми квадратиками помечены вершины, включенные в остовное дерево. Черные вершины — еще не включенные. Жирные линии — это ребра, уже включенные в остов.

Реализация. Главный итерационный процесс добавляет на каждом шаге одну вершину к множеству уже входящих в остовное дерево. Следовательно, процесс можно организовать в виде цикла по количеству вершин. Наш первый фрагмент будет выглядеть так:

```
СчетчикОстова:=0;  
WHILE СчетчикОстова<п-1 DO  
    СчетчикОстова:=СчетчикОстова+1;  
END;
```

Здесь переменная *п* есть количество вершин исходного графа. На каждом шаге новая вершина включается во множество вершин остовного дерева. Для хранения такого множества разумно использовать массив. Но массив вершин для представления графа явно недостаточен, так как остовное дерево – это не только вершины, но и ребра. Поэтому, так же как исходный граф изображается матрицей смежности, так же и для остовного дерева необходимо построить матрицу смежности. Для матрицы смежности остова объявлен двумерный массив **Остов[]**.

Для поиска минимального ребра, очевидно, требуется просмотр всех вершин уже существующего остова, так как каждая из них может оказаться инцидентна искомому ребру. Достроим фрагмент:

```
СчетчикОстова:=0;  
WHILE СчетчикОстова<п-1 DO  
    FOR k:=0 TO СчетчикОстова DO  
        (*Поиск вершины*)  
    END;  
    СчетчикОстова:=СчетчикОстова + 1;  
END;
```

Займемся далее телом цикла, анализирующего вершины остова. Очевидно, что каждая вершина может быть инцидентна нескольким ребрам. Следовательно, необходим цикл просмотра всех ребер для каждой вершины. Но может оказаться, что ребро соединяет вершину остовного дерева с еще одной вершиной остова, такие ситуации, естественно, необходимо исключить. Нужен цикл, проверяющий, является ли вершина-кандидат правильной. Запишем его отдельно:

```
flag:=TRUE;  
FOR i:=0 TO СчетчикОстова DO  
    IF j=ВершиныОстова[i] THEN flag:=FALSE;END;  
END;
```

j – номер вершины-кандидата на включение в остов. Массив **ВершиныОстова[]** – это массив вершин, уже включенных в остов. Таким образом, если вершина-кандидат найдется в остова, то флажок – *flag* примет значение *Ложь*, и на этом основании вершину с номером *j* можно будет исключить из рассмотрения. Величина *j* пробегает всех возможных кандидатов от номера = 0 до *п*–1. Далее полный фрагмент:

```

СчетчикОстова:=0;
ВершиныОстова[0]:=0;
WHILE СчетчикОстова<n-1 DO
  min:=max;
  FOR k:=0 TO СчетчикОстова DO
    FOR j:=0 TO n-1 DO
      flag:=TRUE;
      FOR i:=0 TO СчетчикОстова DO
        IF j=ВершиныОстова[i] THEN flag:=FALSE;END;
      END;
    END;
  END;
  СчетчикОстова:=СчетчикОстова + 1;
END;

```

Осталось включить вершину и соответствующие ребра в остовное дерево. Для чего потребуются следующие операции:

- включение вершины-кандидата в массив остовного дерева;
- включение ребра в матрицу смежности остовного дерева;
- исключение ребра из матрицы смежности исходного графа во избежание его повторного участия в следующих итерациях.

Включение вершины в массив не представляет интереса, а для операций над матрицами смежности заметим, что одно и то же ребро участвует в матрице дважды, поэтому операции включения и отбрасывания ребер также необходимо выполнять дважды, для элемента матрицы сверху от главной диагонали и ему симметричного. Так как матрица квадратная, то симметричные элементы определяются простой перестановкой индексов, элемент с индексами x, y симметричен элементу с индексами y, x .

Листинг 10.1

```

MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE АлгоритмПрима*;
VAR
  Граф,Остов:ARRAY 10, 10 OF INTEGER;
  ВершиныОстова:ARRAY 10 OF INTEGER;
  k,j,n,i,min,max,x,y:INTEGER;
  flag:BOOLEAN;
  СчетчикОстова:INTEGER;
BEGIN
  In.Open;
  In.Int(n);max:=0;
  FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO

```



```

    In.Int(Граф[k,j]);
    IF max<Граф[k,j] THEN max:=Граф[k,j]; END;
    Остов[k,j]:=0;
  END;
END;
СчетчикОстова:=0; ВершиныОстова[0]:=0;
WHILE СчетчикОстова<n-1 DO
  min:=max;
  FOR k:=0 TO СчетчикОстова DO
    FOR j:=0 TO n-1 DO
      flag:=TRUE;
      FOR i:=0 TO СчетчикОстова DO
        IF j= ВершиныОстова[i] THEN flag:=FALSE;END;
      END;
      IF (Граф[ВершиныОстова[k],j]#0) &
        (Граф[ВершиныОстова[k],j]<=min) & flag
      THEN
        y:= ВершиныОстова[k];x:=j;min:= Граф[ВершиныОстова[k],j];
      END;
    END;
  END;
  СчетчикОстова:=СчетчикОстова + 1;
  ВершиныОстова[СчетчикОстова]:=x;
  Остов[y,x]:=min;
  Граф[y,x]:=0;
  Остов[x,y]:=min;
  Граф[x,y]:=0;
END;
FOR k:=0 TO n-1 DO
  FOR j:=0 TO n-1 DO
    StdLog.Int(Остов[k,j]);
  END;
  StdLog.Ln;
END;
END АлгоритмПрима;
END Модуль.

```

Замечание о поиске минимального. На каждой итерации главного цикла требуется минимальное значение веса. Алгоритм поиска минимального в массиве известен. Берем первое значение массива как минимальное, затем сравниваем каждый последующий элемент с уже найденным минимальным. В нашем случае, однако, есть небольшая проблема, что взять в качестве исходного минимума. Специально искать первое минимальное ребро можно, но это заметно утяжелит алгоритм. Иногда в таких ситуациях берут число, которое заведомо больше любого числа,

могущего встретиться в процессе поиска. В нашем случае это не получится. Ведь на веса ребер не наложено никаких ограничений. Выход заключается в том, чтобы взять в качестве исходного минимального значения максимальный вес ребра, имеющегося в матрице, а чтобы его не искать каждый раз, это сделано единожды в цикле ввода.

Алгоритм Краскала

Формулировка задачи. Дан взвешенный граф, в котором веса присвоены ребрам. Необходимо найти минимальное остовное дерево.

Идея алгоритма. Алгоритм Краскала подходит к решению задачи иначе, чем алгоритм Прима. Если алгоритм Прима выбирал вершины, которые можно присоединить дешевым ребром к уже построенному дереву, то алгоритм Краскала на каждой итерации выбирает самое дешевое ребро из оставшихся, не заботясь о связности получаемого дерева (но дерево в конце получается таким, как надо).

Для этого ребра графа нумеруем в порядке возрастания весов. Затем для каждого ребра, начиная с первого, проверяем, соединяет оно или нет две несвязные компоненты, если да, то его можно включить в остовное дерево. Ясно, что если мы имеем N вершин, то работа алгоритма начинается с N -несвязных компонент графа (пока из графа все ребра исключаем). Для того чтобы их связать, необходимо найти $N-1$ ребро.

Другими словами, алгоритм организует процесс роста компонент связности, в процессе которого они объединяются друг с другом до тех пор, пока не останется одна компонента, являющаяся конечным результатом. На рис. 10.4 показаны этапы работы алгоритма Краскала.

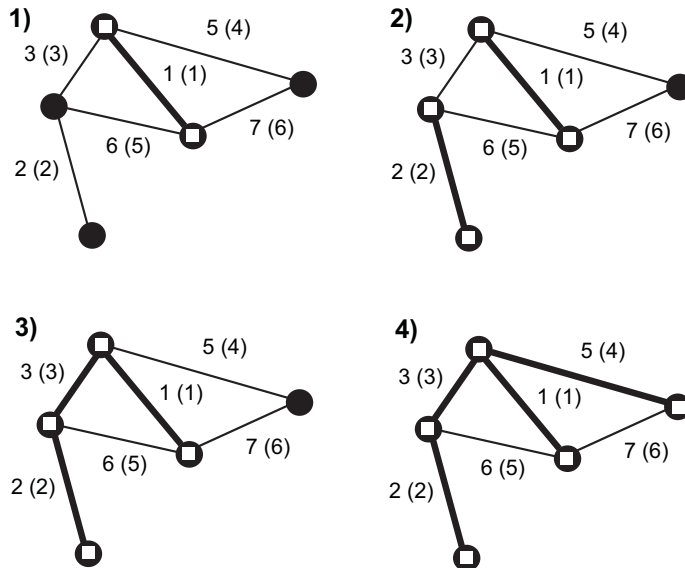


Рис. 10.4

Белыми квадратиками и жирными линиями, как и на рис. 10.3, отмечены элементы графа, уже включенные в остов. На рис. 10.4 изображений на одно меньше. Это не означает, что этапов работы алгоритма Краскала меньше, чем у Прима. В иллюстрации к алгоритму Прима первый рисунок показывает стадию выбора начальной вершины. В алгоритме Краскала выбор вершины отсутствует. Обратите внимание на отличие. В алгоритме Прима дерево остова растет, всегда оставаясь связным графом. В примере алгоритма Краскала после второй итерации появляются две компоненты, содержащие по две вершины.

Реализация. Алгоритм Краскала работает со множеством упорядоченных ребер. Это означает, что для естественной реализации алгоритма нам потребуется специальная конструкция данных, содержащая информацию о ребрах. Если считать, что вершины графа каким-то образом пронумерованы, то структуру данных ребра можно оформить так:

```
ТипРебро=RECORD
    Вершина1, Вершина2, Вес:INTEGER;
END;
```

где **Вершина1** и **Вершина2** – номера вершин, инцидентных данному ребру, а **Вес** – его вес. Определять структуру данных графа будем, как и прежде, матрицей смежности. Сейчас из матрицы смежности необходимо построить массив ребер. Это выполнит следующий программный фрагмент:

```
Nc:=-1;
FOR k:=0 TO n-1 DO
    FOR j:=k+1 TO n-1 DO
        IF Граф[k,j]#0 THEN
            Nc:=Nc+1;
            Ребро[Nc].Вершина2:=k;
            Ребро[Nc].Вершина1:=j;
            Ребро[Nc].Вес:=Граф[k,j];
        END;
    END;
END;
```

Следующее требование алгоритма Краскала – это упорядочение массива ребер в порядке возрастания. Не будем изощряться в алгоритмах сортировки, используем простейший – пузырьек. Фрагмент программы записан в следующем листинге:

```
FOR k:=0 TO Nc-2 DO
    FOR j:=0 TO Nc-k-1 DO
        IF Ребро[j].Вес>Ребро[j+1].Вес THEN
            q:=Ребро[j];
            Ребро[j]:=Ребро[j+1];
            Ребро[j+1]:=q;
        END;
```

```
END;
END;
```

И наконец, переходим к главному – к итерационному процессу построения остовного дерева. Можно строить дерево до тех пор, пока некоторый критерий не скажет, что дерево готово. Например, таким критерием может стать включение всех вершин исходного графа в остовное дерево. Если на каждой итерации в остов включается только одна вершина, то можно считать количество вершин, включаемых в остов, и как только это количество станет равно количеству вершин графа, процесс прекратить. Можно завершение процесса построить на подсчете ребер. Если проведен анализ всех ребер исходного графа, то, очевидно, остов построен. Мы остановимся на этом критерии.

```
s:=0;
WHILE s<=Nc DO
  (*Итерация*)
  s:=s+1;
END;
```

Смысл цикла понятен – выполняется проход всего массива ребер, и для каждого принимается решение, включать его в остов или нет. Из идеи алгоритма ясно, что ребро должно быть включено в остов в том случае, если оно соединяет два не связанных доселе компонента. Следовательно, необходимы две вещи:

- необходимо определить, входят ли вершины, инцидентные ребру, в один компонент или в разные;
- необходимо научиться объединять компоненты.

Определим для вершин графа специальный массив (в программе массив **Компонент[]**). Номер элемента массива – это номер вершины в графе, значение элемента – это номер компонента, которому принадлежит вершина. До начала итерационного процесса каждая вершина – это отдельный компонент, поэтому инициализируем массив вершин номерами вершин. Тогда критерий принадлежности разным компонентам будет таков: если элементы массива вершин имеют разные значения, то соответствующие им вершины, очевидно, принадлежат разным компонентам. Нетрудно построить и операцию объединения компонентов. Предположим, что ребро соединяет две вершины, одна из них имеет номер компонента $N1$, вторая – номер компонента $N2$. Тогда для объединения достаточно просмотреть массив вершин, найти в нем все элементы, имеющие значение $N2$, и присвоить им значение $N1$. Этим проблема объединения полностью решится. После чего останется переписать ребро из матрицы смежности исходного графа в матрицу смежности остова **Остов[]**. Фрагмент, полностью описывающий итерационный процесс, в следующем листинге:

```
s:=0;
WHILE s<=Nc DO
  IF Компонент[Ребро[s].Вершина1]# Компонент[Ребро[s].Вершина2] THEN
```

```

d1:= Компонент[Ребро[s].Вершина1];
d2:= Компонент[Ребро[s].Вершина2];
FOR k:=0 TO n-1 DO
  IF d[k]=d2 THEN d[k]:=d1; END;
END;
Остов[Ребро[s].Вершина2,Ребро[s].Вершина1]:=Граф[Ребро[s].Вершина2,
Ребро[s].Вершина1];
(*Учет симметричного элемента в матрице смежности*)
Остов[Ребро[s].Вершина1,Ребро[s].Вершина2]:=Граф[Ребро[s].Вершина2,
Ребро[s].Вершина1];
END;
s:=s+1;
END;

```

Ниже – полное решение задачи.

Листинг 10.2

```

MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE АлгоритмКраскала*;
TYPE
  ТипРебро=RECORD
    Вершина1, Вершина2, Вес:INTEGER;
  END;
VAR
  Граф,Остов:ARRAY 10, 10 OF INTEGER;
  Компонент:ARRAY 10 OF INTEGER;
  Ребро:ARRAY 100 OF ТипРебро;
  q:ТипРебро;
  k,j,n,Nc,s,d1,d2:INTEGER;
  flag:BOOLEAN;
BEGIN
  In.Open;
  In.Int(n);
  FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO
      In.Int(Граф[k,j]);
      Остов[k,j]:=0
    END;
    Компонент[k]:=k;
  END;
  Nc:=-1;
  FOR k:=0 TO n-1 DO
    FOR j:=k+1 TO n-1 DO
      IF Граф[k,j]#0 THEN

```

```

    Nc:=Nc+1;
    Ребро[Nc].Вершина2:=k;
    Ребро[Nc].Вершина1:=j;
    Ребро[Nc].Вес:= Граф[k,j];
  END;
END;
END;
(*Сортировка*)
FOR k:=0 TO Nc-1 DO
  FOR j:=0 TO Nc-k-1 DO
    IF Ребро[j].Вес>Ребро[j+1].Вес THEN
      q:=Ребро[j];
      Ребро[j]:=Ребро[j+1];
      Ребро[j+1]:=q;
    END;
  END;
END;
s:=0;
WHILE s<=Nc DO
  IF Компонент[Ребро[s].Вершина1]# Компонент[Ребро[s].Вершина2] THEN
    d1:= Компонент[Ребро[s].Вершина1];
    d2:= Компонент[Ребро[s].Вершина2];
    FOR k:=0 TO n-1 DO
      IF Компонент[k]=d2 THEN Компонент[k]:=d1; END;
    END;
    Остов[Ребро[s].Вершина2,Ребро[s].Вершина1]:=Граф[Ребро[s].Вершина2,
      Ребро[s].Вершина1];
    Остов[Ребро[s].Вершина1,Ребро[s].Вершина2]:= Граф[Ребро[s].Вершина2,
      Ребро[s].Вершина1];
  END;
  s:=s+1;
END;
FOR k:=0 TO n-1 DO
  FOR j:=0 TO n-1 DO
    StdLog.Int(Остов[k,j]);
  END;
  StdLog.Ln;
END;
END АлгоритмКраскала;
END Модуль.

```

Задание для самостоятельной работы

Перепишите этот алгоритм, так чтобы процесс завершался по результатам анализа вершин, включенных в остов. Это можно сделать двумя способами. Во-пер-

вых, можно считать вершины, включаемые в остов, и, во-вторых, можно проверять массив вершин на предмет того, не стал ли он одной компонентой.

Алгоритм поиска компонент связности

Формулировка задачи. Дан произвольный граф, возможно состоящий из нескольких компонент связности. И дана произвольная вершина графа. Требуется выделить компоненту связности, содержащую данную вершину.

Идея алгоритма. Предположим, что некоторое количество вершин, принадлежащих компоненте, уже известно. Присвоим этим вершинам некоторое число, будем называть его статусом и договоримся, что для вершин, принадлежащих компоненте, статус имеет вполне определённое значение. Тогда расширить множество вершин, принадлежащих компоненте можно простой процедурой:

- возьмём любую вершину, чей статус говорит о том, что вершина принадлежит компоненте связности;
- передадим её статус всем вершинам, ей смежным.

На рис. 10.5 показаны три стадии построения компоненты связности от вершины, помеченной на первой стадии единицей. На стадии 3 все вершины, принадлежащие одной компоненте, отмечены числом 2.

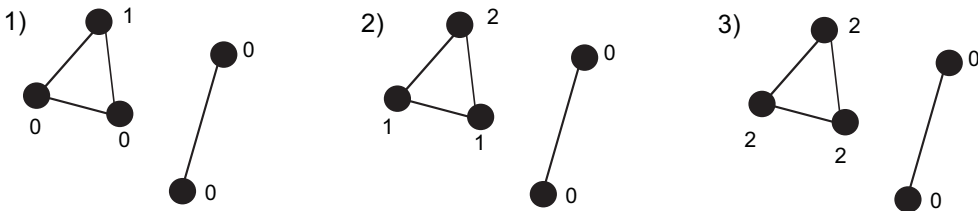


Рис. 10.5

Реализация. Решением задачи будет массив вершин графа, имеющих определенный статус. Договоримся, что этот статус равен двум. Исходные данные составят два массива: двумерный массив – матрица смежности, и одномерный массив – массив статусов вершин. Матрица смежности, естественно, вводится из потока, массив статусов инициализируем нулями. Еще одно данное – это номер вершины, для которой строится компонента связности. Её статус перед началом процесса – единица. Процесс можно оформить условным циклом и в качестве условия продолжения возьмем наличие вершин с единичным статусом. Начнем конструирование программы с этого главного цикла:

```
Статус[ИсходнаяВершина]:=1;
s:=1;
WHILE s>0 DO
  (*Построение компоненты связности*)
END;
```

Здесь **ИсходнаяВершина** – это номер вершины, относительно которой строит-

ся компонента. Величина s (счетчик вершин единичного статуса) изменяется так: при появлении вершины со статусом 1 она увеличивается на 1, а при появлении вершины со статусом 2 уменьшается на 1. Распространение единичного статуса похоже на распространение волны, которая идет от источников во все нулевые вершины. Источники волны – вершины со статусом единица. Следовательно, на каждом шаге итерации необходимо просмотреть весь граф и обнаружить все вершины-источники – вершины статуса 1.

```

Статус[ИсходнаяВершина]:=1;s:=1;
WHILE s>0 DO
  FOR k:=0 TO n-1 DO
    IF Статус[k]=1 THEN
      (*Дальнейшее распространение статуса 1*)
    END;
  END;
END;

```

Обнаружив вершину-источник, пройдем по всем её связям в матрице смежности и статус всех смежных с ней вершин, имеющих нулевой статус, изменим на единицу. Присвоение вершине статуса 1 должно сопровождаться оператором: $s:=s+1$;. После завершения обработки вершины-источника статус источника изменяется на 2. Присвоение статусу вершины статуса 2 должно сопровождаться оператором: $s:=s-1$;. Приведем полное решение.

Листинг 10.3

```

MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE ПостроениеКомпонентыСвязности*;
VAR
  Граф:ARRAY 10, 10 OF INTEGER;
  Статус:ARRAY 10 OF INTEGER;
  k,j,s,ИсходнаяВершина,n:INTEGER;
BEGIN
  In.Open;
  In.Int(n);
  FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO
      In.Int(Граф[k,j]);
    END;
    Статус[k]:=0;
  END;
  In.Int(ИсходнаяВершина);
  Статус[ИсходнаяВершина]:=1;s:=1;
  WHILE s>0 DO
    FOR k:=0 TO n-1 DO

```



```

IF Статус[k]=1 THEN
  FOR j:=0 TO n-1 DO
    IF (Граф[k,j]=1) & (Статус[j]=0) THEN
      Статус[j]:=1;s:=s+1;
    END;
  END;
  Статус[k]:=2;s:=s-1;
END;
END;
FOR k:=0 TO n-1 DO
  IF Статус[k]=2 THEN
    StdLog.Int(k);
  END
END;
END ПостроениеКомпонентыСвязности;
END Модуль.

```

Волновой алгоритм

Формулировка задачи. Дан непустой граф. Необходимо найти путь между двумя вершинами, содержащий наименьшее количество вершин (ребер).

Идея алгоритма. Перед началом работы алгоритма некоторой начальной вершине присваивается число, называемое волновой меткой и имеющее минимальное значение (в алгоритме это минимальное значение равно единице). Далее волна начинает движение по графу, увеличивая свою высоту и расставляя волновые метки в тех вершинах, в которые она приходит. Метки расставляются так: если волна в вершину B пришла из вершины A , при этом в вершине A значение волновой метки равно H , то в вершине B значение волновой метки определяется как $H+1$.

Таким образом, значения волновых меток зависят от длины пути, пройденного волной. Следовательно, можно утверждать, что чем больше значение волновой метки, тем больший путь был пройден волной до данной вершины. Заметим, что волновой алгоритм использует стратегию обхода графа в ширину. На рис. 10.6 показан пример распространения волны по графу.

Примечание. Анализируемый нами алгоритм называется волновым, и для объяснения принципа его работы используется термин «волна». Но если вы будете внимательны, то обязательно обратите внимание на универсальность некоего волнового принципа. Все изучаемые здесь алгоритмы так или иначе запускают на графах процесс волнового распространения некоей числовой величины и делают свои выводы, исходя из анализа поведения этой величины.

Реализация. Граф, естественно, представим матрицей смежности. Но в данном случае этого недостаточно. После прохождения волны граф превращается во взвешенный. При этом веса присваиваются вершинам графа, а какая-либо информация о вершинах никаким образом в матрице смежности не отображается. Это

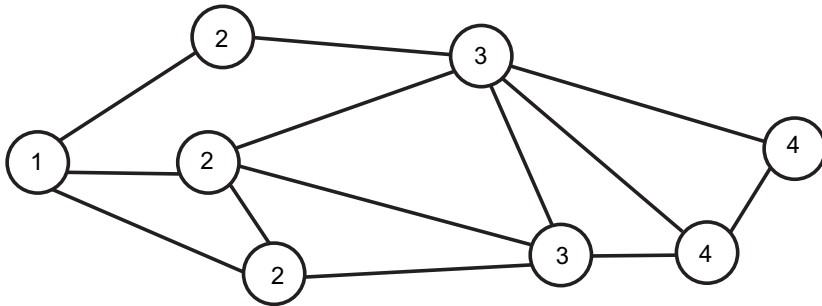


Рис. 10.6

наводит на мысль о необходимости дополнительной структуры данных, сохраняющей информацию о проходящей волне. В качестве такой структуры вполне подойдет одномерный массив. Для формирования массива вершины графа достаточно пронумеровать произвольным образом, тогда индекс элемента массива есть номер вершины графа. Договоримся также, во избежание путаницы с нумерацией, что номер вершины соответствует номеру строки в матрице смежности.

Очевидно, процесс прохождения волны можно смоделировать циклом, работающим до тех пор, пока волна не заполнит весь граф. Если создать счетчик, считающий количество «передач волны», то, видимо, условие завершения цикла будет таким:

```
s:=1;
WHILE s<n DO
  (*шаг волны*)
END;
```

Условие предполагает, что вершины нумеруются с нуля и n – их количество. Рассуждаем дальше. В некоторые из вершин графа волна пришла на предыдущем шаге, такие вершины становятся источником для дальнейшего движения, назовем такие вершины вершинами типа A . Некоторые из вершин уже были источниками – это тип B , и некоторые еще ждут волны – это вершины типа C . Вершины типа C содержат нуль (речь идет о массиве вершин, а не о матрице смежности). Вершины типа A и типа B содержат значения, отличные от нуля, а различие между ними в том, что значение вершин типа A равно текущему значению волны. Следовательно, на каждом шаге главного цикла необходимо просмотреть весь массив вершин и найти вершины, значение которых равно текущему значению волны. Цикл немного усложнится:

```
s:=1;Вершина[0]:=1;Волна:=1;
WHILE s<n DO
  FOR k:=0 TO n-1 DO
    IF Вершина[k]=Волна THEN
      (*Передача волны*)
    END;
```

```

END;
Волна:=Волна+1;
END;

```

В этом фрагменте мы учли еще некоторые важные вещи. А именно:

- в начале процесса нулевая (будем для упрощения считать, что процесс распространения начинается с нулевой вершины) вершина содержит единичное значение высоты;
- высота волны начинается с единицы и после пересчета, на следующей итерации увеличивается на единицу.

Найдя очередную вершину-источник, мы должны найти все смежные ей вершины и передать им значение высоты волны на единицу большее текущего. Это единственный момент, когда становится нужна матрица смежности. По номеру вершины определяем строку, ей соответствующую, проходим эту строку и каждый её элемент, не равный нулю, означает вершину, смежную данной. Единственное – из рассмотрения необходимо исключить вершины, до которых волна уже дошла, это вершины, уже имеющие ненулевое значение. Программный фрагмент приобретает следующий вид:

```

s:=1;Вершина[0]:=1;Волна:=1;
WHILE s<n DO
  FOR k:=0 TO n-1 DO
    IF Вершина[k]=Волна THEN
      FOR j:=k+1 TO n-1 DO
        IF (Граф[k, j]=1) & (Вершина[j]=0) THEN
          Вершина[j]:=Волна +1;
          s:=s+1;
        END;
      END;
    END;
  END;
  Волна:=Волна+1;
END;

```

И наконец, полное решение.

Листинг 10.4

```

MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE ВолновойАлгоритм*;
VAR
  Граф:ARRAY 10, 10 OF INTEGER;
  Вершина:ARRAY 10 OF INTEGER;
  k,j,n,s,Волна:INTEGER;
BEGIN

```

```

In.Open;
In.Int(n);
FOR k:=0 TO n-1 DO
  FOR j:=0 TO n-1 DO
    In.Int(Граф[k,j]);
  END;
  Вершина[k]:=0;
END;
s:=1; Вершина[0]:=1;Волна:=1;
WHILE s<n DO
  FOR k:=0 TO n-1 DO
    IF Вершина[k]=Волна THEN
      FOR j:=k+1 TO n-1 DO
        IF (Граф[k, j]=1) & (Вершина[j]=0) THEN
          Вершина[j]:=Волна+1;
          s:=s+1;
        END;
      END;
    END;
  END;
  Волна:=Волна+1;
END;
FOR k:=0 TO n-1 DO
  StdLog.Int(Вершина[k]);
END;
END ВолновойАлгоритм;
END Модуль.

```

Цикл, записанный по завершении операций обработки, распечатывает окончательное распределение высот прошедшей через граф волны (волновые метки). Это окончательное решение, но это не вполне то, что требуется. Изначально требовалось получить кратчайший путь. Поэтому сейчас рассмотрим способ восстановления пути по полученному распределению высот и матрице смежности.

Идея здесь следующая. Предположим, что до некоторой вершины (назовем её *Очередной*) путь уже построен, и необходимо найти вершину – *Продолжение*. Пусть значение волновой метки в *Очередной* вершине равно **Волна**. Тогда очевидно, что волна пришла в *Очередную* вершину из вершины со значением высоты **Волна-1**. Все, что теперь нужно, – это в соответствующей строке матрицы смежности найти все единичные элементы, а они соответствуют смежным вершинам, и найти одну содержащую волновую метку со значением **Волна-1**, это и будет вершина – *Продолжение*.

Заметим, однако, что таких *Продолжений* может быть несколько. Это естественно, так как кратчайший путь не обязан быть единственным. Задача поиска всех кратчайших путей несколько сложнее, но идея та же.

Задание для самостоятельной работы

Дополните написанную программу процедурой, восстанавливающей все кратчайшие пути, построенные волновым алгоритмом.

Задание для самостоятельной работы

На самом деле острой необходимости в дополнительном массиве для хранения значений волны нет. Для этих целей вполне можно использовать матрицу смежности. Например, если граф не содержит петель, то главная диагональ матрицы смежности содержит нули и не несет в себе никакой информации о ребрах графа, следовательно, для хранения значений волны можно использовать эти диагональные значения.

Еще один вариант. В нашей реализации прохождение волны фиксируется на вершинах, но волна идет по ребрам и, следовательно, её можно фиксировать на ребрах, а матрица смежности хранит в себе всю информацию о ребрах.

Так или иначе, но попробуйте построить реализацию волнового алгоритма без дополнительного массива.

Алгоритм Дейкстры

Формулировка задачи. Имеется взвешенный граф. Некоторая его вершина обозначена как первая. Необходимо найти минимальные пути от первой вершины до остальных вершин графа. Минимальным путём будем называть путь с минимальной суммой цен вдоль пути. Ценой назовем неотрицательное число, являющееся весом ребра.

Идея алгоритма. Цель алгоритма – определить для каждой вершины минимальные цены. Минимальная цена – это стоимость самого дешевого пути от начальной вершины до данной. Предположим, что для некоторого количества вершин такие минимальные стоимости уже определены. Это не слишком обязывающее требование. Перед началом процесса одна такая вершина есть, это начальная, её минимальная стоимость равна нулю, далее после каждой итерации множество таких вершин будет увеличиваться.

Итак, множество вершин с минимальными ценами не пусто. Попробуем в это множество добавить еще одну вершину. Все множество вершин графа на каждой итерации представимо двумя множествами: множество вершин, для которых минимальные цены известны (назовем его множество A), и множество остальных вершин. Эти два множества соединены некоторым количеством ребер, иначе говоря, между двумя множествами можно проложить некоторое количество путей, имеющих разную стоимость. Найдем минимальный путь. Такой путь в одно ребро, начинаясь в вершине с известной минимальной ценой, заканчивается в вершине с еще не определенной ценой. И эту вершину мы и включим в множество A .

Разберемся, как определяется минимальная цена для новой, включаемой вершины. Для этого введем два ценовых понятия: предварительная и окончательная цены. Алгоритм Дейкстры сначала делает некоторую прикидку и дает предварительную цену для вершины, которая затем уточняется, может быть, несколько раз и превращается в окончательную. Ниже в картинках на конкретном числовом

примере показан процесс превращения предварительной цены в окончательную, сейчас же попробуем дать словесное определение этой операции.

Пусть выполнено некоторое количество итераций. Их результат – распределение цен на вершинах графа. Для некоторых вершин уже определены окончательные цены, для некоторых – предварительные, для некоторых оценки еще не было. Предположим, результатом текущей итерации будет объявление окончательной цены для вершины A .

Пусть вершина A смежна некоторому количеству вершин, обозначим их как B_1, \dots, B_k . Как на состояние вершин B_j повлияет объявление окончательной цены для вершины A ? Здесь возможны три ситуации:

1. Для вершины B_j уже известна окончательная цена. Тогда эта цена не изменится, на то она и окончательная.
2. Вершина B_j еще не оценивалась. Тогда её предварительная цена равна сумме цены вершины A и цены ребра, их соединяющего.
3. Вершина B_j уже оценивалась и имеет цену C_1 . Обозначим сумму из предыдущего пункта как C_2 . Тогда новая предварительная цена есть минимум из цен C_1 и C_2 .

Таким образом, вычисляются предварительные цены. Появление такой цены у вершины и её изменение есть следствие объявления окончательной цены у некоторой другой вершины смежной данной. Если рассуждения о предварительных ценах вам понятны, то далее все проще. После пересчета предварительных цен найдем среди них минимальную. Она и будет очередной окончательной.

Еще раз схема расчета: некая предварительная цена становится окончательной, она изменяет множество предварительных цен для тех вершин, с которыми она связана ребром. В новом множестве предварительных цен есть наименьшая или даже несколько наименьших, выбираем любую из них, объявляем окончательной, и процесс повторяется до тех пор, пока все предварительные цены не станут окончательными. А начинается все с начальной вершины, для которой сразу объявляется окончательная цена, равная нулю.

Пример построения. Еще один вариант описания алгоритма – описание в картинках. На рис. 10.7 изображен граф. В кружочках, изображающих вершины, представлены номера вершин около ребер веса. Неподчернутые числа около вершин – это предварительные цены, подчеркнутые числа – это окончательные цены.

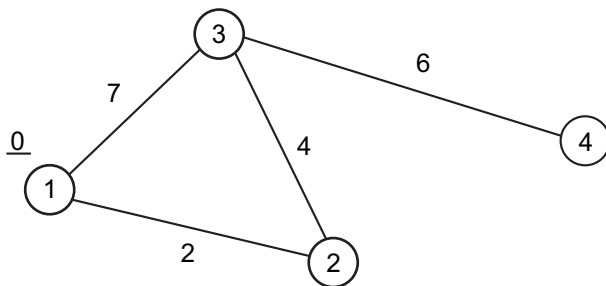


Рис. 10.7

Предположим, что построение пути начинается с первой вершины. Если мы уже здесь, то очевидно, что для неё известна окончательная цена, и она равна нулю. Эта окончательная цена даст две предварительные, так как первая вершина смежна со второй и третьей вершинами.

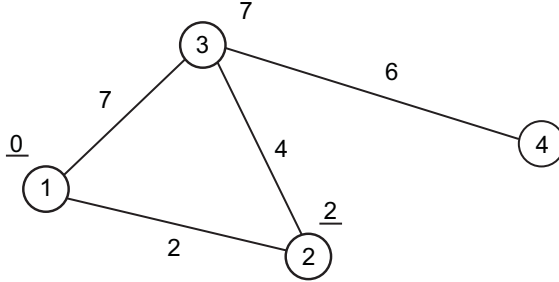


Рис. 10.8

Здесь имеем две предварительные цены. Меньшая цена – 2, и её объявим окончательной. Эта окончательная цена изменит предварительную цену в вершине с номером 3. Оказывается, из 1-го в 3-ий окружной путь дешевле. Получаем следующую картину:

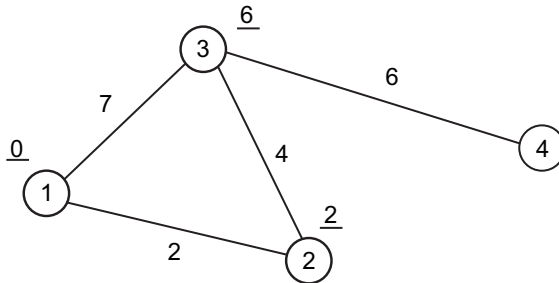


Рис. 10.9

Полученная предварительная цена 6 сразу объявлена окончательной, так как множество предварительных цен состоит только из неё одной. И остается последний шаг – вычисление предварительной цены, которая тут же становится окончательной для четвертой вершины:

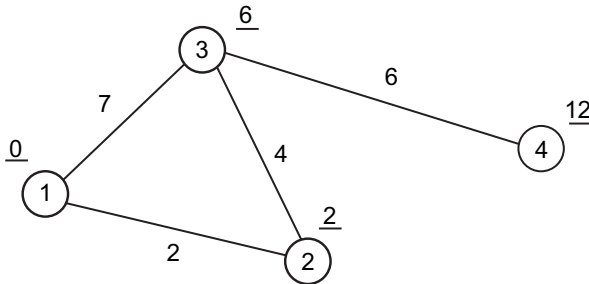


Рис. 10.10

В этом примере, несмотря на его простоту, показана важнейшая особенность алгоритма – предварительные цены могут изменяться. Как и сколько раз, зависит от сложности графа и распределения весов ребер.

Реализация. Каждый шаг итерационного процесса добавляет еще одну окончательную цену и, соответственно, еще одну вершину выводит из вычислительного процесса. Следовательно, итерационный процесс можно представить уже привычным нам циклом:

```
s:=1;  
WHILE s<n DO  
  (*Итерация*)  
  s:=s+1;  
END;
```

Цикл отсчитывает количество итераций, равное количеству вершин за минусом первой. Прежде чем двигаться дальше, обсудим необходимые структуры данных. Одна структура очевидна, это матрица смежности, которой мы всегда пользуемся. Дополнительно необходимо определиться со структурой данных, описывающей вершину. Полное описание включает предварительную и окончательную цены, поэтому для описания множества вершин определим массив записей:

```
Вершина: ARRAY 10 OF RECORD;  
          c1,c2:INTEGER;  
END;
```

Перед началом главного цикла итераций описанная структура данных должна быть инициализирована нулями для каждой вершины графа.

Будем считать, что построение дешевых путей всегда строится с нулевой вершины. Это ни в коей мере не уменьшает универсальности алгоритма, так как простой перенумерацией любую вершину можно сделать нулевой. Еще один технический нюанс нашего решения – это единичное значение окончательной цены исходной вершины. Это сделано для того, чтобы иметь возможность отличать еще не рассмотренную вершину от вершины, для которой окончательная цена не известна (в этом случае цена нулевая). Наше решение предполагает, что окончательная цена – это обязательно ненулевая цена. Впрочем, указанный нюанс также никак не ограничивает общности решения. Просто по завершении итераций можно от всех полученных цен отнять единицу.

Все технические особенности обсуждены, можно продолжить построение главного процесса. На каждой итерации есть ровно одна вершина, способная изменить ситуацию с предварительными ценами, назовем её вершина **Источник**. Перед началом процесса **Источник** = 0 в каждой итерации этот номер вычисляется заново, как номер вершины, для которой будет объявлена окончательная цена. Вершина **Источник** смежна с некоторым количеством вершин, среди которых вполне могут оказаться вершины с неопределенной окончательной ценой. Поэтому необходимо просмотреть соответствующую строку матрицы смежности и найти все такие вершины:


```

WHILE s<n DO
  FOR k:=0 TO n-1 DO
    IF Граф[Источник,k]>0 THEN
      (*Расчеты*)
    END;
  END;
  s:=s+1;
END;

```

Вершина с номером **k** – это вершина, смежная вершине с номером **Источник**. Далее необходимо перевычислить предварительную цену вершины **k**. Здесь возможны два варианта. Во-первых, может оказаться, что предварительная цена вершины (компонент **c1**) нулевая. В этом случае её предварительная цена, очевидно, равна сумме окончательной цены вершины **Источник** и цены ребра, их соединяющего. Если предварительная цена больше упомянутой суммы, то её значение также меняется на значение суммы. Если же предварительная цена вершины **k** ненулевая, но меньше упомянутой суммы, то она (предварительная цена) не изменяется. Все сказанное следующим образом изменяет главный цикл:

```

s:=1; Источник:=0;
Вершина[Источник].c2:=1; Вершина[Источник].c1:=1;
WHILE s<n DO
  FOR k:=0 TO n-1 DO
    IF Граф[Источник,k]>0 THEN
      IF Вершина[k].c2=0 THEN
        IF (Вершина[k].c1=0) OR
           Вершина[k].c1>Вершина[Источник].c1+Граф[Источник,k]) THEN
          Вершина[k].c1:=Вершина[Источник].c1+Граф[Источник,k];
        END;
      END;
    END;
  END;
  s:=s+1;
END;
(*Поиск минимальной цены и новой вершины - Источник *)

```

В этом листинге мы еще добавили начальные условия. После отработки цикла **FOR** полностью построено новое множество предварительных цен, и очередная задача – найти среди них минимальное и определить новое значение – **Источник**. В предыдущем листинге отмечено место, где это должно быть посчитано, переписывать главный цикл мы больше не будем. Для начала поиска минимального значения необходимо взять какое-то число, которое или обязательно присутствует в анализируемом массиве, или это стороннее число, но тогда оно должно быть не меньше любого из тех, которые могут встретиться. Мы построим такое стороннее число как сумму всех цен ребер графа. Очевидно, предварительная цена, большая

такого числа, никогда не появится. Заметим также, в анализе участвуют только те вершины, для которых окончательная цена равна нулю, а предварительная больше нуля. С учетом сказанного фрагмент выглядит следующим образом:

```

min:=max;
FOR k:=0 TO n-1 DO
  IF (Вершина[k].c2=0) & (Вершина[k].c1>0) THEN
    IF Вершина[k].c1<min THEN
      min:= Вершина[k].c1; Источник:=k;
    END;
  END;
END;
Вершина[Источник].c2:= Вершина[Источник].c1;

```

А ниже – листинг полного решения.

Листинг 10.5

```

MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE АлгоритмДейкстры*;
VAR
  Граф:ARRAY 10, 10 OF INTEGER;
  Вершина:ARRAY 10 OF RECORD;
    c1,c2:INTEGER;
  END;
  n,k,j,Источник,s,min,max:INTEGER;
BEGIN
  In.Open;
  In.Int(n);max:=0;
  FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO
      In.Int(Граф[k,j]);max:=max+ Граф[k,j];
    END;
    Вершина[k].c1:=0; Вершина[k].c2:=0;
  END;
  s:=1; Источник:=0;
  Вершина[Источник].c2:=1;
  Вершина[Источник].c1:=1;
  WHILE s<n DO
    FOR k:=0 TO n-1 DO
      IF Граф[Источник,k]>0 THEN
        IF Вершина[k].c2=0 THEN
          IF (Вершина[k].c1=0) OR
            (Вершина[k].c1> Вершина[Источник].c1+Граф[Источник,k]) THEN
            Вершина[k].c1:= Вершина[Источник].c1+ Граф[Источник,k];

```

```
    END;
  END;
END;
END;
min:=max;
FOR k:=0 TO n-1 DO
  IF (Вершина[k].c2=0) & (Вершина[k].c1>0) THEN
    IF Вершина[k].c1<min THEN
      min:= Вершина[k].c1; Источник:=k;
    END;
  END;
END;
Вершина[Источник].c2:= Вершина[Источник].c1;
s:=s+1;
END;
FOR k:=0 TO n-1 DO
  StdLog.Int(Вершина[k].c2-1);
END;
END АлгоритмДейкстры;
END Модуль.
```

Задание для самостоятельной работы

Построенная программа вычисляет цены путей от нулевой вершины ко всем остальным, но она не отвечает на вопрос, как получить путь от исходной вершины до какой-либо конкретной. Алгоритм восстановления пути основан на следующем факте: предположим, путь некоторой длины уже восстановлен, и «путник» в текущий момент находится в вершине A . Цена этой вершины известна, обозначим её как C_A . Тогда среди смежных с ней вершин B (обозначим их все одной буквой) есть хотя бы одна такая, что $C_A = C_B + D_{AB}$, где D_{AB} – стоимость ребра между вершинами A и B . Напишите процедуру, восстанавливающую путь от заданной вершины до исходной, используя указанный факт.

Задание для самостоятельной работы

Ясно, что кратчайших путей может быть несколько. Напишите процедуру, восстанавливающую все возможные пути.

Задание для самостоятельной работы

Граф может быть достаточно большим. Тогда его обработка потребует значительных ресурсов, и станет актуальным вопрос экономии счетного времени. В нашем же алгоритме анализ графа фактически выполняется дважды. Однако второй проход для восстановления пути совсем не является необходимым. Действительно, в момент переопределения цены вершины точно известно, откуда пришла новая цена. Владея такой информацией, самый дешевый путь можно строить в процессе первого обхода. Модифицируйте реализацию с учетом этого требования.

Алгоритм Флойда

Формулировка задачи. Дан взвешенный граф с положительными весами ребер. Требуется найти кратчайшие длины путей между всеми парами вершин графа.

Идея алгоритма. Если в алгоритме Дейкстры волна цен идет от одной вершины, считающейся началом пути, то в алгоритме Флойда «волна запускается от каждой вершины». Фраза взята в кавычки, так как аналогия с алгоритмом Дейкстры здесь возможна только с некоторой натяжкой, кое-что для понимания алгоритма будет добавлено в пункте обоснования, и ниже выполнен достаточно содержательный расчетный пример, детальный анализ которого даст, наверное, исчерпывающее понимание техники работы.

Инициализация структур данных:

Построим матрицу D^0 размерности $|V| \times |V|$, элементы которой (v – исходный граф, V – количество вершин) определяются по правилу:

1. $d_{ii}^0 = 0$;
2. $d_{ij}^0 = \text{цена}(v_i, v_j)$, где $i \neq j$, если в графе существует ребро (дуга) (v_i, v_j) ;
3. $d_{ij}^0 = \text{бесконечность}$, где $i \neq j$, если нет ребра (дуги) (v_i, v_j) .

Основная часть алгоритма:

Для m от 0 до V выполнить действия.

- Строится матрица с индексом, равным номеру шага, обозначим его через m , в которой элементы определяются через элементы матрицы предыдущего шага по следующим формулам: $d_{ij}^{m+1} = \min\{d_{ij}^m, d_{i(m+1)}^m + d_{(m+1)j}^m\}$, где $i \neq j$; $d_{ii}^{m+1} = 0$.

По завершении работы данного алгоритма элементы матрицы равны длинам кратчайших путей между соответствующими вершинами.

Пример расчета

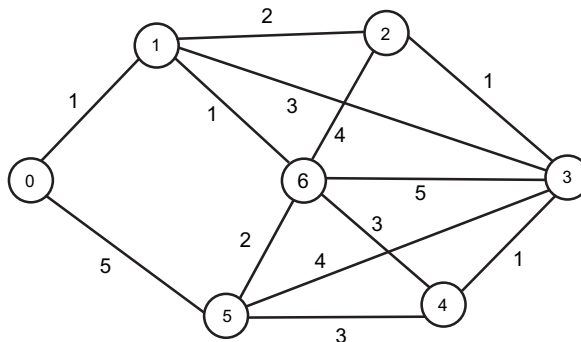


Рис. 10.11

Исходная матрица.

Таблица 10.1

	0	1	2	3	4	5	6
0	0	1	∞	∞	∞	5	∞
1	1	0	2	3	∞	∞	1
2	∞	2	0	1	∞	∞	4
3	∞	3	1	0	1	4	5
4	∞	∞	∞	1	0	3	3
5	5	∞	∞	4	3	0	2
6	∞	1	4	5	3	2	0

Матрица после первой итерации.

Таблица 10.2

	0	1	2	3	4	5	6
0	0	1	3	4	∞	5	2
1	1	0	2	3	∞	∞	1
2	3	2	0	1	∞	∞	3
3	4	3	1	0	1	4	4
4	∞	∞	∞	1	0	3	3
5	5	∞	∞	4	3	0	2
6	2	1	3	4	3	2	0

Матрица-результат.

Таблица 10.3

	0	1	2	3	4	5	6
0	0	1	3	4	5	5	2
1	1	0	2	3	4	7	1
2	3	2	0	1	2	5	3
3	4	3	1	0	1	4	4
4	5	4	2	1	0	3	3
5	5	7	5	4	3	0	2
6	2	1	3	4	3	2	0

Задание для самостоятельной работы

В примере показан результат первой итерации и последней. Но их не две, их больше. Восстановите матрицы для всех промежуточных итераций.

Обоснование алгоритма. Исходная структура содержит цены (веса) для ребер, непосредственно соединяющих две вершины. Иначе можно сказать, что исходная матрица содержит цены уже построенных путей, состоящих из одного ребра. Бесконечные значения появляются в силу того, что есть пары вершин, между которыми нет пути длиной в одно ребро.

Следующая итерация строит пути длиной в два ребра, следующая – длиной в три ребра и т. д. Выбор минимального значения из двух необходим для того, чтобы определить, какой путь имеет более низкую стоимость, – тот, который уже построен, или новый, более длинный (на одно ребро).

Количество итераций равно количеству вершин – 1. Это следует из очевидного факта, максимально длинный путь без циклов (а только такой имеет смысл) может содержать каждую вершину только один раз, а количество ребер, составляющих такой максимальный путь, равно количеству вершин – 1.

Если все цены неотрицательны, то алгоритм Флойда можно рассматривать как более сильный вариант алгоритма Дейкстры. Однако если появляются отрицательные величины, то в случае появления цикла с отрицательной стоимостью алгоритм Флойда уже не справится со своей задачей, но в качестве компенсации он позволяет обнаружить такой плохой цикл. Но опять же напомним, мы немного упростили постановку задачи. При положительных ценах отрицательные циклы не появятся.

Реализация. Матрица для алгоритма Флойда имеет одну важную особенность. В ней в некоторых ячейках стоят бесконечности. Ни один язык программирования не умеет работать с бесконечностью. Впрочем, в этом нет необходимости, наши бесконечные числа бесконечны в том смысле, что все остальные значения матрицы меньше. Поэтому можно в качестве бесконечных значений взять сумму всех элементов матрицы. Больше этой суммы ни один элемент матрицы, естественно, не может быть. Основной итерационный процесс состоит из точно известного количества итераций, поэтому организуем его уже привычным образом:

```
s:=0;
WHILE s<n-2 DO
  (*Выполнение итерации*)
  s:=s+1;
END;
```

Каждый шаг итерации заключается в обходе матрицы и пересчете её элементов согласно формуле. Формула такова, что для корректного её выполнения необходима вспомогательная матрица. Поэтому в процедуре созданы два массива: **Граф1[]** и **Граф2[]**, все расчеты выполняются с массивом **Граф1[]**, результат присваивается элементам массива **Граф2[]**, после чего указатели **Граф1** и **Граф2** переопределяются. Обход матрицы **Граф1[]** выполняется так:

```
s:=0;
WHILE s<n-2 DO
  FOR k:=0 TO n-1 DO
    FOR j:=k+1 TO n-1 DO
      (*Пересчет элементов матрицы*)
    END;
  END;
  c:=Граф1;Граф1:=Граф2;Граф2:=c;
```

```
s:=s+1;
END;
```

Пересчет элементов выполняется по формуле, реализованной оператором выбора:

```
IF Граф1[k,j]<Граф1[k,s+1]+Граф1[s+1,j] THEN
  Граф2[k,j]:=Граф1[k,j];
ELSE
  Граф2[k,j]:=Граф1[k,s+1]+Граф1[s+1,j];
END;
Граф2[j,k]:= Граф2[k,j];
```

Последний оператор присваивания необходим для учета симметрии матрицы смежности. Переписывать еще раз цикл итераций уже не будем, запишем сразу полное решение.

Листинг 10.6

```
MODULE Модуль;
IMPORT In, StdLog;
PROCEDURE АлгоритмФлойда*;
TYPE
  ТипГраф=ARRAY 10,10 OF INTEGER;
VAR
  Граф1,Граф2,c:POINTER TO ТипГраф;
  n,j,k,s,max:INTEGER;
BEGIN
  In.Open;
  In.Int(n);max:=0;
  NEW(Граф1); NEW(Граф2);
  FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO
      In.Int(Граф1[k,j]);max:=max+Граф1[k,j];
    END;
  END;
  FOR k:=0 TO n-1 DO
    FOR j:=k+1 TO n-1 DO
      IF Граф1[k,j]=0 THEN Граф1[k,j]:=max; Граф1[j,k]:=70;END;
    END;
  END;
  s:=0;
  WHILE s<n-2 DO
    FOR k:=0 TO n-1 DO
      FOR j:=k+1 TO n-1 DO
        IF Граф1[k,j]< Граф1[k,s+1]+ Граф1[s+1,j] THEN
          Граф2[k,j]:= Граф1[k,j];
```

```

ELSE
  Граф2[k,j]:= Граф1[k,s+1]+Граф1[s+1,j];
END;
Граф2[j,k]:= Граф2[k,j];
END;
END;
c:=Граф1; Граф1:=Граф2; Граф2:=c;
s:=s+1;
END;
FOR k:=0 TO n-1 DO
  FOR j:=0 TO n-1 DO
    StdLog.Int(Граф1[k,j]);
  END;
  StdLog.Ln;
END;
END Алгоритм Флойда;
END Модуль.

```

Задание для самостоятельной работы

Как и алгоритм Дейкстры, алгоритм Флойда только считает цены, полученная матрица смежности не содержит информации о кратчайших путях. Попробуйте реализовать необходимые дополнения самостоятельно. Для восстановления путей перед началом работы алгоритма можно построить матрицу P с начальными значениями элементов $p_{ij} = i$. Каждый раз, когда на очередном шаге итерации значение d_{ij}^{m+1} уменьшается (то есть когда $d_{i(m+1)}^m + d_{(m+1)j}^m < d_{ij}^m$), выполняется присваивание $p_{ij} := p_{(m+1)j}$. В конце работы алгоритма матрица P будет определять кратчайшие пути между всеми парами вершин: а именно значение p_{ij} равно номеру предпоследней вершины в пути между i и j (либо $p_{ij} = i$, если путь не существует).

Нахождение максимального потока

Существует ряд задач, в которых требуется через конструкцию, представимую графом, пропустить поток какого-то вещества или предметов, в общем чего-то материального. Задачей такого рода можно считать задачу проектирования сети городских дорог (если, конечно, такую задачу в действительности кто-нибудь решает), расчет сети трубопроводов и ряд других. Общее в них то, что проектировщика интересует не просто принципиальная возможность пропустить некоторое количество воды, нефти, газа или автомобильного транспорта через проектируемую сеть, а пропустить их как можно больше. В этом, по сути, и заключается задача нахождения максимального потока.

Для того чтобы точно сформулировать задачу, разберемся с некоторыми понятиями. Поток назовем набор дополнительных числовых значений, поставленных в соответствии дугам графа. Речь о дугах, так как задача нахождения максимального потока имеет смысл для ориентированного графа. Числовые значения,

описывающие поток, конечно же, не любые. Ниже на рис. 10.12 две ситуации. Ситуация (а) описывает некоторый поток. Ситуация (б) некорректна.

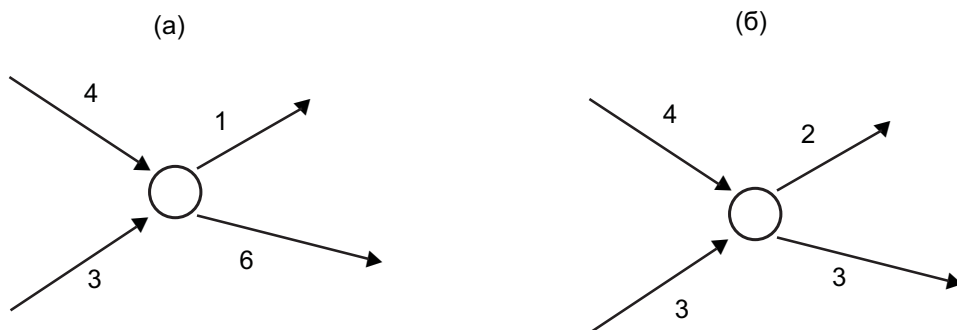


Рис. 10.12

На каждом из рисунков мы видим одну вершину с двумя входящими и двумя исходящими дугами. На рисунке (а) входящий поток равен 7, и этому же значению равен исходящий поток. Такая вершина называется сбалансированной, какой поток в неё поступает, такой поток и уходит. Вершина на рисунке (б) не сбалансированна. Входящее значение потока = 7, а выходит поток = 5. Имеет смысл рассматривать только такие потоки, для которых все вершины графа являются сбалансированными.

Взвешенный граф для нас уже не новость, но для потоковых задач ситуация немного усложняется. Здесь придется дугам сопоставить два числовых значения. Одно значение – это поток, проходящий через дугу, и второе число характеризует пропускную способность дуги. Поток есть величина изменяемая. Пропускная способность – это строго определенная характеристика дуги, ограничивающая максимально возможное значение потока.

Еще два важных потоковых понятия – это исток и сток. Истоком называется вершина графа, через которую внешний поток поступает в граф. Стоком называется вершина, через которую поток уходит из графа. В общем случае необходимо рассматривать граф с несколькими истоками и несколькими стоками. Но принципиально множественность истоков и стоков в задачу ничего не вносит, поэтому есть смысл рассматривать для большей прозрачности рассуждений графы с одним истоком и одним стоком. А сейчас дадим общую формулировку задачи.

Дан граф с одним истоком и одним стоком. Необходимо найти максимальный поток, который можно пропустить через граф от истока до стока.

Договоримся еще об одной условности. Так как пропускные способности есть характеристика дуги, а не вершины, то пусть исток имеет одну специальную дугу, соединяющую граф с внешним миром, но не имеющую пропускной способности, и соответственно, похожая дуга для стока. Такие немного виртуальные дуги, не ограничивающие специально потока, но удобные, чтобы видеть, откуда поток приходит и куда он уходит. Ниже на рис. 10.13 пример графа с определенным максимальным потоком.

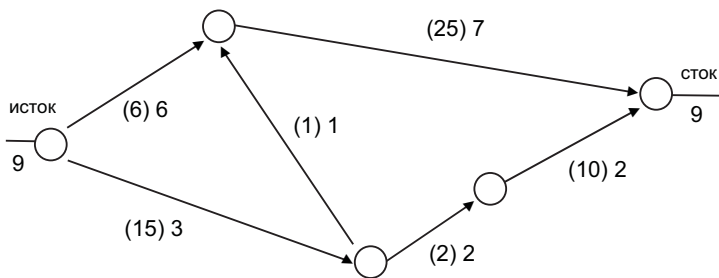


Рис. 10.13

Здесь в скобочках обозначены пропускные способности, значения потока – без скобочек. Все вершины графа сбалансированны. Входящий поток 9 единиц, выходящий также 9 единиц. Заметим, что не все дуги графа использованы эффективно, по некоторым можно было бы пустить больший поток, но тогда некоторые вершины графа оказались бы несбалансированными. На данном графе мы имеем одно возможное решение. Но в общем случае максимальный поток, не меняя своего общего значения, может по-разному распределяться между дугами графа. На рис. 10.14 пример.

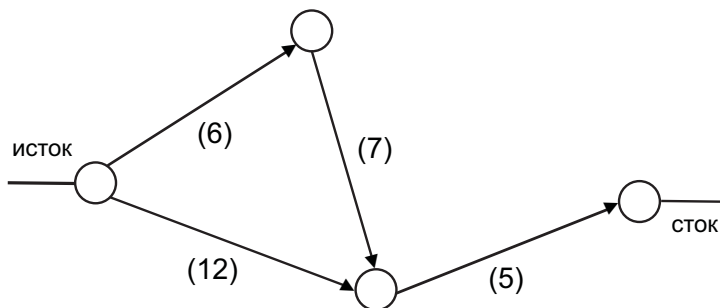


Рис. 10.14.

Проблема – в дуге, входящей в сток. Она единственная, и её пропускная способность очень низка. В вершину, из которой эта дуга выходит, может поступить поток в 19 единиц, а пройти через дугу может только 5 единиц. Значит, 5 единиц – это максимум потока, и этот максимум можно распределить между остальными дугами графа несколькими способами.

Рассмотрим принципиальную возможность построения максимального потока. Для этого заметим, что ограничителем потока является требование баланса входящих и выходящих потоков для каждой вершины графа. Отсюда следует идея. Предположим, что через граф уже идет некоторый поток. Это утверждение всегда справедливо. Как минимум нулевой поток есть всегда. Введем понятие остаточной пропускной способности вершины. Это число, равное величине потока, который можно пропустить через данную вершину дополнительно к потоку, уже через неё проходящему.

Предположим, что через некоторую вершину проходит нулевой поток, тогда остаточная пропускная способность равна минимуму из двух сумм: пропускных

способностей входящих дуг и суммы пропускных способностей исходящих дуг. Необходимость счета именно минимума очевидна. Максимальный исходящий поток равен сумме пропускных способностей исходящих дуг, но мы не можем завести в вершину поток больший, чем сумма пропускных способностей входящих дуг. Если же через вершину уже идет ненулевой поток, то его необходимо вычесть из остаточной пропускной способности вершины при нулевом потоке.

Пусть далее некоторый поток по графу уже идет. Если при этом есть возможность пустить дополнительный поток, то, следовательно, есть вершины с ненулевой остаточной пропускной способностью. Найдем такую вершину. Пусть через неё возможно пропустить дополнительный поток величины L . Это означает, что поток величины L необходимо подвести к вершине и поток величины L необходимо от вершины отвести. Рассмотрим вторую задачу.

Как отвести поток величины L от вершины. Поток от вершины можно отвести только по исходящим дугам. Ясно, что поток величины L можно (возможно, несколькими способами) распределить между исходящими дугами. Выполним каким-нибудь образом такое распределение. Для каждой из исходящих дуг возможны две ситуации:

- доля потока, пришедшегося на дугу, не превосходит остаточной пропускной способности этой дуги. Это хороший случай, оставляем все как есть;
- доля потока превосходит остаточную пропускную способность дуги. В этом случае уменьшаем долю и соответственно уменьшаем добавочный поток через вершину.

Рассмотрев, таким образом, все исходящие дуги, мы согласуем доли добавочного потока с реальными возможностями этих дуг. Аналогичное согласование необходимо провести для входящих в вершину дуг.

Изложенная идея грешит двумя серьёзными недостатками. Во-первых, нет никаких указаний на порядок обхода вершин графа, во-вторых, совершенно не ясно, как распределять поток между исходящими дугами. И самое главное – метод основан на пересчете локального баланса в окрестности одной вершины, гарантирует ли это установление баланса в масштабах всего графа – вопрос открытый. Впрочем, интуитивно ясно, что какое-то приближение к максимальному потоку мы получим. Но главное не это. В описанном методе содержится важная идея построения максимального потока. Сформулируем её четче. Метод построения максимального потока может представлять собой последовательность итераций, каждая из которых пытается построить некоторый добавочный поток, исходя из неизрасходованных ресурсов дуг. Именно так работает метод Форда-Фалкерсона.

Алгоритм Форда-Фалкерсона

Если возможен дополнительный поток, то, значит, существует путь, по которому поток может пройти. Речь идет о последовательности дуг от истока до стока, каждая из которых может пропустить какой-то дополнительный поток. Построение дополнительного потока заключается в организации перебора возможных путей и вычислении для найденного пути добавочного потока. Полный поток через

граф считается максимально возможным, если нет пути, по которому возможен дополнительный поток. Сейчас мы должны ответить на два вопроса:

- как построить путь по графу;
- как рассчитать дополнительный поток по найденному пути.

Предположим, что дополнительный путь уже построен. Он состоит из некоторого количества дуг. Через каждую дугу можно пропустить некоторый дополнительный поток (к тому, который уже проходит через дугу). Выберем из этих локальных потоков минимальный, он и будет искомым дополнительным потоком для всего пути. Остается учесть ориентированность графа. Путь по графу ориентирован от истока графа до стока. А так как выбор пути произволен, то вполне может оказаться, что ориентация некоторых дуг противоположна ориентации пути. Поэтому для:

- дуг, ориентированных вдоль пути, минимальный поток прибавляется;
- дуг, ориентированных против пути, минимальный поток вычитается.

Несколько следующих рисунков иллюстрируют простой процесс построения потоков. На первом из них (рис. 10.15) показано распределение пропускных способностей. Жирными линиями выделен возможный путь. Минимальный поток, который можно пропустить по дугам этого пути, равен 2. Это значение и становится значением добавочного потока.

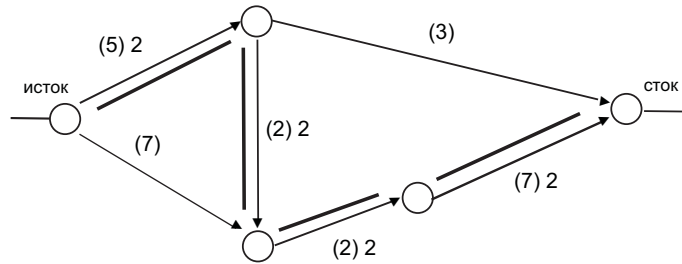


Рис. 10.15

Минимальный поток следующего пути также равен 2. Поэтому к существующему потоку опять надо добавить значение 2. Но одна дуга ориентирована против пути. Для такой дуги должно выполнить вычитание добавочного. В результате произойдет перераспределение потоков между дугами (рис. 10.16).

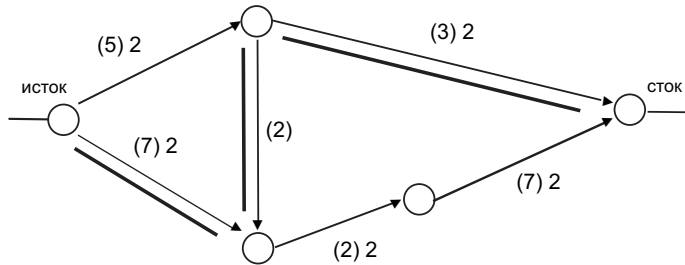


Рис. 10.16.

Выполним еще один шаг (рис. 10.17):

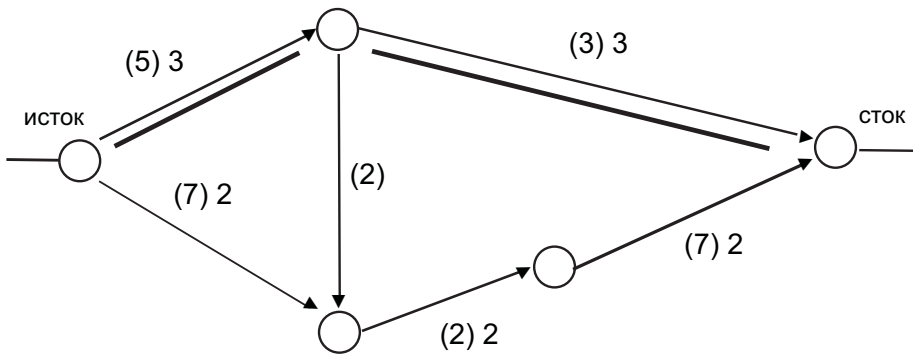


Рис. 10.17

Этот шаг для графа последний. Максимально возможный поток из истока в сток 5 единиц. Визуально видно, что больше нельзя. Но, к сожалению, компьютер не умеет принимать решения визуально, поэтому займемся разработкой критерия завершения процесса и выработаем процедуру построения всех возможных увеличивающих поток путей (далее будем говорить просто «увеличивающий путь»).

Критерий завершения процесса таков: максимальный поток получен, если невозможно построить увеличивающий путь. Это утверждает теорема Форда-Фалкерсона, которую мы примем без доказательств. А вот проблема построения очередного увеличивающего пути решается не совсем просто.

Построение увеличивающего пути поиском в глубину

Пусть в обходе в глубину мы достигли некоторой вершины. Эта вершина инцидентна некоторому количеству дуг. Рассмотрим их все, за исключением, разумеется, той, по которой была достигнута данная. Кандидатами на продолжение пути из них выбираются только те, через которые можно пропустить часть дополнительного потока, дошедшего до данной вершины. Дуги могут быть двух типов:

- ориентированные вдоль пути. Такая дуга является кандидатом, если её остаточная пропускная способность больше нуля;
- ориентированные против пути. Такая дуга является кандидатом, если по ней уже идет ненулевой поток (есть что уменьшить).

По выбранной дуге пропускается дополнительный поток, равный:

- если дуга ориентирована вдоль пути, то меньшей величине из двух: величине потока, дошедшего до рассматриваемой вершины, и величине остаточной пропускной способности. Это поток со знаком $+$, он добавляется к уже прошедшему по дуге;
- если дуга ориентирована против пути, то меньшей величине из двух: потоку, дошедшему до рассматриваемой вершины, и потоку, уже идущему по дуге. Это поток со знаком $-$, он вычитается из уже идущего по дуге.

Увеличивающий путь считается построенным, если достигнут сток графа. Немного ниже мы рассмотрим еще один метод построения увеличивающего пути, именуемый методом расстановки меток. А сейчас важное замечание об эффективности метода Форда-Фалкерсона. Заметим, что увеличивающий путь может строиться вдоль дуг, ориентированных против направления пути. Это означает, что поток на отдельных участках может попеременно увеличиваться и уменьшаться. А если учесть, что величина дополнительного потока на каждом шаге итерации также может уменьшаться, то вопрос сходимости итерационного процесса оказывается открытым. И действительно, при иррациональных значениях пропускных способностей существуют примеры несходящихся процессов. При целочисленных значениях решение будет получено гарантированно, но при большом разбросе значений пропускных способностей количество итераций может оказаться очень велико даже на простом графе. Рассмотрим следующий, уже ставший классическим пример:

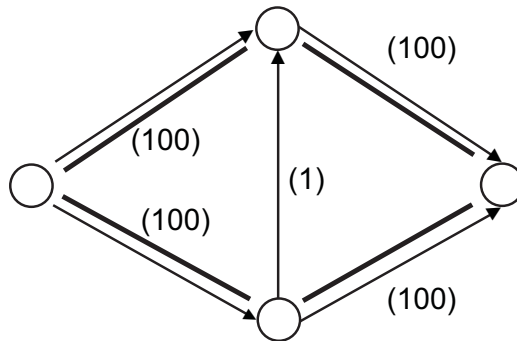


Рис. 10.18

На рис. 10.18 показаны два увеличивающих пути, вдоль которых может пойти максимально возможный на этом графе поток. Но метод Форда-Фалкерсона не описывает точной процедуры выбора оптимальных путей, следовательно, итерации могут быть иными. Рассмотрим два шага иного решения.

Шаг первый:

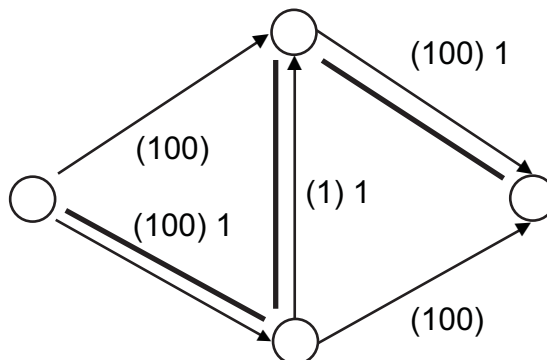


Рис. 10.19

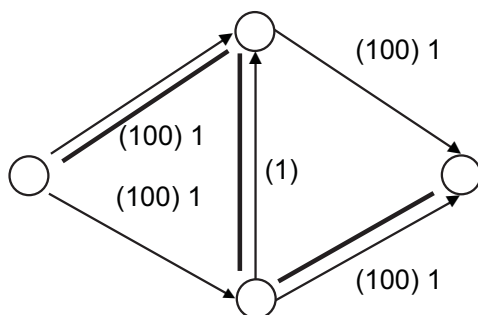
Шаг второй:

Рис. 10.20

Как видите, за два шага мы мало что получили. И до полного решения, которое можно получить за две итерации, еще очень и очень далеко. Ниже рассмотрим второй подход поиска увеличивающего пути, а сейчас последний вопрос – вопрос завершения процесса. Уже было сказано, что процесс считается завершенным, если невозможно построить увеличивающий путь. Что означает эта невозможность?

Поиск в глубину требует движения по дугам графа вперед до тех пор, пока не встретился тупик. Если, пройдя из вершины A в вершину B , проходчик графа зафиксировал тупик, то он возвращается в вершину A . Возможно, из вершины A есть другие пути, в этом случае возобновляется движение вперед, если других путей нет, то возврат продолжается, и если обходчик вынужденно вернется в вершину, с которой обход был начат, и обходчик зафиксирует состояние тупика в этой исходной вершине, это и будет означать невозможность построения увеличивающего пути.

Задание для самостоятельной работы

Попробуйте построить пример графа, для которого процесс построения увеличивающих путей по алгоритму Форда-Фалкерсона не будет сходиться.

Метод расстановки меток

Этот метод несколько похож на волновой алгоритм. От истока по графу до стока запускается числовая волна, если она дойдет до стока, то увеличивающий путь получен, иначе итерационный процесс завершен. Значение числовой волны – это дополнительный поток, который можно пропустить через дугу, инцидентную данной вершине.

Разобьем все вершины графа на два класса: помеченные и не, помеченные. Помеченные вершины – это вершины, для которых определен поток, к ним подходящий. Непомеченные – это соответственно вершины с неопределенным потоком. Помеченные вершины также разобьем на два класса: рассмотренные и нерассмотренные. Рассмотренная вершина – это вершина, сыгравшая роль источника волны, нерассмотренная соответственно – нет. Договоримся также, что пометка состоит из трех компонентов: величины дополнительного потока, номера вершины, из которой пришел поток, и флага, информирующего об ориентации дуги, из которой

пришел поток (поток может быть положительным и отрицательным).

Процесс заключается в распространении меток. На нулевой итерации вершина исток считается помеченной и нерассмотренной. Вершины, помеченные на очередной итерации, передаются на следующую, как нерассмотренные. Далее, каждая нерассмотренная вершина рассматривается на предмет передачи метки вершинам смежным с данной. По завершении анализа нерассмотренной вершины она объявляется рассмотренной. Обозначим помеченную, нерассмотренную вершину как A .

Пометка вершины заключается в следующих действиях:

- записываем в пометку анализируемой вершины номер вершины A ;
- если анализируемая вершина соединена с A дугой, ориентированной вдоль пути, то флаг пометки $+$, иначе флаг пометки $-$;
- записываем в пометку меньшую из двух величин: величину потока, дошедшего до вершины A , и величины остаточной пропускной способности дуги от A до анализируемой вершины.

Итерация, на которой вершина сток оказалась помечена, является последней. После чего можно восстановить увеличивающий путь от стока до истока. Исследуем процедуру восстановления. Но для начала – пример, из которого будет видно, как восстановить путь:

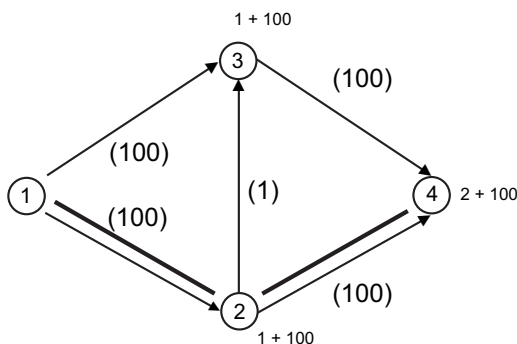


Рис. 10.21.

Путь восстанавливается естественным образом, так как в метке для каждой из вершин явным образом указывается, откуда пришел поток. Следующий вопрос – завершение процесса. Максимальный поток получен, если не удалось построить увеличивающего пути. Поэтому сейчас, как и для метода поиска в глубину, посмотрим, что означает фраза «нельзя построить увеличивающий путь».

Волна меток, начинаясь от истока, должна завершить свой путь в стоке. На каждом шаге волна смещает свой фронт в новые вершины. Для того чтобы фронт волны переместился из, например, вершины A в вершину B , необходимо, чтобы дуга, соединяющая эти вершины, имела ненулевой поток, если она ориентирована из B в A , или имела остаточную пропускную способность, если она ориентирована из A в B . Если окажется, что ни для одной дуги ни одной вершины фронта это условие не выполняется, то волна не сместится на новые вершины и это будет означать завершение процесса построения максимального потока.

Реализация алгоритма. Наша реализация использует обход графа в глубину. Начнем строить реализацию с главного процесса, который по уже установившейся традиции опишем циклом по условию. Главный процесс состоит из трех действий:

1. Построение пути от начальной вершины к конечной.
2. Определение минимального потока, который по этому пути можно пропустить.
3. Добавление вычисленного значения минимального потока.

Очевидно, что ключевое действие здесь – это построение пути. Если путь возможен, то и остальные действия могут быть выполнены, если путь невозможен, то работа программы должна быть завершена. Следующая конструкция выполняет описанную работу:

```
WHILE ПоискПути() DO
    (*Расчет потока вдоль найденного пути*)
    (*Добавление вычисленного потока*)
END;
```

Детальное построение функции **ПоискПути()** можно отложить на потом, но что является результатом её действия и что она должна получить на входе, обсудим сейчас. Так как внутри цикла выполняются операции над найденным добавочным путем, то, видимо, функция должна возвращать **Истину**, если таковой путь был найден, и **Ложь** при неудаче поиска. Ясно также, что функция не просто выясняет возможность или невозможность, пусть она строит путь, но так как возвращаем мы одно логическое значение, то разумно путь описать глобальной структурой данных, которая в теле функции и будет определяться. В качестве такой структуры определим массив для записи номеров вершин, вдоль которых проходит путь. Найденный путь может иметь различную длину, поэтому в теле функции вычисляется еще одна глобальная величина – длина пути. Итак, пусть на очередном шаге вычислен путь как последовательность вершин. Займемся расчетом добавочного потока.

Идея здесь такова: необходимо пройти по всем дугам пути и найти, так сказать, самую узкую дугу. Поток, который можно через неё пропустить, и есть максимально возможный. Здесь есть два технических нюанса. Во-первых, есть дуги, ориентированные вдоль пути, и дуги, ориентированные против пути. И во-вторых, необходимо где-то взять первое минимальное значение. Ясно, что в качестве первого минимального значения можно взять добавочный поток вдоль первой дуги найденного пути, но это дополнительные операторы в теле главного цикла. Мы используем прием, уже встречавшийся ранее. Определим величину, большую любого могущего встретиться минимального потока, и ее возьмем в качестве первого. Такая большая величина может быть, например, суммой пропускных способностей дуг.

Первая проблема немного сложнее. Дуги с разной ориентацией относительно пути необходимо по-разному обрабатывать. Для дуг, ориентированных вдоль пути, с текущим минимумом необходимо сравнивать остаток пропускной способности

(разность веса дуги с уже идущим по дуге потоком), а для дуг, ориентированных против пути, с текущим минимумом будем сравнивать уже идущий поток (так как для противоположноориентированных дуг поток вычитается). Для того чтобы понять следующий программный фрагмент, договоримся о структурах данных:

- **Граф[]** – квадратная матрица смежности, содержащая пропускные способности дуг;
- **Поток[]** – квадратная матрица смежности, содержащая потоки, проходящие по дугам;
- **Вершина[]** – одномерный массив вершин найденного пути.

Таким образом, запись **Граф[Вершина[k],Вершина[k+1]]** означает пропускную способность между вершинами, имеющими номера **Вершина[k]** и **Вершина[k+1]**, а запись **Поток[Вершина[k],Вершина[k+1]]** означает поток, проходящий между этими вершинами.

```

min:=max;
FOR k:=0 TO ДлинаПути-1 DO
  IF (Граф[Вершина[k],Вершина[k+1]]>0) &
    (Граф[Вершина[k],Вершина[k+1]]-
      Поток[Вершина[k],Вершина[k+1]]<min)
  THEN
    (*ориентирована вдоль*)
    min:=Граф[Вершина[k],Вершина[k+1]]-
      Поток[Вершина[k],Вершина[k+1]];
  ELSIF (Граф[Вершина[k+1],Вершина[k]]>0) &
    (Поток[Вершина[k+1],Вершина[k]]< min)
  THEN
    (*ориентирована против*)
    min:=Поток[Вершина[k+1],Вершина[k]];
  END;
END;
```

Следующая операция – это добавление вычисленного потока. Вычисленное значение находится в переменной **min**. Для каждой дуги увеличивающего пути, ориентированной вдоль пути, значение **min** необходимо добавить к уже существующему потоку, а для дуг, ориентированных против пути, значение **min** необходимо отнять от уже существующего потока. Существование дуги между вершинами проверять нет необходимости, если дуга включена в путь, значит, вопрос о её существовании уже решен. Для существующей дуги есть только два варианта ориентации, поэтому достаточно проверять, ориентирована ли дуга вдоль, если нет, то, очевидно, она ориентирована против. Дуга с вершинами **Вершина[k]** и **Вершина[k+1]** (вершины соседствующие на найденном пути) ориентирована вдоль пути, если **Граф[Вершина[k],Вершина[k+1]]>0**.

```

FOR k:=0 TO ДлинаПути-1 DO
  IF Граф[Вершина[k],Вершина[k+1]]>0 THEN
```

```

(*Ориентирована вдоль пути*)
Поток[Вершина[k],Вершина[k+1]]:=
Поток[Вершина[k],Вершина[k+1]]+min;
ELSE
(*Ориентирована против пути*)
Поток[Вершина[k+1],Вершина[k]]:=
Поток[Вершина[k+1],Вершина[k]]- min;
END;
END;

```

Записанный выше фрагмент заполняет матрицу потока, которая изначально инициализирована нулями. Значения в этой матрице, как видно, могут не только увеличиваться, но и уменьшаться. Заметим также, что матрица пропускных способностей (весов дуг) никак не изменяется. Этим фрагментом мы завершаем рассмотрение операций, выполняемых над найденным увеличивающим путем. И сейчас самое время рассмотреть функцию **ПоискПути()**.

Начнем рассуждения с некоторых общих идей. Пусть функция **ПоискПути()** строит путь, добавляя к уже построенному по очереди все вершины, смежные с последней вершиной пути. Очевидно, что такую функцию удобно реализовать как рекурсивную. Начинает свою работу функция с вершины с номером ноль и заканчивает, либо когда путь продолжить не удастся, либо когда достигнута вершина с номером $n-1$ (последняя). Напомним, что для простоты понимания мы зафиксировали исток и сток графа. Очевидно, что обязательным аргументом функции должен быть номер очередной вершины. Общая структура тела функции может быть такова:

```

Вершина[L]:=num;
IF num=n-1 THEN
  ДлинаПути:=L;
  RETURN TRUE;
ELSE
  (*Попытка продолжить путь*)
END;

```

num – это передаваемый аргумент, номер вершины в которую путь достраивается на текущей активации **ПоискПути()**. Эта вершина записывается в глобальный массив вершин пути. Для выполнения записи необходимо отслеживать текущую длину массива, для чего длину можно сделать еще одним аргументом функции. На короткое время вернемся к главному циклу и перепишем его с учетом полученной информации:

```

WHILE ПоискПути(0,0) DO
  (*Расчет потока вдоль найденного пути*)
  (*Добавление вычисленного потока*)
END;

```

Аргументы функции означают, что построение пути всегда начинается с нулевой вершины и нулевого порядкового номера вершины на пути. Продолжим построение функции **ПоискПути()**. Её главный каркас – это условный оператор. Его смысл в том, что если достигнут сток, то процесс рекурсивных активаций необходимо прекратить и начать сворачивание рекурсии, иначе необходимо осуществить попытку продвинуться еще на одну вершину. Кандидатов на смежность с текущей вершиной **num** ровно $n-1$, включая и саму вершину **num**. Поэтому попытка продолжить путь реализуется следующим циклом:

```
k:=0;
WHILE k<n DO
  (*Какая-то проверка*)
  k:=k+1;
END;
RETURN FALSE;
```

Выход из цикла означает неудачу со всеми вероятными кандидатами на продолжение пути, поэтому по выходе из цикла записан возврат ложного значения. Выбор правильного кандидата выполняется по следующим критериям: во-первых, дуга, соединяющая вершину **num** с вершиной-кандидатом, должна иметь ненулевую остаточную пропускную способность, если она ориентирована вдоль пути, и ненулевой поток, если она ориентирована против пути. Заметим также, что дуги, ориентированные вдоль пути и против пути, в матрице смежности обозначены симметричными элементами. Поэтому если **Граф[num, k]** – дуга, ориентированная вдоль пути, то **Граф[k, num]** – дуга, ориентированная против пути. Условие отсева дуг (вершин) запишется так:

```
k:=0;
WHILE k<n DO
  IF (Граф[num,k]-Поток[num,k]>0) OR (Поток[k,num]>0) THEN
    (*Операции по поиску пути*)
  END;
  k:=k+1;
END;
RETURN FALSE;
```

Но это еще не все. Если дуга и, соответственно, вершина удовлетворяют записанному критерию, надо убедиться, что путь еще не проходил через эту дугу (вершину). Для этого необходимо просмотреть все вершины, уже встроенные в путь, и убедиться, что анализируемой вершины (с номером **k**) там нет. Фрагмент, дополненный необходимым для этого кода, будет выглядеть так:

```
k:=0;
WHILE k<n DO
  IF (Граф[num,k]-Поток[num,k]>0) OR (Поток[k,num]>0) THEN
    flag:=TRUE;
```

```

FOR j:=0 TO L DO
  IF Вершина[j]=k THEN flag:=FALSE END;
END;
END;
k:=k+1;
END;
RETURN FALSE;

```

Если по выходе из цикла **FOR** величина **flag** есть истина, то это будет означать, что вершина с номером **k** действительно может рассматриваться как кандидат на продолжение пути и, следовательно, возможно выполнить следующий рекурсивный вызов функции:

```

k:=0;
WHILE k<n DO
  IF (Граф[num,k]-Поток[num,k]>0) OR (Поток[k,num]>0) THEN
    flag:=TRUE;
    FOR j:=0 TO L DO
      IF Вершина[j]=k THEN flag:=FALSE END;
    END;
    IF flag & ПоискПути(k,L+1) THEN RETURN TRUE END;
  END;
  k:=k+1;
END;
RETURN FALSE;

```

Важный технический нюанс. Если **ПоискПути**(k,L+1) вернет ложное значение, то это означает неудачу продолжения пути с текущей вершиной (с номером **k**), но это не означает окончательной неудачи, процесс перебора и анализа вершин может быть продолжен. Но если **ПоискПути**(k,L+1) вернет значение истина, это будет означать, что искомый увеличивающий путь обнаружен, и необходимо начать сворачивание рекурсивных активаций. Именно в этом смысл оператора

```
IF flag & ПоискПути(k,L+1) THEN RETURN TRUE END;
```

Построение функции **ПоискПути**() завершено, операции обработки увеличивающего пути также полностью разобраны, поэтому осталось привести полный листинг реализации.

Листинг 10.7

```

MODULE Модуль;
  IMPORT StdLog,In;
  PROCEDURE АлгоритмФалкерсона*;
  VAR
    Граф, Поток:ARRAY 10, 10 OF INTEGER;
    Вершина:ARRAY 10 OF INTEGER;
    n,j,k,ДлинаПути,min,max:INTEGER;

```

```

PROCEDURE ПоискПути(num,L:INTEGER):BOOLEAN;
VAR
  k:INTEGER;
  flag:BOOLEAN;
BEGIN
  Вершина[L]:=num;
  IF num=n-1 THEN
    ДлинаПути:=L;
    RETURN TRUE;
  ELSE
    k:=0;
    WHILE k<n DO
      IF (Граф[num,k]-Поток[num,k]>0) OR (Поток[k,num]>0) THEN
        flag:=TRUE;
        FOR j:=0 TO L DO
          IF Вершина[j]=k THEN flag:=FALSE END;
        END;
        IF flag & ПоискПути(k,L+1) THEN RETURN TRUE END;
      END;
      k:=k+1;
    END;
    RETURN FALSE;
  END;
END ПоискПути;
BEGIN
  In.Open;
  In.Int(n);
  max:=0;
  FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO
      In.Int(Граф[k,j]);Поток[k,j]:=0;
      max:=max+Граф[k,j];
    END;
  END;
  WHILE ПоискПути(0,0) DO
    (*Обработка добавочного потока*)
    FOR k:=0 TO ДлинаПути DO
      StdLog.Int(Вершина[k]);
    END;
    StdLog.Ln;
    min:=max;
    FOR k:=0 TO ДлинаПути-1 DO
      IF (Граф[Вершина[k],Вершина[k+1]]>0) &
        (Граф[Вершина[k],Вершина[k+1]]-

```

```

    Поток[Вершина[k],Вершина[k+1]]<min)
THEN
    (*Ориентирована вдоль пути*)
    min:=Граф[Вершина[k],Вершина[k+1]]-
        Поток[Вершина[k],Вершина[k+1]];
ELSIF (Граф[Вершина[k+1],Вершина[k]]>0) &
    (Поток[Вершина[k+1],Вершина[k]]< min)
THEN
    (*Ориентирована против пути*)
    min:=Поток[Вершина[k+1],Вершина[k]];
END;
END;
FOR k:=0 TO ДлинаПути-1 DO
    IF Граф[Вершина[k],Вершина[k+1]]>0 THEN
        (*Ориентирована вдоль пути*)
        Поток[Вершина[k],Вершина[k+1]]:=
            Поток[Вершина[k],Вершина[k+1]]+min;
    ELSE
        (*Ориентирована против пути*)
        Поток[Вершина[k+1],Вершина[k]]:=
            Поток[Вершина[k+1],Вершина[k]]- min;
    END;
END;
END;
FOR k:=0 TO n-1 DO
    FOR j:=0 TO n-1 DO
        StdLog.Int(Поток[k,j]);
    END;
    StdLog.Ln;
END;
END АлгоритмФалкерсона;
END Модуль.

```

Задание для самостоятельной работы

Второй вариант реализации – обход графа в ширину, или, иначе, метод расстановки меток, – реализуйте самостоятельно. Если возникнут технические проблемы с волной меток, проанализируйте волновой алгоритм, он по своей сути выполняет ту же самую работу.

В заключение. Алгоритмы, работающие с графами, опираются на довольно развитую математическую теорию, именуемую теорией графов. В приложении к нашей книге дана только некоторая часть основных определений. Для того чтобы приобрести систематические знания по теории графов и хорошо понять технологию алгоритмизации задач на графах, нужна специальная литература, например [16].

Приложения

Приложение 1.	
Элементы комбинаторики	297
Приложение 2.	
Теория графов	301
Приложение 3.	
Элементы теории	
вероятности	309
Приложение 4.	
Синтаксис языка	
Компонентный Паскаль	315

Приложение 1.

Элементы комбинаторики

Предмет комбинаторики

Комбинаторика изучает проблемы, связанные с конструированием некоторых комбинаций из элементов множества любой природы. Существует достаточно много задач, в которых требуется найти комбинацию элементов, обладающую некоторыми заданными свойствами. При этом возникает масса вопросов. Например, сколько таких комбинаций можно построить, как организовать процесс такого построения? Что произойдет, если потребовать для комбинации определенный порядок или, наоборот, порядок игнорировать?

Очень многое зависит от типа конструируемой комбинации, поэтому цель комбинаторики – выделить типы комбинаций и изучить их свойства. Таких базовых комбинаций можно определить несколько: перестановки, сочетания с повторением и без и размещения с повторением и без. Алгоритмические проблемы их построения достаточно подробно рассмотрены в главе 7. Цель приложения – дать определения комбинациям и указать некоторые важные свойства, например их количество.

Перестановки

Перестановка – это комбинация, в которой участвуют все элементы исходного множества. Элемент перестановки – это элемент исходного множества, занимающий определенное место. Две перестановки считаются различными, если они отличаются положением двух элементов (если есть отличие в одном элементе, то, очевидно, есть отличие еще в одном). Если исходное множество содержит N элементов, то возможно построение $N!$ перестановок.

Доказательство. На множестве из 1 элемента можно построить 1 перестановку, это очевидно. Допустим, что для подмножества в M элементов посчитано количество перестановок, и оно равно L . Возьмем еще один элемент. Его можно добавить к каждой уже полученной перестановке из M элементов $M+1$ способом. А так как число перестановок длины M равно L , то, очевидно, перестановок длины $M+1$ будет $L*(M+1)$. Очевидно, что с введением следующего элемента получим $L*(M+1)*(M+2)$ перестановок и т. д. Тогда удлинение до исходного множества в N элементов даст выражение

$$L*(M+1)*(M+2)*...*(N).$$

Если принять $M=1$, то $L=1$ (это наше первое утверждение), и тогда формула принимает следующий вид:

$$1*2*3*...*N = N!$$

Что и требовалось доказать. Завершим примером. Пусть исходное множество состоит из трех элементов: a, b, c . На этом множестве можно построить следующие перестановки: (a, b, c) , (a, c, b) , (b, a, c) , (b, c, a) , (c, a, b) , (c, b, a) ; всего $3!=6$ перестановок.

Сочетания без повторений

Сочетание – это комбинация, в построении которой участвует часть элементов исходного множества. Если, например, говорят о сочетании из трех по два, то это означает, что исходное множество содержит три элемента, а в построении сочетания участвуют только два. Приведем пример всех сочетаний из трех по два. Пусть исходное множество состоит из следующих элементов: $\{a, b, c\}$. Тогда возможны следующие сочетания без повторений: (a, b) , (a, c) , (b, c) . В сочетании порядок элементов не играет роли. Поэтому сочетание (a, b) и сочетание (b, a) – на самом деле одно и то же сочетание.

Как и в случае перестановок, интересен вопрос о количестве сочетаний. Введем обозначение: количество сочетаний из n по k обозначают так C_n^k . Формула для расчета обозначенной величины следующая:

$$C_n^k = \frac{n!}{k!(n-k)!}.$$

Проверим формулу на количество сочетаний из трех по два. Мы уже знаем, что таких комбинаций возможно только три. Формула дает:

$$C_3^2 = \frac{3!}{2!(3-2)!} = \frac{6}{2} = 3.$$

Если интересно рассчитать количество всех возможных сочетаний, то необходимо найти сумму:

$$\sum_{k=0}^n C_n^k.$$

Рассчитаем все возможные сочетания из 3 элементов:

$$\sum_{k=0}^3 C_3^k = C_3^0 + C_3^1 + C_3^2 + C_3^3 = \frac{3!}{0!*3!} + \frac{3!}{1!*2!} + \frac{3!}{2!*1!} + \frac{3!}{3!*0!} = 1 + 3 + 3 + 1 = 8.$$

Заметим, что $0!=1$. Число 8 является степенью двойки. И оказывается, что для нахождения полного количества сочетаний нет необходимости проводить такие длинные расчеты. Количество всех возможных сочетаний равно 2^n , если n – количество элементов исходного множества.

Количество сочетаний связано еще с двумя интересными алгебраическими

объектами. Первый из них – бином Ньютона. Если разложить по степеням выражение $(a+b)^n$, то получим:

$$(a+b)^n = \sum_{k=0}^n C_n^k a^{n-k} b^k,$$

где величины количеств сочетаний без повторений играют роль биномиальных коэффициентов. Это, кстати, говорит об очень большой условности в разбиении математики по областям знаний. В математике такая ситуация, когда одна и та же величина играет самые разные роли, достаточно обычна. Еще один интересный объект, в котором проявляются количества сочетаний без повторений, – это так называемый треугольник Паскаля – математическая конструкция, выглядящая следующим образом:

Таблица 11.1

0									1									1
1								1		1								2
2							1		2		1							4
3						1		3		3		1						8
4					1		4		6		4		1					16
5				1		5		10		10		5		1				32
6			1		6		15		20		15		6		1			64
7		1		7		21		35		35		21		7		1		128
8	1		8		28		56		70		56		28		8		1	256

Величины, стоящие в треугольнике, – это C_n^k , где n – это номер строки, а k – номер позиции в строке. Например, $C_7^3 = 35$. Сверимся с формулой:

$$C_7^3 = \frac{7!}{3! \cdot 4!} = 35.$$

Формула и треугольник дали один результат. Конечно, так и должно быть. Кроме того, треугольник еще раз подтверждает наше убеждение в том, что общее количество сочетаний без повторений равно степени двойки. Столбец справа в таблице содержит суммы элементов треугольника по горизонтали. Как видно, все они есть степени двойки.

Сочетания с повторениями

Отличие от сочетаний без повторений находится в точном соответствии с названием. При построении сочетания с повторением использованный элемент исходного множества не отбрасывается и может продолжать участвовать в сочетании. Для примера построим все сочетания с повторением из трех по два на множестве $\{a, b, c\}$. Получим следующие сочетания: (a, a) , (a, b) , (a, c) , (b, b) , (b, c) , (c, c) . Порядок элементов, так же как и в сочетании без повторений, не учитывается. Ниже – формула для количества. Обозначим количество таких сочетаний через S . Тогда:

$$S_N^k = C_{n+k-1}^k = \frac{(N+k-1)!}{k!(N-1)!}.$$

Для приведенного примера $S_3^2 = C_4^2 = \frac{4!}{2!2!} = 6$.

Размещения без повторений

Размещение – это уже принципиально иная комбинация. Здесь важна позиция элемента. Поэтому размещение (a, b) и (b, a) – это уже разные размещения. В этом отличие от сочетаний.

Размещения можно получать как все возможные перестановки от сочетаний, откуда понятно, что количество размещений без повторений есть произведение количества сочетаний без повторения на количество перестановок. Обозначим количество размещений через S . Тогда:

$$S_N^k = C_N^k k!.$$

Размещения с повторениями

Размещения с повторениями отличаются от размещений без повторений тем же, что и соответствующие сочетания. При построении размещений с повторениями элемент, уже принявший участие в построении размещения, не отбрасывается и может быть включен в размещение сколько угодно раз

И для размещений с повторениями формула количества максимально проста:

$$S_N^k = N^k.$$

Приложение 2.

Теория графов

Предмет теории графов

Существует большой класс задач, в котором существенную роль играют не только числовые значения данных, но и их структура. Рассмотрим для примера классическую задачу о коммивояжере. Дано несколько городов, соединенных некоторым количеством дорог известной длины. Коммивояжер, начиная свой путь из одного из городов, должен пройти их все при минимально возможной длине общего пути (рис. 11.1).

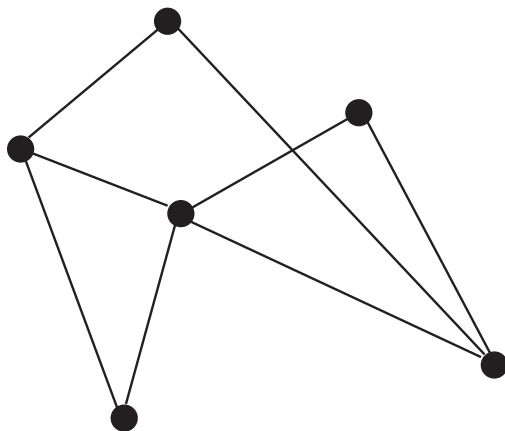


Рис. 11.1

Конечно, каждый из городов имеет название, численность населения, какую-то промышленность, но для данной задачи все это не имеет никакого значения. Для нашего коммивояжера, чем бы он ни занимался, в рамках задачи интересует только факт существования города и факт соединения двух городов дорогой. Это дает возможность очистить изучаемую структуру данных от лишней информации и выделить то единственное, что действительно важно. Существенным здесь же будет следующее: структура состоит из вершин (будем так называть города), соединенных ребрами (так будем называть дороги). Называется такая структура графом, исследование её свойств и является предметом теории графов.

Разумеется, применение теории графов не ограничивается задачей коммивояжера. Существует достаточно много задач, сводимых к структуре, состоящей из двух компонентов: вершин и ребер, их соединяющих. Иногда структуру графа можно увидеть и там, где вроде бы нет никаких физических путей. Для примера рассмотрим задачу о разбиении кучи камней на две. В этой задаче из исходной кучи камней с известными весами необходимо сделать две, такие что их веса отличаются минимальным значением. Назовем вершинами конкретные разбиения ис-

ходной кучи на две. Будем считать, что разбиение A (вершина A) связано с разбиением B (вершиной B), если разбиение B можно получить из разбиения A , поменяв местоположение одного камня (переложить его из одной кучи в другую). Таким образом, все множество вариантов разбиений укладывается в некий граф.

Ниже мы дадим некоторое количество строгих определений для основных понятий теории графов, а сейчас – небольшой нестрогий обзор темы.

Если по любому ребру графа можно двигаться в обе стороны, то такой граф называется неориентированным, но для ребер можно задать ориентацию, то есть определить одностороннее движение вдоль ребер, которые в этом случае превращаются в дуги.

Иногда о ребрах графа известно несколько больше, чем факт связи двух вершин. Каждому ребру можно присвоить число, называемое весом, в этом случае граф становится взвешенным. В задаче о коммивояжере вес ребер – это длина дорог.

Граф может иметь петли – это ребра, начинающие в некоторой вершине и в ней же заканчивающиеся.

Иногда по ребрам графа можно, выйдя из вершины, пройти по кругу и в эту же вершину вернуться. Если таких путей в графе нет, то он называется деревом.

Графическое представление графа удобно для восприятия, но неудобно для обработки, поэтому были придуманы методы представления графа числовыми матрицами. Они неудобны для восприятия, но удобны для обработки, так как легко моделируются числовыми массивами.

Граф можно разбивать на подграфы, обладающие различными свойствами.

Вершины графа можно нумеровать, удачная нумерация дает некоторые дополнительные возможности для исследования графа, но и представляет собой специальную проблему.

Основные определения

Неориентированным графом, или просто графом, $G = (X, U)$ называется упорядоченная пара (X, U) , где X – непустое множество вершин графа, а U – множество неупорядоченных пар элементов из X , называемых ребрами графа.

Ориентированным графом (орграфом) $G = (X, Y)$ называется упорядоченная пара (X, Y) , где X – непустое множество вершин орграфа (ориентированного графа), а Y – множество упорядоченных пар элементов из X , называемых дугами орграфа.

Очень много задач теории графов исследуют именно возможность построения пути из одного пункта в другой, поэтому дадим ещё несколько важных определений, что такое путь на графе, какие виды путей существуют.

Пусть u – дуга орграфа вида (x, y) , $u = (x, y)$, вершина x называется началом дуги u , а вершина y – концом дуги u . При этом говорят, что дуга выходит из x и входит в y . Если дуга или входит в вершину, или выходит из неё, говорят, что она инцидентна вершине. Дуга, у которой начало и конец находятся в одной и той же вершине, называется петлёй. Вершины, соединённые дугой (ребром), называются смежными.

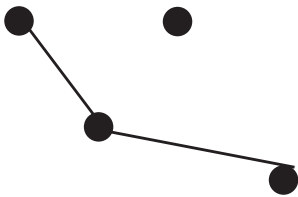
Путем из вершины a в вершину b называется последовательность вершин и дуг вида $a(a, x_1), x_1(x_1, x_2) \dots x_{n-1}(x_{n-1}, b)b$. Путь называется простым, если ни одна вершина в нём не встречается дважды.

Если в орграфе две вершины (a, b) связаны путём $u(a, b)$, то говорят, что b достижима из вершины a . Орграф называется односторонне связанным, если одна вершина достижима из другой. Орграф называется сильно связанным, если для любой пары вершин каждая из них достижима из другой.

Путь называется эйлеровым, если он содержит все дуги графа ровно по одному разу, и он называется гамильтоновым, если он содержит все вершины ровно по одному разу.

Примечание. Обратите внимание на понятие связности. Это одно из важнейших понятий. Оно говорит о том, насколько граф представляет собой единое целое. Это понятие не характеризует количество ребер, через которые можно дойти до вершины, только сам факт, можно дойти или нельзя. Ниже – примеры связанных и несвязанных графов. Первые два примера – это неориентированный связанный и несвязный граф (рис. 11.2).

Несвязный граф



Связный граф

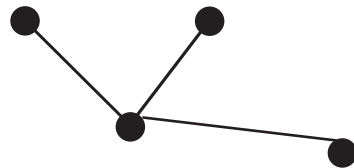
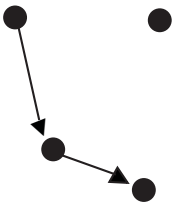


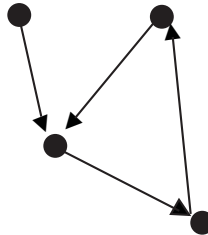
Рис. 11.2

А теперь примеры на связность ориентированного графа (рис. 11.3):

Несвязный граф



Одностороннесвязный



Сильносвязный

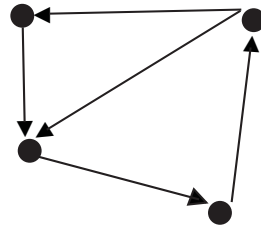


Рис. 11.3

Выше уже было сказано, что многие задачи теории связаны с возможностью обхода графа или поиском пути из одной вершины в другую. Поэтому дадим ещё несколько определений, описывающих особые вершины графа или их особые свойства.

Степенью вершины называется общее число дуг, инцидентных вершине. Число дуг, входящих в вершину, называется полустепенью входа, и число дуг, исходящих из вершины, называется полустепенью исхода. Заметим, что для неориентированных графов эти две полустепени равны, для ориентированных равенство не обязательно.

Предшественником вершины называется вершина, соединённая с данной вершиной дугой, по которой из предшественника можно пройти до данной вершины. Данная вершина по отношению к предшественнику называется потомком.

Входом, или начальной вершиной, графа называется вершина, у которой полустепень входа равна нулю. Входов может быть несколько.

Выходом, или конечной вершиной, графа называется вершина, у которой полустепень выхода равна нулю. Выходов, как и входов, может быть несколько.

Примечание. Понятно, что три последних определения действуют только для ориентированных графов.

Контур. Во многих графах можно построить путь, который будет заканчиваться в той же вершине, в какой он и начинается. Такой путь называется контуром.

Для ориентированного графа может так получиться, что две вершины соединены последовательностью ребер и все же одна из них не достижима из другой. Такие вершины есть на одностороннесвязном графе – рис. 11.3. Но тем не менее в этом примере все вершины соединены между собой, ни одну из них нельзя убрать из графа, не разрывая дуг. Для анализа таких ситуаций вводятся ещё два понятия:

Цепь. Цепью называется последовательность ребер (p_1, p_2, \dots, p_n) неориентированного графа следующего вида $p_i = (v_i, v_{i+1})$, $i = 1, 2, \dots, n$. Вершины цепи могут иметь степень, равную 1. Вершина со степенью 1 называется концевой. Цепь называется составной, если в ней повторяется хотя бы одно ребро, сложной – если повторяется хотя бы одна вершина, и простой – в противном случае.

Циклом называется цепь, у которой начальная и конечная вершины совпадают.

Примечание. Цикл и цепь, контур и путь используются для определения понятия связности, но первые два используются для неориентированных графов, а вторые два понятия – для ориентированных.

Контур простой и эйлеровый. Путь, начало и конец которого совпадают, называется контуром. Контур называется простым, если ни одна вершина в нём не повторяется дважды, эйлеровым – если он содержит все дуги графа в точности по одному разу, гамильтоновым – если он содержит все вершины графа в точности по одному разу. Длиной пути или контура называется число дуг, входящих в него.

Дерево. Существует ещё один особый вид графов – это деревья. Деревом называется связный граф с одним входом, в котором нет ни одного контура. На рис. 11.4 приведен пример двоичного дерева. Конечно, дерево, как и любой другой граф, может быть ориентированным и неориентированным. Нетрудно заметить, что ориентированное дерево не может обладать сильной связностью.

Взвешенный граф. Вернёмся ненадолго к первому примеру задачи, использующей понятие графа. В ней говорилось о коммивояжере, который должен пройти некоторое количество городов с минимальными затратами, найти, так сказать,

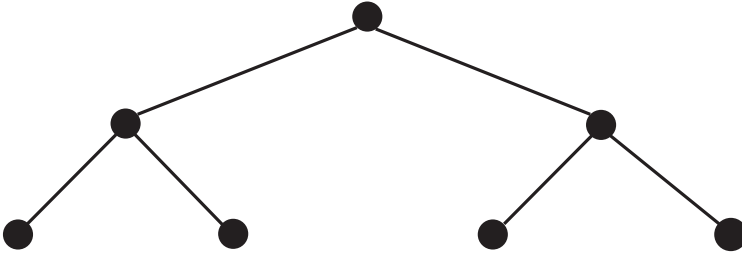


Рис. 11.4

кратчайший путь. Слова кратчайший, более дешевый говорят о том, что для решения задачи необходимо уметь сравнивать, а сравнивать на больше – меньше можно только числа, поэтому обычный граф уже не подходит.

Взвешенный граф – это граф, в котором каждой вершине (или ребру/дуге) сопоставлено некоторое число, называемое весом.

Подграфы. Граф может оказаться очень большим, содержащим огромное количество вершин и рёбер, составляющих очень сложную структуру. В такой ситуации часто бывает полезно выделить из большого графа меньший, с какими-то определёнными свойствами. Конечно, подграфы тоже могут быть сколь угодно сложными, но существуют три стандартных типа подграфов.

Частичным графом называется граф, состоящий из некоторого подмножества дуг исходного орграфа вместе с их концами.

Подграфом называется граф, состоящий из некоторого подмножества вершин исходного орграфа и тех дуг, оба конца которых принадлежат данному подмножеству.

Суграфом называется граф, содержащий все вершины исходного орграфа и некоторое подмножество дуг исходного орграфа.

Остовное дерево. Остовным деревом называется подграф взвешенного (веса присвоены ребрам) графа, обладающий следующими свойствами:

- содержит все вершины исходного графа;
- не содержит циклов (является деревом);
- является минимальным деревом из возможных.

Пример построения остовного дерева (рис. 11.5):

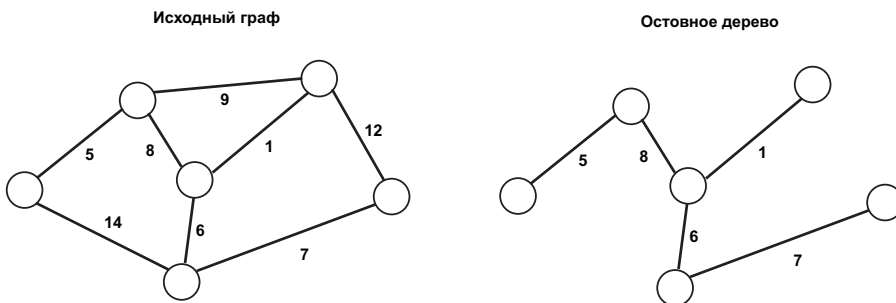


Рис. 11.5

Представление графа матрицами

Выше было дано формальное определение графа и было сказано, что картинка, иллюстрирующая граф и его свойства, – это не граф, а наглядное изображение. Зачастую же о картинке говорят как о самом графе. Это удобно, но все же необходимо понимать, что изображение математического объекта и сам математический объект – это не одно и то же.

Дело в том, что граф допускает ещё одно представление, не такое наглядное, но, быть может, более полезное. Это представление называется матрицей (таблицей) инцидентности/смежности.

Из формального определения следует, что граф состоит из двух видов компонентов: вершин и ребер (или дуг). Если ребро входит или выходит из вершины, то говорят, что эта вершина и это ребро инциденты друг другу.

Инцидентность (или неинцидентность) всех пар ребер и вершин можно записать в виде таблицы. Покажем на примере, как это делается. Возьмём простейший неориентированный граф. Вот такой (рис. 11.6):

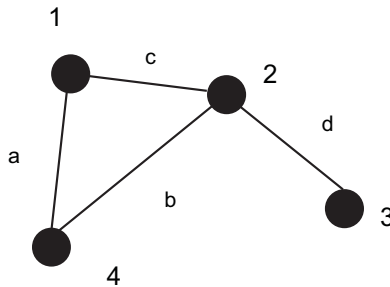


Рис. 11.6

Этот граф состоит из четырёх пронумерованных вершин и четырех ребер, обозначенных буквами a , b , c , d . Матрица инцидентности для этого графа выглядит так:

Таблица 11.2

	a	b	c	d
1	1	0	1	0
2	0	1	1	1
3	0	0	0	1
4	1	1	0	0

Дадим строгое определение матрицы.

Матрицей инцидентности называется матрица $a[i, j]$ – такая, что:

$a[i, j] = 1$, если i -е ребро инцидентно j -ой вершине;

$a[i, j] = 0$, если i -е ребро не инцидентно j -ой вершине.

Конечно, данное определение работает только для неориентированных графов.

Для ориентированных определение выглядит так:

$a[i, j] = 1$, если i -я дуга входит в j -ую вершину;
 $a[i, j] = -1$, если i -я дуга выходит из j -ой вершины;
 $a[i, j] = 0$, если i -я дуга неинцидентна j -ой вершине.

Необходимо заметить, что для ориентированного графа с петлями (то есть такими дугами, которые входят в ту же вершину, из которой и выходят) матрицу инцидентности не построить, так как одна и та же ячейка в таблице должна быть равна и 1, и -1. В этих случаях строят две матрицы, одна содержит информацию о входящих дугах, а вторая – о исходящих.

Ещё одна форма матричного представления графа называется матрицей смежности. Эта матрица содержит информацию о парах вершин, соединённых ребрами (дугами). Граф из предыдущего примера может быть представлен матрицей смежности так:

Таблица 11.3

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	1	0	0

Представление графа матрицами смежности и инцидентности помогает решать многие задачи. Для примера рассмотрим задачу поиска пути из пункта A в пункт B . Пункты A и B – это города на карте, на которой есть и ещё города. Некоторые из них соединены дорогами, некоторые нет. В общем случае такую карту можно представить в виде графа с циклами. Так как по каждой дороге можно ехать как в одну, так и в другую сторону, то этот граф конечно неориентированный. Представим этот граф в виде матрицы смежности.

Тогда из построенной матрицы легко определить, соединены ли два города дорогой непосредственно или нет. Напомним для последующих рассуждений, что название каждого города присутствует в верхней строке матрицы и в левом столбце. Вот так:

Таблица 11.4

	A	B
A		
B		

Только, конечно, в матрице, описывающей реальную ситуацию, строк и столбцов будет значительно больше. Если нас интересует, соединены ли два города дорогой, мы можем поступить следующим образом:

1. Имя первого найти в верхней строке матрицы. Соответствующий ему столбец обозначим как СТОЛБЕЦ.

2. Имя второго найти в левом столбце. Соответствующую ему строку обозначим как СТРОКА.
3. Найдём пересечение СТРОКИ и СТОЛБЦА. Если на пересечении стоит единица, то два города соединены дорогой, иначе – нет.

Нумерации на графе

Множество вершин графа никак не упорядочено, мы не можем сказать, какая вершина первая, какая вторая и т. д. Это плохо, если бы вершины графа пронумеровать, то работать с графом было бы намного проще. Поэтому сейчас займёмся разработкой способов нумерации вершин. Таких способов существует достаточно много, мы рассмотрим только один из них, называемый базисной нумерацией.

Базисная нумерация

Базисная нумерация строится на способах обхода графа в глубину. Вот алгоритм такого обхода:

1. Находясь в ОЧЕРЕДНОЙ вершине, ищем среди вершин, связанных с данной, ещё не пройденную.
2. Если таковая вершина нашлась, то переходим в неё, запоминая дугу, по которой пришли.
3. Если таковой вершины нет, то из ОЧЕРЕДНОЙ вершины возвращаемся в ПРЕДЫДУЩУЮ по дуге, которая была запомнена на предыдущем шаге.

Нумерация при обходе по описанному выше алгоритму строится так: узлы графа нумеруются в порядке, в котором они посещаются первый раз в процессе обхода графа. Нумерация, построенная таким образом, для ориентированных графов обладает несколькими свойствами:

Свойство 1: после завершения обхода орграфа при поиске в глубину частичный граф, образованный отмеченными дугами, есть корневое дерево с корнем в начале обхода.

Свойство 2: для того чтобы после завершения обхода орграфа при поиске в глубину все вершины оказались пройденными, необходимо и достаточно, чтобы обход начинался со входа и этот вход был единственным. При этом корневое ордеревое, образованное отмеченными дугами, имеет множество вершин, совпадающее со всем множеством вершин графа, и тем самым является каркасом графа, называемым деревом поиска в глубину.

Свойство 3: текущий путь есть простой путь в орграфе. Его можно продолжить либо до тупикового, т.е. заканчивающегося висячей вершиной, либо до циклического, то есть до встречи с уже пройденной вершиной.

Приложение 3.

Элементы теории вероятности

Предмет теории

Любая математическая теория имеет дело с величинами и необходимостью вычисления их значений. Иногда значение величины можно определить точно, таковой, например, будет величина наибольшего общего делителя. Иногда значение величины, возможно, определить лишь приблизительно, например $\sqrt{2}$, но значение все же есть. Теория вероятности имеет дело с иными величинами, чье точное значение в принципе получить невозможно. Классический пример – игральный кубик. Свяжем с кубиком величину, которую мы назовем числовое значение грани, выпавшее после броска кубика». У кубика 6 граней. И возможных значений 6. Но результатом конкретного броска будет не шесть значений, а одно. Сказать точно, какое именно, невозможно.

Величина такой природы называется случайной величиной, и именно она и является предметом изучения теории. Теория утверждает, что если нельзя точно сказать, какое значение примет случайная величина, это не означает, что нельзя сказать о ней ничего содержательного. И для случайных величин можно выделить несколько характеристик, которые делают их вполне закономерными и довольно-таки предсказуемыми. Ниже мы опишем несколько таких характеристик.

Вероятность случайной величины

Сразу заметим, что существует строгая теория вероятностей, которая не слишком увлекается объяснением природы характеристик случайных величин. Она построена, как все строгие математические теории, на аксиоматической основе. В её основу положены несколько неопределяемых понятий и несколько утверждений, принятых как истинные. Забота теории при таком подходе заключается в непротиворечивом развитии системы теорем. Наш подход будет более «физическим» и нестрогим.

Вернемся к игральному кубiku. Возможные значения изучаемой случайной величины – это числа: 1, 2, 3, 4, 5, 6. Интуитивно ясно, бросок кубика может дать любое из этих значений. И более того, нельзя сказать, что какое-то из них будет предпочтительнее других. Нельзя сказать, что скорее всего выпадет 1. Нельзя сказать, что скорее всего выпадет 4. Эту фразу «скорее всего выпадет...» нельзя сказать в отношении ни одного из возможных значений. В этом случае говорят, что значения случайной величины равновероятны. Термин «равновероятно» уже в какой-то степени является числовой оценкой, и её можно уточнить.

Введем понятие «*события*». Событие – это то, что уже произошло или может произойти. В нашем случае за событие можно считать полученное значение случайной величины. Событие можно описать фразой «выпало значение 3». Это событие. В случае с кубиком возможных событий для нашей случайной величины 6. Каждое из них может случиться, а может не случиться. Но есть одно событие, которое случится в любом случае. Это кубиковое событие обозначим фразой «*в результате броска кубика выпало некоторое значение случайной величины*». Действительно, какой бы гранью ни лег кубик, это событие всегда будет иметь место. Оно называется достоверным событием.

Обозначим его вероятность за единицу. Достоверное кубиковое событие состоит из шести возможных, все шесть возможных равнозначны, или, в нашей терминологии, равновероятны и в сумме должны дать достоверное. Отсюда можно сделать вывод, что вероятность любого из возможных следует принять за $1/6$, если считать, что вероятность достоверного равна 1.

Общий вывод. Для подсчета вероятности возможного события необходимо определить достоверное. Выше было сказано, что его вероятность равна единице. На самом деле это не есть объективная истина, это результат договоренности. Принципиально вероятность достоверного события может быть любой. Исходя из определенной вероятности достоверного события, определяются вероятности возможных. Пусть их N . Тогда вероятность каждого из возможных равна $1/N$.

Конечно, это несколько упрощенная схема рассуждений. Для того чтобы сказанное было верно, необходимо, чтобы события, составляющие достоверное, были равноправны, а это не всегда так. Необходимо, чтобы они были независимы, то есть одно событие фактом своего появления не должно изменить вероятности другого. А именно такая взаимозависимость зачастую и имеет место. Например, две случайные величины: количество осадков и размер урожая зерновых. Это не вполне случайные величины, но в некоторых пределах они ведут себя вероятностным образом. Понятно, что при очень маленьком размере осадков небольшие цифры собранного урожая становятся более вероятными. Все такие ситуации также являются предметом теории, но мы не ставим своей целью копнуть глубоко, наша цель – краткая сущность понятия.

Попробуем разобраться, что собственно означает фраза «вероятность события равна...». Рассмотрим еще один классический пример – монету. Бросок монеты может привести к двум событиям: выпала решка и выпал орел. Если решку обозначить за 1, а орла, например, за 0, то получим случайную величину с двумя возможными значениями $\{0, 1\}$. Вероятность для каждого из значений $1/2$. Рассмотрим серию из 10 испытаний. Вероятность в $1/2$ предполагает, что 5 раз выпадет орел и столько же раз – решка. Но этого может и не случиться. Вполне возможно событие «*10 раз выпала решка*» или «*10 раз выпал орел*». Кроме того, появляется масса и других возможных событий. Например, «*6 раз решка и 4 раза орел*». Любая комбинация орлов и решек имеет право на существование. И равновероятность орла и решки ни одну комбинацию не делает невозможной.

Но событие «*10 раз выпал орел*» маловероятно. Еще более маловероятно событие «*100 раз выпал орел*» при проведении сотни испытаний. При 10 испыта-

ниях можно ожидать 4, 5, 6 орлов, 3 или 7 уже пореже, 8 или 9 еще реже. При 100 испытаниях количество орлов будет ближе к 50, хотя 50 орлов и 50 решек таёже маловероятно, как и 100 орлов. Речь идет именно о приближении к 50, и только. Но это соображение наводит на мысль, что при увеличении количества испытаний количество выпадений орла (соответственно, решки) начинает все ближе и ближе колебаться вокруг $N/2$, если N – это количество испытаний. Можно сказать, что количество событий стремится к величине, определяемой вероятностью, при стремлении количества испытаний к бесконечности. Или более точно:

Вероятность события A можно рассматривать как предел отношения количеств событий A к общему количеству испытаний при стремлении последних к бесконечности.

Это так называемое частотное понимание вероятности. Им мы и ограничимся.

Математическое ожидание

Предположим, что выполняется серия испытаний, в которых определяется случайная величина, могущая принимать некоторое количество числовых значений: A_1, A_2, \dots, A_N . Значения вполне могут быть различны. Для общего случая предположим, что значения не равновероятны, их вероятности – это также некоторые числа: p_1, p_2, \dots, p_N . Но конечно же:

$$\sum_{k=1}^N p_k = 1.$$

Попробуем выяснить, какое среднее значение случайной величины можно ожидать после проведения серии испытаний. Рассмотрим частный случай. Пусть случайная величина имеет два равновероятных значения: $\{1, 2\}$. Равная вероятность означает, что в L испытаниях мы вправе ожидать $L/2$ единиц и $L/2$ двоек.

Следовательно, наше ожидание средней величины будет равно $\frac{1+2}{2} = 1.5$. Пусть

теперь эти два значения не равновероятны. Пусть $p_1=1/3$ и $p_2=2/3$. Это означает, что в L испытаниях мы ожидаем увидеть $L/3$ единиц и $2L/3$ двоек. Среднее же значение будет получено такое:

$$\frac{1 * \frac{L}{3} + 2 * \frac{2L}{3}}{L} = \frac{1}{3} + \frac{4}{3} = \frac{5}{3}.$$

Заметим, что окончательное значение не содержит числа L . То есть получена характеристика собственно случайной величины. Эта собственная характеристика называется математическим ожиданием случайной величины. А ниже – формула для общего случая:

$$M = \sum_{k=1}^N A_k p_k .$$

Характеристики разброса случайной величины

Случайность величины означает принципиальную возможность любого значения. Понятно, что разные значения имеют разные вероятности, но вероятности все же не нулевые. Поэтому можно представить случайную величину размазанной по некоторой области. Математическое ожидание говорит о том, к какой точке этой области значения величины сгущаются, становятся плотнее. Но математическое ожидание не характеризует самой области. Необходимо еще как-то охарактеризовать величину разброса случайной величины в пределах области, и такой характеристикой является дисперсия.

Нулевая дисперсия означает нулевой разброс. То есть при нулевой дисперсии все значения лягут в точке математического ожидания, и случайная величина превратится в закономерную. Ненулевая дисперсия, таким образом, будет означать наличие отклонения от точки математического ожидания. Это отклонение есть разность между значением случайной величины и её математическим ожиданием. Но разброс случайной величины описывают не разностью, а её квадратом, то есть величиной вида:

$$(X - Mx)^2 .$$

Так как в данное выражение входит случайная величина X , то и сама величина отклонения также является случайной. Следовательно, для неё также можно посчитать математическое ожидание:

$$Dx = M(x - Mx)^2 = \sum_x (x - Mx)^2 p(x) .$$

Математическое ожидание квадрата отклонения и называется дисперсией. Еще одна характеристика – это среднеквадратичное отклонение, связанное с дисперсией следующей простой формулой:

$$Dx = \sigma^2 .$$

Центральная предельная теорема и распределение случайной величины

Значения случайной величины могут распределяться как угодно, на то она и случайная. Так, наверное, кажется с точки зрения здравого смысла. Действительно, если мы ничего про величину не знаем, то, наверное, ничего не можем сказать и о распределении её значений. Но это все же не так. Действительность несколько сложнее. Во-первых, сама фраза «мы ничего не знаем о величине» к случайной величине не применима. Парадокс заключается в том, что утверждение о случайном

характере величины дает очень серьезную закономерную информацию. Попробуем её выделить.

Еще раз о природе случайности. Почему, бросая монетку, мы не можем точно сказать, как она упадет, орлом или решкой? На самом деле можем. Но для этого надо точно знать массу монетки, как эта масса распределена, крутящий момент, передаваемый монете щелчком, и вертикальную скорость, которую она также получает от щелчка, сопротивление воздуха, возможно, на процесс влияет и что-то еще. Если все эти факторы удастся точно учесть, то можно вычислить, как она упадет. Возможно, расчеты окажутся достаточно сложными, но принципиально здесь ничего невозможного нет. Проблема заключается в очень высокой степени сложности учета влияния каждого фактора и взаимосвязи. В действительности все посчитать мы не можем или не хотим, поэтому и объявляем падение монеты случайной величиной.

Невозможность точного расчета – условие обязательное, но не достаточное. Бутерброд чаще всего падает маслом вниз, это нам известно, но расчет точной траектории бутерброда вряд ли проще расчета траектории монеты. Дело здесь в том, что сторона, намазанная маслом, тяжелее, то есть появляется фактор, чье точное воздействие посчитать нельзя, но он более значим, нежели другие, в результате чего процесс становится менее случайным. Отсюда вывод: *в качестве достаточного условия необходимо потребовать, кроме множественности факторов и трудности их учета, их равный вклад в выбор значения случайной величины в каждом испытании.*

Что это дает? Оказывается, при выполнении этого требования распределение случайной величины будет не каким угодно, а вполне определенным, известным математике под названием нормального распределения. В этом и заключается смысл центральной предельной теоремы.

Несколько слов о нормальном распределении. Построим систему координат, в которой на оси абсцисс отложены значения случайной величины, а на оси ординат – вероятности значений. Построим график какого-либо нормального распределения.

Из рис. 11.7 видны основные свойства нормального распределения. Величина плавно возрастает, достигает максимума в точке математического ожидания и затем плавно убывает. График нормального распределения обязательно симметричен относительно линии перпендикуляра к точке математического ожидания. График имеет форму колокола. Очевидно, что ширина колокола характеризует разброс случайной величины, следовательно, его ширина определяется дисперсией случайной величины.

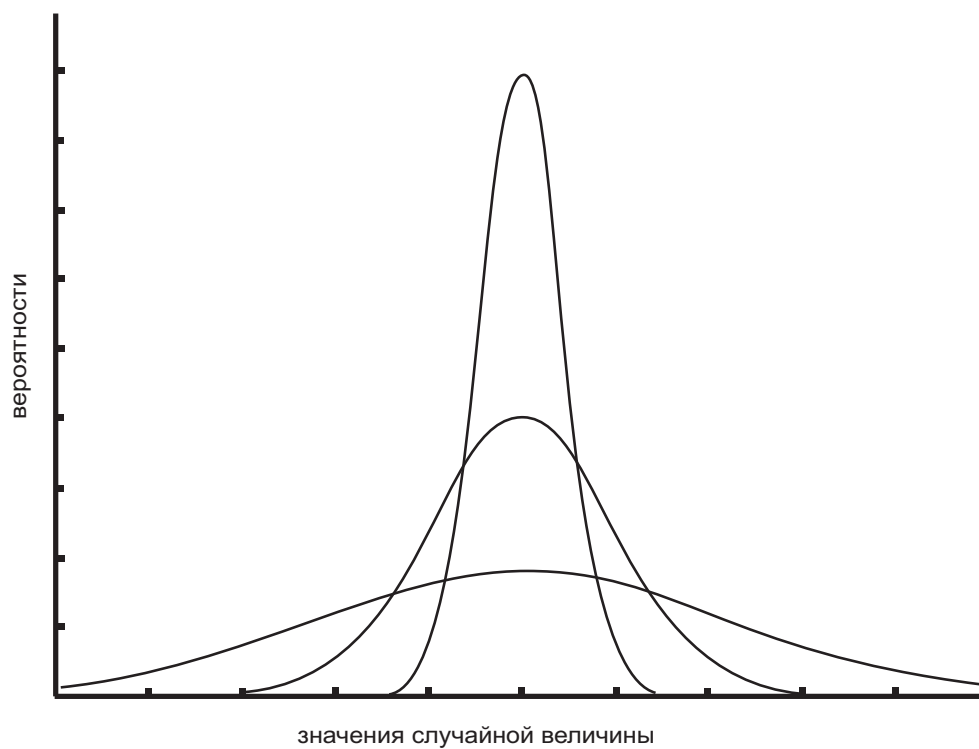


Рис. 11.7

Приложение 4.

Синтаксис языка

Компонентный Паскаль

Понятие идентификатора

ident = (letter | “_”) {letter | “_” | digit}.

letter = “A” .. “Z” | “a” .. “z” | “А” .. “Ц” | “Ш” .. “ц” | “ш” .. “я”.

digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”.

Константные выражения

ConstantDeclaration = IdentDef “=” ConstExpression.

ConstExpression = Expression.

Переменная

VariableDeclaration = IdentList “:” Type.

Объявление типа

TypeDeclaration = IdentDef “=” Type.

Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.

- TypeDeclaration – объявление типа;
- IdentDef – определяемый идентификатор;
- Type – тип;
- Qualident – уточненный идентификатор;
- ArrayType – тип массива;
- RecordType – тип записи;
- PointerType – указательный тип;
- ProcedureType – процедурный тип.

Основные типы

Имя типа	Значения
BOOLEAN	Логические значения TRUE и FALSE
SHORTCHAR	Литеры набора Latin-1 (0X .. 0FFX)
CHAR	Литеры набора Unicode (0X .. 0FFFFX)
BYTE	Целые от MIN(BYTE) до MAX(BYTE)
SHORTINT	Целые от MIN(SHORTINT) до MAX(SHORTINT)
INTEGER	Целые от MIN(INTEGER) до MAX(INTEGER)
LONGINT	Целые от MIN(LONGINT) до MAX(LONGINT)
SHORTREAL	Вещественные числа от MIN(SHORTREAL) до MAX(SHORTREAL), значение INF (INF – предопределенное значение, которым замещается вещественная величина в случае выхода за пределы допустимого интервала. Знак INF совпадает со знаком исходного значения)
REAL	Вещественные числа от MIN-REAL) до MAX-REAL), значение INF
SET	Множества целых чисел из диапазона от 0 до MAX(SET)

Типы записей

RecordType = RecAttributes RECORD [“(BaseType)”]
 FieldList {“,” FieldList} END.
 RecAttributes = [ABSTRACT | EXTENSIBLE | LIMITED].
 BaseType = Qualident.
 FieldList = [IdentList “.” Type].
 IdentList = IdentDef {“,” IdentDef}.

Указательный тип

PointerType = POINTER TO Type.

Определение операций

Expression = SimpleExpression [Relation SimpleExpression].
 SimpleExpression = [“+” | “-”] Term {AddOperator Term}.
 Term = Factor {MulOperator Factor}.
 Factor = Designator | number | character | string | NIL | Set |
 “(Expression)” | “~” Factor.
 Set = “{” [Element {“,” Element}] “}”.
 Element = Expression [“..” Expression].

Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.

AddOperator = "+" | "-" | OR.

MulOperator = "*" | "/" | DIV | MOD | "&"

Условный оператор

```
IfStatement = IF Expression THEN StatementSequence
               {ELSIF Expression THEN StatementSequence}
               [ELSE StatementSequence]
               END.
```

Оператор выбора

```
CaseStatement = CASE Expression OF Case {" Case}
                 [ELSE StatementSequence] END.
```

Case = [CaseLabelList ":" StatementSequence].

CaseLabelList = CaseLabels {" ," CaseLabels}.

CaseLabels = ConstExpression [".." ConstExpression].

Цикл с предусловием (WHILE)

```
WhileStatement = WHILE Expression DO StatementSequence END.
```

Цикл с постусловием (REPEAT UNTIL)

```
RepeatStatement = REPEAT StatementSequence UNTIL Expression.
```

Цикл с шагом (FOR TO BY DO END)

```
ForStatement = FOR ident ":" Expression TO Expression [BY ConstExpression]
               DO StatementSequence END.
```

Безусловный цикл (LOOP)

```
LoopStatement = LOOP StatementSequence END.
```

Оператор конкретизации типа WITH

```
WithStatement = WITH [ Guard DO StatementSequence ]
                 {" [ Guard DO StatementSequence} ]
                 [ELSE StatementSequence] END.
```

Guard = Qualident ":" Qualident.

Описание процедуры

```
ProcedureDeclaration = ProcedureHeading [";" ProcedureBody ident ].
```

ProcedureHeading = PROCEDURE [Receiver] IdentDef

```

[FormalParameters] MethAttributes.
ProcedureBody = DeclarationSequence
[BEGIN StatementSequence] END.
DeclarationSequence = {CONST {ConstantDeclaration “,”} |
    TYPE {TypeDeclaration “,”} |
    VAR {VariableDeclaration “,”} }
    {ProcedureDeclaration “,” | ForwardDeclaration “,”}.
ForwardDeclaration = PROCEDURE “^” [Receiver] IdentDef
[FormalParameters] MethAttributes.

```

Список формальных параметров

```

FormalParameters = “(“ [FPSection {“,” FPSection}] “)” [“:” Type].
FPSection = [VAR | IN | OUT] ident {“,” ident} “:” Type.

```

Определение модуля

```

Module = MODULE ident “,” [ImportList] DeclarationSequence
[BEGIN StatementSequence]
[CLOSE StatementSequence] END ident “.”.
ImportList = IMPORT Import {“,” Import} “,”.
Import = [ident “:=”] ident.

```

Оператор возврата

```

RETURN

```

Оператор прерывания

```

EXIT

```

Список литературы

1. Дал У., Дейкстра Э., Хоор К. Структурное программирование. – М.: Вильямс, 1975
2. Кнут Д. Получисленные алгоритмы, Т. 2. – М.: Вильямс, 2000
3. Кнут Д. Сортировка и поиск, Т. 3.М., – М.: Вильямс, 2000
4. Потопахин В.В. Решение сложных задач. – СПб.: БХВ-Петербург, 2006
5. Потопахин В.В. Современное программирование с нуля. – М.: ДМК-Пресс, 2010
6. Брассар Ж. Современная криптология. – М.: Полимед, 1999
7. Сمارт Н. Криптография. – М.: Техносфера, 2005
8. Василенко О. Н. Теоретико-числовые алгоритмы в криптографии. – М.,: МЦМНО, 2003
9. Виноградов И. М. Основы теории чисел.
10. Бухштаб Теория чисел – М.: Лань, 2004
11. Вейль А. Основы теории чисел. – М.: Едиториал, УРРС, 2004
12. Савельев А. Я. Основы Информатики. – М.: МГТУ им. Баумана, 2001
13. Гельфонд А. О. Решение уравнений в целых числах. – М.: Наука, 1983
14. Серпинский В. Н. О решении уравнений в целых числах. – М, Наука, 1961
15. Вирт Н. Алгоритмы и структуры данных. – М.: ДМК Пресс, 2010
16. Свами М., Тхуларисиман К. Графы, сети и алгоритмы. – М.: Мир, 1984
17. Кудрявцев Л. Д. Математический анализ. – М.: Высшая школа, 1973
18. Феллер В. Введение в теорию вероятностей и её приложения. – Либроком, 2010
19. Марчук Г. И. Методы вычислительной математики. – М.: Наука, 1977
20. Кларнер Д. А. Математический цветник. – М.: Мир, 1983
21. Соболев И. М. Метод Монте-Карло. – М.: Наука, 1985
22. Дейкстра Э. В. Дисциплина программирования. – М.: Мир, 1978
23. Кунцман Ж. Численные методы – М.: Наука, 1979
24. Бахвалов Н. С. Численные методы. – Лаборатория базовых знаний, 2003

Книги издательства «ДМК Пресс» можно приобрести в торгово-издательском холдинге «АЛЬЯНС-КНИГА» (АЛЬЯНС БУКС) наложенным платежом или выслать письмо на почтовый адрес: 115533, Москва, Нагатинская наб., д.6, стр.1.

При оформлении заказа в письме следует указать полностью Ф.И.О. и почтовый адрес заказчика (с индексом).

Эти книги Вы также можете заказать на сайте: **www.aliants-kniga.ru**.

Оптовые продажи: тел. **(495) 258-91-94, 258-91-95 (факс)**.

Электронный адрес: **books@aliants-kniga.ru**.

Потопахин Виталий Валерьевич

Искусство алгоритмизации

Главный редактор Мовчан Д. А.
dm@dmk-press.ru

Корректор Синяева Г. И.

Верстка Паранская Н. В.

Дизайн обложки Мовчан А. Г.

Подписано в печать 17.11.2010. Формат 70×100 1/16 .

Гарнитура «Литературная». Печать офсетная.

Усл. печ. л. 26,65. Тираж 1000 экз.

Заказ №

Web-сайт издательства: www.dmkpress.ru

Электронный адрес издательства: books@dmkpress.ru