

Учебник входит в УМК по информатике для старшей школы (10–11 классы).
Соответствует федеральному государственному образовательному стандарту
среднего (полного) общего образования (2012 г.).

Включён в федеральный перечень учебников, рекомендованных Министер-
ством образования и науки Российской Федерации.



Константин Юрьевич Поляков — доктор технических наук, профес-
сор кафедры судовой автоматики и измерений Санкт-Петербургского
государственного морского технического университета, учитель ин-
форматики школы № 163 Санкт-Петербурга. Победитель Всеросий-
ского конкурса для педагогов по включению ресурсов Единой кол-
лекции цифровых образовательных ресурсов в образовательный
процесс. Лауреат профессиональной премии «Лучший учитель Санкт-Петербурга».
Награжден знаком «Почетный работник общего образования РФ». Ведущий автор
руководитель авторского коллектива по разработке комплекта учебников по ин-
форматике углубленного уровня.



Евгений Александрович Еремин — кандидат физико-математиче-
ских наук. Преподает связанные с информатикой курсы с момента
появления этого предмета в школе в 1985 году. Хорошо известен
по публикациям в газете «Информатика» издательства «Первое
сентября» (около 50 статей). Всего имеет более 150 публикаций, в
том числе несколько книг.

К. Ю. Поляков
Е. А. Еремин

ИНФОРМАТИКА 11 (2)

ФГОС

11



К. Ю. Поляков
Е. А. Еремин

ИНФОРМАТИКА

2

УГЛУБЛЕННЫЙ УРОВЕНЬ

SBN 978-5-9963-3293-9



ИЗДАТЕЛЬСТВО
БИНОМ

УДК 004.9
ББК 32.97
П54

П54 Поляков К. Ю.
Информатика. Углубленный уровень : учебник для
11 класса : в 2 ч. Ч. 2 / К. Ю. Поляков, Е. А. Еремин. —
6-е изд., стереотип. — М. : БИНОМ. Лаборатория знаний,
2017. — 312 с. : ил.

ISBN 978-5-9963-3293-9 (Ч. 2)
ISBN 978-5-9963-3294-6

Учебник предназначен для изучения информатики на углубленном уровне в 11 классах общеобразовательных организаций. Содержание учебника является продолжением курса 10 класса и опирается на изученный в 7–9 классах курс информатики для основной школы.

Рассматриваются вопросы передачи информации, информационные системы и базы данных, разработка веб-сайтов, компьютерное моделирование, методы объектно-ориентированного программирования, компьютерная графика и анимация.

Учебник входит в учебно-методический комплект (УМК), включающий в себя также учебник для 10 класса и компьютерный практикум.

Предполагается широкое использование ресурсов портала Федерального центра электронных образовательных ресурсов (<http://fcior.edu.ru/>).

Соответствует федеральному государственному образовательному стандарту среднего общего образования (2012 г.).

УДК 004.9
ББК 32.97

Учебное издание

Поляков Константин Юрьевич
Еремин Евгений Александрович

ИНФОРМАТИКА. УГЛУБЛЕННЫЙ УРОВЕНЬ

Учебник для 11 класса
В 2 частях
Часть 2

Ведущий редактор О. Полежаева
Ведущие методисты И. Сметенская, И. Хлобыстова
Художник Н. Новак
Технический редактор Е. Денюкова
Корректор Е. Климина
Компьютерная верстка В. Носенко

Подписано в печать 20.07.17. Формат 70х100/16. Усл. печ. л. 25,35.
Тираж 3000 экз. Заказ 6200.

ООО «БИНОМ. Лаборатория знаний»
127473, Москва, ул. Краснопролетарская, д. 16, стр. 1,
тел. (495) 181-53-44, e-mail: binom@lbz.ru
<http://www.lbz.ru>, <http://metodist.lbz.ru>



Отпечатано в ОАО «Можайский полиграфический комбинат»
143200, г. Можайск, ул. Мира, 93.
www.omompk.ru, www.omompk.ru тел.: (495) 745-84-28, (49638) 20-685

Оглавление

Глава 5. Элементы теории алгоритмов	5
§ 34. Уточнение понятия алгоритма	5
§ 35. Алгоритмически неразрешимые задачи	20
§ 36. Сложность вычислений	26
§ 37. Доказательство правильности программ	36
Глава 6. Алгоритмизация и программирование	49
§ 38. Целочисленные алгоритмы	49
§ 39. Структуры (записи)	57
§ 40. Множества	66
§ 41. Динамические массивы	72
§ 42. Списки	79
§ 43. Стек, очередь, дек	88
§ 44. Деревья	101
§ 45. Графы	113
§ 46. Динамическое программирование	129
Глава 7. Объектно-ориентированное программирование	143
§ 47. Что такое ООП?	143
§ 48. Объекты и классы	146
§ 49. Создание объектов в программе	151
§ 50. Скрытие внутреннего устройства	157

ISBN 978-5-9963-3293-9 (Ч. 2)
ISBN 978-5-9963-3294-6

© ООО «БИНОМ. Лаборатория знаний», 2013

§ 51. Иерархия классов	163
§ 52. Программы с графическим интерфейсом	177
§ 53. Основы программирования в RAD-средах	181
§ 54. Использование компонентов	188
§ 55. Совершенствование компонентов	197
§ 56. Модель и представление	202
Глава 8. Компьютерная графика и анимация	211
§ 57. Основы растровой графики	211
§ 58. Ввод изображений	215
§ 59. Коррекция фотографий	220
§ 60. Работа с областями	227
§ 61. Фильтры	230
§ 62. Многослойные изображения	232
§ 63. Каналы	237
§ 64. Иллюстрации для веб-сайтов	241
§ 65. Анимация	244
§ 66. Контуры	247
Глава 9. Трёхмерная графика	251
§ 67. Введение	251
§ 68. Работа с объектами	256
§ 69. Сеточные модели	261
§ 70. Модификаторы	267
§ 71. Кривые	272
§ 72. Материалы и текстуры	276
§ 73. Рендеринг	283
§ 74. Анимация	292
§ 75. Язык VRML	302

Глава 5

Элементы теории алгоритмов

§ 34

Уточнение понятия алгоритма

Зачем нужно определение алгоритма?

Как вы знаете, **алгоритмом** называют точный набор инструкций для исполнителя, который приводит к решению задачи за конечное время.

Особый интерес проявляли к алгоритмам математики. Один из древнейших известных алгоритмов — алгоритм Евклида для вычисления наибольшего общего делителя (НОД) двух натуральных чисел. Само слово «алгоритм» (от имени математика IX века аль-Хорезми, которого считают основателем алгебры) вошёл в науку в XVII веке немецкий математик Г. В. Лейбниц.

Долгое время считалось, что для любой математической задачи можно найти метод (алгоритм) решения, просто для ряда задач такие алгоритмы ещё не найдены. Эту идею высказал аль-Хорезми, такой же точки зрения придерживались и другие математики вплоть до начала XX века.

Однако, несмотря на все усилия, решить некоторые задачи не удавалось в течение столетий. Например, безуспешно закончились многочисленные попытки найти алгоритм доказательства правильности любой теоремы на основе заданной системы аксиом.

В 1931 г. австрийский математик К. Гёдель доказал теорему о неполноте, смысл которой состоит в том, что в любой достаточно сложной формальной системе, основанной на аксиомах (например, в арифметике, где введены натуральные числа и операции сложения и умножения), есть утверждение, которое невозможно ни доказать, ни опровергнуть в рамках этой системы. Поэтому было высказано предположение о том, что некоторые задачи **алгоритмически неразрешимы**, т. е. для них в принципе не существует алгоритма решения, и поэтому искать его бессмысленно. Чтобы строго доказать или опровергнуть эту гипотезу, нужно было ввести математическое понятие алгоритма.

«Определение» алгоритма, которое мы привели в начале главы, часто называют *интуитивным*, потому что оно содержит такие

«нематематические» понятия, как «точный набор», «инструкция», «исполнитель», «решение задачи». Эти термины невозможно записать строго, используя язык математики и логики, поэтому для математического доказательства такое определение не подходит.

Исследования в этой области, которые начали активно проводиться в 30-х годах XX века, привели к возникновению **теории алгоритмов**, которая занимается:

- доказательством алгоритмической неразрешимости задач;
- анализом сложности алгоритмов;
- сравнительной оценкой качества алгоритмов.

Значительный вклад в развитие теории алгоритмов внесли математики А. Тьюринг (Великобритания), Э. Пост (США), А. Чёрч (Великобритания), С. Клини (США) и А. А. Марков (СССР).

Что такое алгоритм?

Первые известные алгоритмы — это правила выполнения арифметических действий с числами. В них чётко определены объекты (числа в десятичной записи) и элементарные шаги (сложить, вычесть, перемножить два однозначных числа — вспомните таблицы сложения и умножения). Постепенно сложность задач, которые решались с помощью алгоритмов, увеличивалась, и понятие «шаг алгоритма» оказалось нечётким, размытым. Например, можно ли считать элементарным шагом разложение числа на простые множители или сложение многозначных чисел?

Со временем понятие алгоритма расширилось — сейчас мы говорим об алгоритмах для исполнителей, которые работают с текстами и другими объектами реального мира. Однако оказалось, что все эти объекты можно тем или иным способом закодировать в виде цепочек символов, так что любой алгоритм сводится к преобразованию одной символьной строки в другую. В алгоритме шахматной игры объекты — это фигуры на доске, но их расположение легко закодировать в символьной форме (вспомните запись шахматных партий). Таким же способом можно представить и классические вычислительные алгоритмы — операции с цифрами.

Поэтому можно рассматривать только алгоритмы обработки символьных строк, а полученные результаты будут применимы к любым алгоритмам. Как вы знаете, текст, записанный с помощью любого алфавита, всегда можно перевести в двоичный код, поэтому, вообще говоря, достаточно рассматривать только алгоритмы, работающие с двоичными последовательностями.

Про любой алгоритм можно сказать следующее:

- алгоритм получает на вход дискретный объект (например, слово);
- алгоритм обрабатывает входной объект по шагам (дискретно), строя на каждом шаге промежуточные дискретные объекты; этот процесс может закончиться или не закончиться;
- если выполнение алгоритма заканчивается, его результат — это объект, построенный на последнем шаге;
- если выполнение алгоритма не заканчивается (алгоритм закикливается) или заканчивается аварийно (например, в результате деления на 0), то результат его работы при данном входе не определён.

Любой алгоритм рассчитан на определённого исполнителя: он должен использовать только понятные этому исполнителю команды. Задание для исполнителя — это текст на специальном (формальном) языке, который обычно называют программой. Поэтому можно определить алгоритм как программу для некоторого исполнителя.

Напомним, что, с точки зрения теории алгоритмов, достаточно рассматривать только алгоритмы, работающие с цепочками символов, которые называют **словами** (рис. 5.1).



Рис. 5.1

Каждый алгоритм задаёт (вычисляет) функцию, которая преобразует входное слово в результат (выходное слово). Такая функция может быть не определена для некоторых входных слов, если алгоритм закикливается или завершается аварийно.

Функция, заданная алгоритмом, может быть нигде не определена. Например, алгоритм

```

нц пока да
кц
  
```

закикливается при любом входном слове.

Алгоритмы называются **эквивалентными**, если они задают одну и ту же функцию. То есть при любом входном слове оба алгоритма должны приводить к одному и тому же результату или заикливаться (оба алгоритма не выдают никакого результата). Например, следующие алгоритмы для выбора минимального из значений переменных a и b эквивалентны:

если $a < b$ то	$M := b$
$M := a$	если $a < b$ то
иначе	$M := a$
$M := b$	все
все	

Универсальные исполнители

Как мы уже видели, понятие алгоритма оказывается «привязанным» к его исполнителю и некоторому языку программирования. Это не позволяет определить алгоритм как математический объект. Поэтому возникла идея попытаться построить **универсальный исполнитель**.

Универсальный исполнитель — это исполнитель, который может моделировать работу любого другого исполнителя. Это значит, что для любого алгоритма, написанного для любого исполнителя, существует эквивалентный алгоритм для универсального исполнителя.

Такой исполнитель можно было бы использовать для доказательства разрешимости или неразрешимости задач. Если удастся построить алгоритм решения задачи для универсального исполнителя, то задача разрешима. Если доказано, что алгоритм не существует, то задача неразрешима. Система команд такого исполнителя должна быть как можно проще — так его будет легче использовать в доказательствах.

В середине XX века разными учёными независимо друг от друга были предложены несколько исполнителей, претендующих на роль универсальных (они будут рассмотрены далее), причём в теории алгоритмов доказано, что все они эквивалентны друг другу. Это означает, что для любого алгоритма для одного уни-

версального исполнителя можно построить эквивалентный алгоритм для другого универсального исполнителя.

Как же связан универсальный исполнитель с проблемой строгого определения алгоритма?

Любой алгоритм может быть представлен как программа для универсального исполнителя.

Это основная идея теории алгоритмов. Строго доказать это утверждение невозможно, потому что здесь используется интуитивное понятие «алгоритм».

Как мы увидим, каждый универсальный исполнитель описывается с помощью математических терминов, поэтому на его основе можно дать строгое определение алгоритма:

Алгоритм — это программа для универсального исполнителя.

Универсальный исполнитель — это некоторая модель вычислений, которая задаёт способ описания алгоритмов и их выполнения. Модель вычислений должна содержать:

- «процессор», задающий систему команд и способ их выполнения;
- «память», определяющую способ хранения данных;
- язык программирования (способ записи программ);
- способ ввода данных (чтения входного слова);
- способ вывода слова-результата.

Все универсальные исполнители эквивалентны, поэтому последнее приведённое определение алгоритма не зависит от конкретного исполнителя.

Машина Тьюринга

Первым предложил универсальный исполнитель английский математик Алан Тьюринг. Придуманное им воображаемое устройство состоит из трёх частей:

- бесконечной ленты, разделённой на ячейки;
- каретки (читающей и записывающей головки);
- программируемого автомата.

Программируемый автомат управляет кареткой, посылая ей команды в соответствии с заложенной в него сменяемой программой. Лента выполняет роль памяти компьютера, автомат — роль процессора, а каретка служит для ввода и вывода данных. Такое устройство называют **машиной Тьюринга**.

Теоретически лента в машине Тьюринга бесконечна, однако в каждый момент времени работы машины используется лишь конечная её часть.

Каретка в любой момент времени находится над одной ячейкой, автомат может читать и изменять содержимое этой ячейки, которая называется **текущей (рабочей) ячейкой** (рис. 5.2).



А. Тьюринг
(1912–1956)

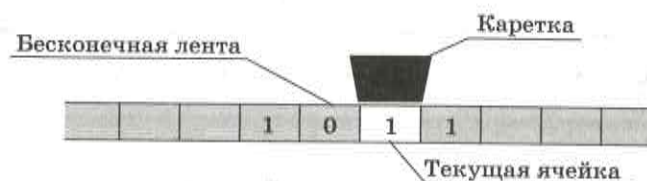


Рис. 5.2

В каждую ячейку ленты можно записать один любой символ, принадлежащий выбранному алфавиту. Любой алфавит обязательно содержит пробел (пустой символ, соответствующий «чистым» участкам ленты), который мы будем обозначать знаком $\square$. Алфавит обычно обозначается буквой A , а его элементы — строчными буквами a с индексами: $A = \{a_1, a_2, \dots, a_N\}$. Например, алфавит машины Тьюринга, работающей с двоичными числами, задаётся в виде $A = \{0, 1, \square\}$.

Непрерывную цепочку символов на ленте называют **словом**. На рисунке 5.2 лента содержит слово «1011», которое можно воспринимать как двоичное число.

Автоматом называют устройство, работающее без участия человека. Автомат в машине Тьюринга имеет несколько **состояний** и при определённых условиях переходит из одного состояния в другое. Состояние автомата определяет ту промежуточную задачу,

которую решает автомат в данный момент. Это напоминает состояния человека: ночью он спит (состояние 1), утром встаёт и умывается (состояние 2), завтракает (состояние 3), идёт на работу (состояние 4) и т. д.

Множество всех состояний автомата обозначается буквой Q , а его элементы — строчными буквами q с индексами: $Q = \{q_1, q_2, \dots, q_M\}$. Принято, что в начальный момент машина Тьюринга находится в состоянии q_1 .

Особое состояние q_0 — это **состояние останова**. Если машина переходит в это состояние, выполнение программы сразу останавливается.

Автомат управляется **программой**. Во время каждого шага программы автомат выполняет последовательно три действия:

- 1) изменяет символ в рабочей ячейке на другой (или оставляет без изменений);
- 2) перемещает каретку влево или вправо (или оставляет на месте);
- 3) переходит в другое состояние (или остаётся в прежнем состоянии).

Поэтому при составлении программы для каждой пары (*символ, состояние*) нужно определить три параметра: символ a_i из выбранного алфавита A , направление перемещения каретки ($\leftarrow$ — влево, $\rightarrow$ — вправо, точка — нет перемещения) и новое состояние автомата q_k . Например, команда $1 \leftarrow q_2$ обозначает «заменить символ на 1, переместить каретку влево на 1 ячейку и перейти в состояние q_2 ».

Пример 1. На ленте записано число в двоичной системе счисления. Каретка находится где-то над числом. Требуется увеличить число на единицу.

Для того чтобы построить машину Тьюринга, нужно:

- определить алфавит машины Тьюринга A ;
- выделить простейшие подзадачи и определить набор возможных состояний Q ; задать начальное состояние q_1 и конечное состояние q_0 (в котором машина останавливается);
- составить программу, т. е. для каждой пары (a_i, q_k) определить команду, которую должен выполнить автомат.

Как мы уже выяснили, алфавит машины Тьюринга, работающей с двоичными числами, включает символы 0, 1 и пробел: $A = \{0, 1, \square\}$. Определим возможные состояния (разобьём задачу на элементарные подзадачи):

- 1) q_1 — автомат ищет правый конец слова (числа) на ленте;
- 2) q_2 — автомат увеличивает число на 1, проходя его справа налево, и останавливается, закончив работу.

Теперь займёмся программой. На первом этапе, когда автомат ищет конец слова, его работа может быть описана так:

- 1) если в рабочей ячейке записана цифра 0, переместиться вправо;
- 2) если в рабочей ячейке записана цифра 1, переместиться вправо;
- 3) если в рабочей ячейке пробел, переместить каретку влево и перейти в состояние q_2 .

Тогда действия автомата в состоянии q_1 можно представить в виде таблицы, где в заголовках строк записываются символы алфавита, а в заголовках столбцов — состояния (рис. 5.3).

	q_1
0	$0 \rightarrow q_1$
1	$1 \rightarrow q_1$
$\square$	$\square \leftarrow q_2$

Рис. 5.3

Заметим, что во всех случаях символ под кареткой не меняется. Кроме того, состояние меняется только в последней ячейке. Поэтому для упрощения записи не будем указывать в таблице то, что остаётся без изменений. Так, на наш взгляд, более кратко и понятно (рис. 5.4).

	q_1
0	$\rightarrow$
1	$\rightarrow$
$\square$	$\leftarrow q_2$

Рис. 5.4

Второй этап — увеличение двоичного числа на единицу. Это можно сделать следующим способом (вспомните тему «Системы счисления»):

- 1) если в рабочей ячейке записана цифра 0, записать в неё 1 и стоп;
- 2) если в рабочей ячейке записана цифра 1, выполнить перенос в старший разряд — записать в ячейку 0 и переместиться влево;
- 3) если в рабочей ячейке пробел, записать в неё 1 и стоп.

Для того чтобы остановить работу машины Тьюринга, нужно перевести её в состояние останова q_0 . Теперь можно добавить в таблицу столбец, соответствующий состоянию q_2 (рис. 5.5).

	q_1	q_2
0	$\rightarrow$	$1 \cdot q_0$
1	$\rightarrow$	$0 \leftarrow$
$\square$	$\leftarrow q_2$	$1 \cdot q_0$

Рис. 5.5

Построенная полная таблица (см. рис. 5.5) — это и есть программа для машины Тьюринга. Обратите внимание, что мы разбили исходную задачу на подзадачи, для каждой из них составили программу, а потом их соединили. Две подзадачи связаны через ячейку $(\square, q_1)$, в которой состояние автомата изменяется на q_2 . В данном простейшем случае в каждом из двух алгоритмов было использовано только одно состояние, но это не обязательно — можно таким же способом соединять и более сложные алгоритмы. Если алгоритмы A и B можно запрограммировать на машине Тьюринга, то и любую их комбинацию тоже можно запрограммировать.

Тьюринг предположил, что:

Любой алгоритм (в интуитивном смысле этого слова) может быть представлен как программа для машины Тьюринга.

Это утверждение в теории алгоритмов известно как тезис Чёрча–Тьюринга.

Машина Тьюринга может быть строго задана с точки зрения математики. Алфавит A и набор возможных состояний Q могут быть записаны в виде множеств, а программа — в виде пятёрок вида $(a_i, q_k, a_j, d_{ik}, q_m)$, задающих команду «если машина находится в состоянии q_k и в рабочей ячейке записан символ a_i , то записать в рабочую ячейку символ a_j , сместиться в направлении d_{ik} и перейти в состояние q_m ». Например, приведённая выше программа увеличения двоичного числа на 1, записанная в виде таких пятёрок, выглядит так:

$(0, q_1, 0, \rightarrow, q_1), (1, q_1, 1, \rightarrow, q_1), (\square, q_1, \square, \leftarrow, q_2),$
 $(0, q_2, 1, \cdot, q_0), (1, q_2, 0, \leftarrow, q_2), (\square, q_2, 1, \cdot, q_0).$

Эта машина — математический объект, и данное на её основе определение алгоритма может использоваться для доказательств. Едва ли можно применить машину Тьюринга для решения практических задач, но эта простая модель алгоритма очень удобна для проведения теоретических исследований. В отличие от интуитивного определения алгоритма новое определение не содержит таких неопределённых понятий, как «инструкция», «исполнитель», «решение задачи». Таким образом, формальное определение слова «алгоритм» (по Тьюрингу) выглядит так: алгоритм — это программа для машины Тьюринга.

Машина Поста

Практически одновременно с Тьюрингом (в том же 1936 г.) и независимо от него американский математик Э. Л. Пост предложил ещё более простую систему обработки данных, на основе которой позднее была построена так называемая **машина Поста**.

Лента в машине Поста (так же как и в машине Тьюринга) бесконечна и разбита на ячейки. Каждая ячейка может содержать метку (быть отмечена) или не содержать её (пустая ячейка) (рис. 5.6).

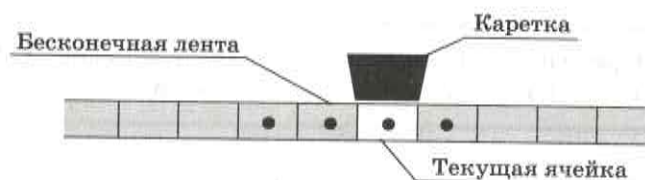


Рис. 5.6

Таким образом, Пост сократил алфавит всего до двух цифр. Это допустимо, потому что любые данные можно перекодировать в двоичный код, сопоставив каждой букве исходного алфавита уникальную последовательность нулей и единиц.

Кроме того, алгоритм работы машины Поста задаётся не в виде таблицы, а как программа, состоящая из отдельных команд. Система команд машины Поста содержит только 6 команд:

- ← — переместить каретку на 1 ячейку влево;
- — переместить каретку на 1 ячейку вправо;
- 0 — стереть метку в рабочей ячейке (записать 0);



Э. Л. Пост
(1897–1954)

- 1 — поставить метку в рабочей ячейке (записать 1);
- ? n_0, n_1 — если в рабочей ячейке нет метки, перейти к строке n_0 , иначе перейти к строке n_1 ;
- стоп — остановить машину.

Попытка стереть метку там, где её нет, или поставить метку повторно считается ошибкой, и машина аварийно останавливается.

Все строки в программе нумеруются по порядку, это необходимо для работы команды ветвления (? n_0, n_1). С помощью этой команды можно также строить циклы как с предусловием, так и с постусловием. Например, следующая программа перемещает каретку влево до первой отмеченной ячейки:

- 1. ←
- 2. ? 1, 3
- 3. стоп

Если после выполнения команды ←, →, 0 или 1 требуется перейти не на следующую строку, а на какую-то другую, то номер этой строки можно записать в конце команды. Например, команда

← 3

означает «переместить каретку влево и перейти на строку 3».

При работе с машиной Поста числа обычно записывают в унарной (единичной) системе счисления, в виде непрерывной цепочки меток нужной длины (вспомните счётные палочки в младшей школе). Например, на ленте, показанной на рис. 5.6, записано число 4.

Пост предположил, что любой алгоритм может быть записан как программа для предложенного им исполнителя. В теории алгоритмов доказано, что машины Поста и Тьюринга одинаковы по своим возможностям. Это значит, что круг задач, который они решают, тоже одинаков.

Нормальные алгорифмы Маркова

Советский математик А. А. Марков, который в середине XX века изучал разрешимость некоторых задач алгебры, предложил новую модель вычислений, которую он назвал нормальными алгорифмами.

Нормальные алгорифмы Маркова (НАМ) — это строгая математическая форма записи алгорит-



А. А. Марков
(младший)
(1903–1979)

мов обработки символьных строк, которую можно использовать для доказательства разрешимости или неразрешимости различных задач. Марков предположил, что любой алгоритм можно записать как НАМ. В отличие от машин Тьюринга и Поста, НАМ — это «чистый» алгоритм, который не связан ни с каким «аппаратным обеспечением» (лентой, кареткой и т. п.).

НАМ преобразует одно слово (цепочку символов некоторого алфавита) в другое и задаётся алфавитом и системой подстановок. Заметьте, что в жизни мы нередко применяем такие замены. Например, при умножении в столбик мы не вычисляем каждый раз произведение $7 \cdot 8$, а просто помним, что оно равно 56.

Пример 1. Пусть алфавит НАМ — это русские буквы, и задана система подстановок:

$a \rightarrow н$
 $ух \rightarrow ло$
 $м \rightarrow с$

Применим эту систему подстановок к начальному слову «муха». Подстановки нужно просматривать по порядку, начиная с первой. Первая подстановка означает: «если в слове есть буквы «а», заменить первую букву «а» на букву «н»». В слове «муха» есть буква «а», поэтому заменяем её на «н». Получается «мухн».

Начинаем просмотр подстановок сначала. Букв «а» больше нет, поэтому переходим ко второй подстановке. Сочетание «ух» есть в слове «мухн», поэтому вторая подстановка срабатывает, и мы заменяем «ух» на «ло»: получается «млон».

Теперь ни первая, ни вторая подстановки не применимы, а использование третьей даёт в результате слово «слон». Больше ни одну подстановку сделать нельзя, и НАМ заканчивает работу. Таким образом, приведённая система подстановок преобразует слово «муха» в слово «слон».

При поиске образца рабочая цепочка символов просматривается с начала. Если в строке слово-образец встречается несколько раз, то за один шаг заменяется только первое из них. Так как на следующем шаге просмотр опять начинается с начала цепочки, после первой выполненной замены может «сработать» совсем другая подстановка.

В записи подстановок слово-образец может быть пустым, в этом случае слово-замена приписывается в начало рабочей строки:

$\rightarrow 0$

Такая подстановка всегда должна быть последней в списке, иначе программа заикнется: в начало слова будут постоянно дописываться всё новые и новые нули.

Если после слова-замены стоит точка, после выполнения такой подстановки работа программы заканчивается. Например, если применить НАМ

$a \rightarrow 0.$

к слову «карова», то в результате получим «корова», потому что после первого же действия работа программы закончится, и последняя буква не будет заменена.

Пример 2. Построим НАМ для следующей задачи: удалить из строки, состоящей из букв «а» и «b», первый символ. Например, строка «abba» должна быть преобразована в «bba». Казалось бы, здесь нужно использовать систему подстановок:

$a \rightarrow .$
 $b \rightarrow .$

Однако такой НАМ будет неправильно работать для слов, начинающихся с буквы «b», например для слова «bba», в котором будет удалена последняя буква, потому что первая подстановка выполняется раньше, чем вторая. Перестановка двух строк также не даёт решения — теперь алгоритм неправильно работает для слов, начинающихся с буквы «а». Чтобы решить эту задачу, в алфавит НАМ добавляем ещё один специальный символ, например символ «*». Этим символом помечают начало слова, используя подстановку.

$\rightarrow *$

Полный алгоритм выглядит так:

$*a \rightarrow .$
 $*b \rightarrow .$
 $\rightarrow *$

Сначала срабатывает третья подстановка (ставим «*» в начало строки), затем, в зависимости от первой буквы исходного слова, работает первая или вторая подстановка, и алгоритм заканчивает работу. Дополнительный символ похож на маркер в текстовом редакторе — он отмечает место в тексте, с которым потом будут выполняться какие-то действия.

Как показано в теории алгоритмов, любой алгоритм для машин Тьюринга и Поста можно записать как НАМ, и наоборот.

Поэтому все три рассмотренных подхода к строгому определению понятия «алгоритм» эквивалентны (равносильны).



Вопросы и задания

1. Зачем понадобилось уточнять понятие «алгоритм»?
2. Какие задачи рассматриваются в теории алгоритмов?
3. Почему можно ограничиться алгоритмами обработки символьных строк? Можно ли рассматривать только алгоритмы для преобразования двоичных кодов?
4. Как вы понимаете утверждение «Алгоритм задаёт некоторую функцию»?
5. Как связаны понятия «алгоритм» и «исполнитель»?
6. Что такое программа?
7. В каком случае говорят, что два алгоритма эквивалентны?
8. Что такое универсальный исполнитель?
9. Сравните интуитивное и строгое понятия алгоритма.
10. Опишите устройство и систему программирования машины Тьюринга.
11. Что такое состояние машины Тьюринга?
12. Сопоставьте устройство машины Тьюринга с устройством компьютера. Какие устройства машины Тьюринга выполняют те же функции, что и аналогичные устройства компьютера?
13. В чем особенность состояний q_0 и q_1 машины Тьюринга?
14. Как можно построить программу для машины Тьюринга, которая последовательно выполняет операции А и Б, если уже есть две программы, которые выполняют эти две операции по отдельности?
15. Сформулируйте тезис Чёрча–Тьюринга.
16. Сравните машины Тьюринга и Поста.
17. Зачем нумеруются строки в программе для машины Поста?
18. Что такое нормальный алгоритм Маркова?
19. Зачем используют специальные символы в НАМ?
20. Что означает эквивалентность различных универсальных исполнителей?



Подготовьте сообщение

- а) «Какие бывают машины Тьюринга?»
- б) «Эзотерические языки программирования»
- в) «Рекурсивные функции»



Задачи

1. Что делают следующие программы для машины Тьюринга?

а)

	q_1
0	←
1	←
□	→ q_0

б)

	q_1
0	→ q_0
1	→ q_0
□	←

в)

	q_1	q_2
а	q_2	□ ←
б	q_2	□ ←
□	←	q_0

В каких случаях эти программы закливаются?

2. Предложите программу для машины Тьюринга и начальное состояние ленты, при котором эта программа закливается.
3. Составьте программу для машины Тьюринга, которая уменьшает двоичное число на 1.
4. Составьте программы для машины Тьюринга, которые увеличивают и уменьшают на единицу число, записанное в десятичной системе счисления.
5. Составьте программу для машины Тьюринга, которая складывает два числа в двоичной системе, разделенные на ленте знаком «+».
6. Составьте программы для машины Тьюринга, которые выполняют сложение и вычитание двух чисел в десятичной системе счисления.
7. Что делают следующие программы для машины Поста?

а)

1. 1
2. →
3. → 1

б)

1. →
2. ? 3,4
3. 1 1
4. стоп

в)

1. ? 2,3
2. 1 4
3. → 1
4. стоп

Как будет работать каждая из программ при различных начальных состояниях ленты?

8. Напишите программу для машины Поста, которая увеличивает (уменьшает) число в единичной системе счисления на единицу. Каретка расположена слева от числа.
9. Напишите программу для машины Поста, которая складывает два числа в единичной системе счисления. Каретка расположена над единственным пробелом, разделяющим эти числа на ленте.
10. Что делают следующие НАМ, если применить их к символьной цепочке, состоящей из нулей и единиц?

а)

0 → 00
1 → 11

б)

0 → 0
1 → 1
* → =,
→ *

в)

0 → 00
1 → 11
* → ,
→ *

11. Напишите НАМ, который сортирует цифры двоичного числа так, чтобы сначала стояли все нули, а потом — все единицы.
12. Напишите НАМ, который удаляет последний символ строки, состоящей из цифр 0 и 1. Какую операцию он выполняет, если рассматривать строку как двоичную запись числа?
13. Напишите НАМ, который умножает двоичное число на 2, добавляя 0 в конец записи числа.

§ 35

Алгоритмически неразрешимые задачи

Вычислимые и невычислимые функции

Мы уже говорили, что любой алгоритм определяет некоторую функцию, которая для каждого допустимого входного слова однозначно задаёт результат — выходное слово. Такие функции называются вычислимыми.

Вычислимая функция — это функция, для вычисления которой существует алгоритм.

Любая вычислимая функция может задаваться разными алгоритмами (разными программами для выбранного универсального исполнителя). Например, следующие два нормальных алгоритма Маркова решают одну и ту же задачу — заменяют во входном двоичном слове все буквы «а» на нули и все буквы «б» на единицы:

$a \rightarrow 0$ $b \rightarrow 1$
 $b \rightarrow 1$ $a \rightarrow 0$

Любая вычислимая функция может быть вычислена с помощью любого универсального исполнителя: машин Тьюринга и Поста, нормальных алгоритмов Маркова и др.

Рассмотрим, например, такую функцию, определённую для всех натуральных чисел:

$$f(n) = \begin{cases} 1, & \text{если } n \text{ — чётное;} \\ 0, & \text{если } n \text{ — нечётное.} \end{cases}$$

Попробуем составить программу для машины Тьюринга, которая вычисляет эту функцию. Будем считать, что число записано в единичной системе счисления (в виде цепочки единиц¹), и каретка в начальный момент стоит над самой левой единицей. Оказывается, такая программа действительно существует (рис. 5.7).

	q_1	q_2	q_3	q_4
1	$\rightarrow q_2$	$\rightarrow q_1$	$\leftarrow q_4$	$\square \leftarrow$
$\square$	$\leftarrow q_3$	$\leftarrow q_4$		q_0

Рис. 5.7

Как принято, в начальный момент машина находится в состоянии q_1 . Затем она движется вправо вдоль числа, поочередно переходя из состояния q_1 (пройдено чётное число единиц) в состояние q_2 (пройдено нечётное число единиц) и обратно. Таким образом, если встречен пробел и машина находится в состоянии q_2 , то число нечётное и нужно просто стереть все единицы (состояние q_4). Если машина закончила просмотр в состоянии q_1 , то число чётное; при этом нужно оставить одну единицу (состояние q_3) и перейти в состояние q_4 (стереть все остальные единицы). Обратите внимание, что ячейка ($\square, q_3$) в таблице пустая — это невозможное состояние (покажите это самостоятельно).

Таким образом, рассмотренная функция вычислима, т. е. её можно вычислять с помощью машины Тьюринга, а значит, и с помощью любого универсального исполнителя. Например, нормальный алгоритм Маркова для алфавита $A = \{1\}$ выглядит так:

$11 \rightarrow ""$
 $1 \rightarrow .$
 $\rightarrow 1.$

В первой подстановке две соседние единицы удаляются (слово-замена здесь пустое, для ясности оно взято в кавычки, которыми можно ограничивать слова в НАМ). Это происходит до тех пор, пока не будут удалены все пары, поскольку эта подстановка стоит первой. Если остаётся одна единица, она удаляется с помощью второй подстановки, и работа программы заканчивается. Если все единицы удалены (число чётное), то с помощью третьей подстановки мы ставим одну единицу и останавливаем автомат.

¹ Пустая лента соответствует числу 0.

Существуют и **невыхислимые функции**. Рассмотрим простой пример, предложенный В. А. Успенским в книге «Машина Поста». Известно, что математическая постоянная π — иррациональное число, его десятичная запись бесконечна и непериодична. Введем функцию $h(n)$, которая для любого натурального числа n равна 1, если в десятичной записи числа π есть n стоящих подряд девяток, окружённых другими цифрами, и равна нулю, если такой цепочки девяток нет. Как вычислить значение этой функции при некотором заданном n ? Конечно, можно вычислять друг за другом десятичные знаки числа π (такие алгоритмы математикам известны!) и проверять, не нашлась ли в полученной последовательности цифр цепочка из n девяток. С помощью такого «наивного» алгоритма можно найти такие значения n , при которых $h(n) = 1$: обнаружив требуемую цепочку, алгоритм закончит работу. Например, анализ первых 800 знаков показывает, что $h(n) = 1$ при $n = 1, 2, 6$. Но если для какого-то n функция $h(n)$ равна нулю, то «наивный» алгоритм никогда не остановится. Более того, для этой функции вообще не существует алгоритма, который при любом n останавливается и выдает значение $h(n)$ в качестве результата. Поэтому такая функция невычислима.

Когда задача алгоритмически неразрешима?

Как вы знаете, невозможно создать вечный двигатель, потому что это противоречит универсальным физическим законам сохранения. Точно так же в математике и информатике существуют задачи, для которых решение в общем виде отсутствует.

Поскольку алгоритм работает только с дискретными объектами, любая алгоритмическая задача — это функция, заданная на множестве дискретных объектов (входных слов).

Пусть, например, требуется по шахматной позиции определить, кто выигрывает при правильной игре — белые, чёрные или будет ничья. Построим функцию, соответствующую этому алгоритму. Для этого выберем способ кодирования, при котором каждая позиция может быть закодирована словом (символьной строкой) v в подходящем алфавите. Тогда приведённой задаче может соответствовать функция $f(v)$, заданная на множестве таких слов:

$$f(v) = \begin{cases} \text{'Б'}, & \text{если } v \text{ — код позиции, в которой выигрывают белые,} \\ \text{'Ч'}, & \text{если } v \text{ — код позиции, в которой выигрывают чёрные,} \\ \text{'0'}, & \text{если } v \text{ — код позиции, в которой будет ничья,} \\ \text{'?'}, & \text{если } v \text{ — ошибочный код позиции.} \end{cases}$$

Если функция, соответствующая задаче, вычислима, то задача называется **алгоритмически разрешимой** — для её вычисления можно построить алгоритм. Если определённая в задаче функция невычислима, то алгоритма для её решения не существует.

Алгоритмически неразрешимая задача — это задача, соответствующая невычислимой функции.

В 1900 г. на Международном математическом конгрессе в Париже известный математик Давид Гильберт сформулировал 23 нерешённые математические проблемы¹. В знаменитой «десятой проблеме Гильберта» требуется найти метод, который позволяет определить, имеет ли заданное алгебраическое уравнение с целыми коэффициентами решение в целых числах. Например, уравнение

$$x^2 + y^3 + 2 = 0$$

имеет два целочисленных решения, $(5; -3)$ и $(-5; -3)$. Сложность состояла в том, что требовалось найти единый метод (алгоритм), позволяющий решить задачу для *любого* такого уравнения со многими неизвестными.

В начале XX века была уверенность, что такой алгоритм есть, и поэтому его упорно искали. Однако в 1970 г. советскому математику Ю. В. Матиясевичу удалось доказать, что общего алгоритма решения этой задачи не существует.



Ю. В. Матиясевич
(род. в 1947)

Немецкий математик Г. В. Лейбниц в XVII веке безуспешно пытался найти метод проверки правильности любых математических утверждений. Как вы знаете, почти все математические теории основаны на использовании *аксиом* (положений, принимаемых без доказательства), из которых выводятся все остальные утверждения (*теоремы*). Задача заключалась в том, чтобы разработать алгоритм, позволяющий установить, можно ли вывести формулу Б из формулы А в рамках заданной системы аксиом

¹ Сейчас большинство из них решено полностью или частично.

(«проблема распознавания выводимости»). В 1936 г. американский математик А. Чёрч доказал, что эта задача *в общем виде* алгоритмически неразрешима, поэтому нельзя сформулировать универсальный алгоритм, пригодный для доказательства любой теоремы¹.

Таким образом, уточнённые определения алгоритма, основанные на понятии универсальных исполнителей, сыграли в науке очень важную роль — позволили получить *отрицательные результаты*, т. е. доказать, что алгоритмов решения некоторых задач в общем виде не существует.

Для того чтобы доказать неразрешимость какой-то новой задачи, пытаются свести её к уже известным алгоритмически неразрешимым задачам. Если это удастся, значит, и новая задача алгоритмически неразрешима².

Существуют также задачи, про которые неизвестно, алгоритмически разрешимы они или нет — решение не найдено, но алгоритмическая неразрешимость не доказана.

Алгоритмически неразрешимые задачи встречаются не только в математике, но и в информатике, например при разработке программ. Оказывается, невозможно написать программу для машины Тьюринга (алгоритм), которая по тексту любой программы P и её входным данным X определяет, завершается ли программа P при входе X за конечное число шагов или закидывается. Это так называемая **проблема останова**. Её неразрешимость означает, в частности, что нельзя полностью автоматизировать тестирование любых программ, поручив это компьютеру. Однако для некоторых классов алгоритмов проблему останова решить можно. Например, линейная программа, не содержащая ветвлений и циклов, всегда завершится.

Было доказано, что алгоритмически неразрешима **проблема эквивалентности**: по двум заданным алгоритмам определить, будут ли они выдавать одинаковые результаты для любых допустимых исходных данных. Следовательно, невозможно полностью



А. Чёрч
(1903–1995)

¹ Тем не менее отдельные классы теорем можно доказывать на компьютере.

² Допустим, что (1) задача A неразрешима и (2) если мы можем построить алгоритм для решения задачи B , то с его помощью можно построить алгоритм решения задачи A . Тогда задача B тоже неразрешима.

автоматизировать решение многих важных задач, связанных с разработкой программ, например:

- по заданному тексту программы определить, что она «делает»;
- определить, правильно ли работает программа при любых допустимых исходных данных;
- найти ошибку в программе, работающей неправильно.

Поэтому при отладке программы большую роль играет интуиция. Помогают (но не решают проблему полностью!) стандартные приёмы, позволяющие найти ошибку:

- сравнение результатов работы программы с результатами ручного счёта;
- эксперименты с программой при различных исходных данных для того, чтобы выявить закономерность появления ошибок;
- временное отключение (комментирование) частей программы и др.

Поскольку многие этапы разработки программного обеспечения в принципе невозможно представить в виде алгоритмов, программирование остаётся работой человека. Полностью поручить его компьютеру не удастся, хотя решение некоторых конкретных задач всё же можно автоматизировать.

Вопросы и задания

1. Что такое вычислимая функция?
2. Приведите пример невычислимой функции.
3. Что такое алгоритмически неразрешимые задачи? Приведите известные вам примеры.
4. Что такое проблема останова? Каковы её следствия?
5. Что такое проблема эквивалентности?
6. Как можно доказать алгоритмическую неразрешимость новой задачи?

Задачи

1. В качестве доказательства того, что следующая функция вычислима, напишите программу для машины Поста.

$$f(n) = \begin{cases} 1, & \text{если } n \text{ — чётное;} \\ 0, & \text{если } n \text{ — нечётное.} \end{cases}$$

2. Докажите, что следующая функция вычислима:

$$f(n) = \begin{cases} 1, & \text{если } n \text{ делится на } 3; \\ 0, & \text{если } n \text{ не делится на } 3. \end{cases}$$

В качестве доказательства напишите программы для машин Тьюринга и Поста, а также НАМ.

*3. Первой задачей, неразрешимость которой была доказана, была **проблема самоприменимости**: по заданному тексту программы P определить, останавливается ли программа P , если ей на вход подать текст этой же программы. Докажите, что проблема останова сводится к проблеме самоприменимости (именно так и была доказана неразрешимость проблемы останова).

§ 36

Сложность вычислений

Что такое сложность вычислений?

Центральная задача теории алгоритмов — выяснить, существует ли алгоритм решения той или иной задачи. Если существует, то возникает следующий вопрос: а можно ли им воспользоваться на практике, при современном уровне развития вычислительной техники? То есть способен ли компьютер за приемлемое время получить результат? Например, в игре в шахматы возможно лишь конечное количество позиций и, значит, только конечное количество различных партий. Следовательно, теоретически можно перебрать все возможные партии и выяснить, кто побеждает при правильной игре — белые или чёрные. Однако количество вариантов настолько велико, что современные компьютеры не могут выполнить такой перебор за приемлемое время.

Что мы хотим от алгоритма? Во-первых, чтобы он работал как можно быстрее. Во-вторых, чтобы объём необходимой памяти был как можно меньше. В-третьих, чтобы он был как можно более прост и понятен, что позволяет легче отлаживать программу. К сожалению, эти требования противоречивы, и в серьёзных задачах редко удаётся найти алгоритм, который был бы лучше остальных по всем показателям.

Часто говорят о *временной сложности* алгоритма (*быстродействии*) и *пространственной сложности*, которая определяется объёмом необходимой памяти. Поскольку память постоянно дешевеет, а быстродействие компьютеров растёт медленно, мы будем рассматривать главным образом временную сложность — время выполнения программы, работающей по данному алгоритму.

В общем случае, говоря о сложности алгоритма, нужно уточнить, о каком исполнителе идёт речь, какие элементарные операции мы используем. Как правило, это один из универсальных исполнителей (во многих случаях универсальным исполнителем можно считать компьютер). Временем работы алгоритма называется количество выполненных им элементарных операций T . Такой подход позволяет оценивать именно качество алгоритма, а не свойства исполнителя (например, быстродействие компьютера, на котором выполняется алгоритм).

Как правило, величина T будет существенно зависеть от объёма исходных данных: поиск в списке из 10 элементов завершится гораздо быстрее, чем в списке из 10 000 элементов. Поэтому сложность алгоритма обычно связывают с размером входных данных n и определяют как функцию $T(n)$. Например, для алгоритмов обработки массивов в качестве размера n используют длину массива. Функция $T(n)$ называется **временной сложностью алгоритма**.

Примеры

Рассмотрим алгоритмы выполнения различных операций с массивом A длины n , который может быть объявлен в программе на школьном алгоритмическом языке как

целтаб $A[1:n]$

Пример 1. Требуется вычислить сумму первых трёх элементов массива (при $n \geq 3$).

Решение этой задачи содержит всего один оператор:

$Sum := A[1] + A[2] + A[3]$

Этот алгоритм включает две операции сложения и одну операцию записи значения в память, поэтому его сложность $T(n) = 3$ не зависит от размера массива вообще.

Пример 2. Требуется вычислить сумму всех элементов массива. В этой задаче уже не обойтись без цикла:

```
Sum:=0
нц для i от 1 до n
    Sum:=Sum+A[i]
кц
```

Здесь выполняется n операций сложения и $n + 1$ операций записи в память¹, поэтому его сложность $T(n) = 2n + 1$ возрастает линейно с увеличением длины массива².

Пример 3. Требуется отсортировать все элементы массива по возрастанию методом выбора.

Напомним, что метод выбора предполагает поиск на каждом шаге минимального из оставшихся неупорядоченных значений (здесь $i, j, nMin$ и c — целочисленные переменные):

```
нц для i от 1 до n-1
    nMin:=i;
    нц для j от i+1 до n
        если A[j]<A[nMin] то nMin:=j все
    кц
    если nMin<>i то
        c:=A[i]; A[i]:=A[nMin]; A[nMin]:=c
    все
кц
```

Подсчитаем отдельно количество сравнений $T_c(n)$ и количество перестановок $T_p(n)$. Количество сравнений элементов массива не зависит от данных и определяется числом шагов внутреннего цикла:

$$T_c(n) = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n.$$

Число перестановок зависит от данных. Например, если массив уже отсортирован в нужном порядке, перестановок не будет вообще. В худшем случае на каждом шаге основного цикла происходит перестановка, всего их будет $T_p(n) = n - 1$.

¹ Здесь и далее для упрощения выводов мы не учитываем команды, необходимые для организации цикла, потому что при больших n время их выполнения очень мало в сравнении со временем выполнения остальных операторов.

² Предполагается, что сложение любых двух чисел выполняется одинаковое время.

Что такое асимптотическая сложность?

Допустим, что нужно выбрать между несколькими алгоритмами, которые имеют разную сложность. Какой из них лучше (работает быстрее)? Оказывается, для этого необходимо знать размер массива данных, которые нужно обрабатывать. Сравним, например, три алгоритма, сложность которых

$$T_1(n) = 10\,000 \cdot n, \quad T_2(n) = 100 \cdot n^2, \quad T_3(n) = n^3.$$

Построим эти зависимости на графике (рис. 5.8). При $n \leq 100$ получаем $T_3(n) < T_2(n) < T_1(n)$, при $n = 100$ количество операций для всех трёх алгоритмов совпадает, а при больших n имеем $T_3(n) > T_2(n) > T_1(n)$.

Обычно в теоретической информатике при сравнении алгоритмов используется их асимптотическая сложность, т. е. скорость роста количества операций при больших значениях n . При этом запись $O(n)$ (читается «О большое от n ») обозначает, что, начиная с некоторого значения $n = n_0$, количество операций ограничено функцией $c \cdot n$, где c — некоторая константа:

$$T(n) \leq c \cdot n \text{ для } n \geq n_0.$$

Такие алгоритмы имеют линейную сложность, т. е. при увеличении размера данных в 10 раз количество операций увеличивается тоже примерно в 10 раз.

Пусть, например, $T(n) = 2n - 1$, как в алгоритме поиска суммы элементов массива. Очевидно, что при этом $T(n) \leq 2n$ для всех $n \geq 1$, поэтому алгоритм имеет линейную сложность.

Многие известные алгоритмы имеют квадратичную сложность, которая обозначается как $O(n^2)$. Это значит, что сложность алгоритма ограничена функцией $c \cdot n^2$:

$$T(n) \leq c \cdot n^2 \text{ для } n \geq n_0.$$

При этом если размер данных увеличивается в 10 раз, то количество операций (и время выполнения) увеличивается примерно

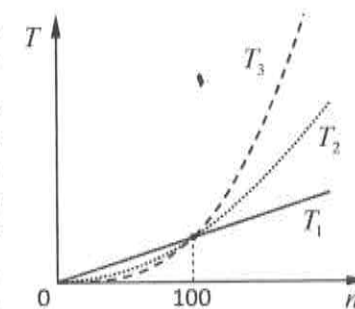


Рис. 5.8

но в 100 раз. Пример такого алгоритма — сортировка методом прямого выбора, для которой число сравнений

$$T_c(n) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{для всех } n \geq 0.$$

Алгоритм имеет **асимптотическую сложность** $O(f(n))$, если найдётся такая постоянная c , что для всех $n \geq n_0$ выполняется условие $T(n) \leq c \cdot f(n)$.

Это значит, что при $n \geq n_0$ график функции $c \cdot f(n)$ идёт выше, чем график функции $T(n)$ (рис. 5.9).

Если количество операций не зависит от размера данных, то говорят, что сложность алгоритма $O(1)$, т. е. количество операций меньше некоторой постоянной при любых n .

Существует также немало алгоритмов с кубической сложностью — $O(n^3)$. При больших значениях n алгоритм с кубической сложностью требует большего количества вычислений, чем алгоритм со сложностью $O(n^2)$, а тот, в свою очередь, работает дольше, чем алгоритм с линейной сложностью. Заметьте, что при малых значениях n всё может быть наоборот; это зависит от постоянной c для каждого из алгоритмов.

Известны и алгоритмы, для которых количество операций растёт быстрее, чем любой полином, например как $O(2^n)$ или $O(n!)$. Они встречаются чаще всего в задачах оптимизации, которые решаются только методом полного перебора. Самая известная задача такого типа — это **задача коммивояжёра** (бродячего торговца), который должен посетить по одному разу каждый из указанных городов и вернуться в начальную точку. Для него нужно выбрать оптимальный маршрут, при котором стоимость поездки (или общая длина пути) будет минимальной.

Ещё один пример сложной задачи, которая решается только полным перебором всех вариантов, — **задача выполнимости**. Дано логическое выражение, которое содержит только имена логических переменных, скобки, а также операции «И», «ИЛИ» и «НЕ». Требуется определить, существует ли набор значений логических переменных, при котором заданное выражение истинно.

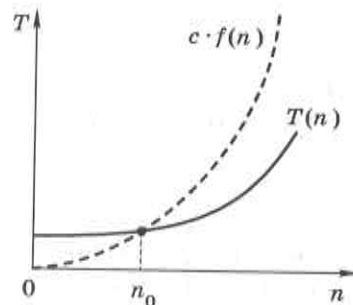


Рис. 5.9

В следующей таблице показано примерное время выполнения алгоритмов, имеющих различную временную сложность $T(n)$, при $n = 100$ на компьютере с быстродействием 10^9 операций в секунду.

$T(n)$	Время выполнения
n	100 нс
n^2	10 мс
n^3	0,001 с
2^n	10^{13} лет
$n!$	10^{141} лет

Алгоритмы поиска

Сравним вычислительную сложность двух наиболее известных алгоритмов поиска.

Пример 4 (линейный поиск). Дан массив, в котором элементы расположены в произвольном порядке. Требуется найти в нём заданное значение X или сообщить, что его нет.

Решение этой задачи сводится к последовательному просмотру всех элементов массива:

```

nX:=0
нц для i от 1 до n
  если A[i]=X то
    nX:=i
  выход
все
кц
если nX>0 то
  вывод "A[", nX, "]= ", X
иначе
  вывод "Элемент не найден"
все
  
```

В этом алгоритме число сравнений (в худшем случае) равно $T(n) = n$, поэтому он имеет линейную сложность.

Пример 5 (двоичный поиск). Дан массив, в котором элементы упорядочены по возрастанию. Требуется найти в нём заданное значение X или сообщить, что его нет.

По сравнению с предыдущей задачей, элементы массива отсортированы, и это ускоряет решение, потому что можно применить метод двоичного поиска (дихотомии):

```

L:=1; R:=n+1
нц пока L<R-1
  c:=div(L+R, 2) | или c:=L+div(R-L, 2)
  если X<A[c] то
    R:=c
  иначе
    L:=c
все
кц
если A[L]=X то
  вывод "A[" , L, "]=", X
иначе
  вывод "Элемент не найден"
все

```

Попробуем определить, сколько раз выполняется основной цикл при двоичном поиске. Сначала ширина интервала поиска — все n элементов массива. На каждом шаге этот интервал делится на 2, процесс завершается, когда левая и правая границы интервала совпадут. Предположим, что число элементов — это целая степень двойки, т. е. $n = 2^m$. Тогда за m шагов ширина интервала сужается до 1, а на следующем шаге его границы совпадут, и цикл закончится. Таким образом, количество шагов цикла равно $m + 1$. Из равенства $n = 2^m$ получаем $m = \log_2 n$, так что

$$T(n) = \log_2 n + 1.$$

Например, при $n = 2^{16}$ линейный поиск потребует в худшем случае $2^{16} = 65\,536$ сравнений, а двоичный — всего $16 + 1 = 17$ сравнений.

Таким образом, алгоритм двоичного поиска имеет асимптотическую сложность $O(\log n)$. Основание логарифма обычно не указывают, потому что выражения $\log_a n$ и $\log_b n$ различаются на постоянный множитель (который можно включить в постоянную c):

$$\log_a n = \frac{1}{\log_b a} \cdot \log_b n.$$

Можно ли сказать, что алгоритм двоичного поиска лучше алгоритма линейного поиска? Нет! Ведь алгоритм линейного поиска применим к любым массивам данных, а алгоритм двоичного

поиска — только к упорядоченным (отсортированным). А если мы сначала отсортируем массив, а потом применим к нему двоичный поиск, то общее время работы будет больше, чем при линейном поиске.

Ситуация меняется, если нам нужно многократно выполнять операцию поиска для одних и тех же данных (так, как правило, бывает при работе с базами данных). Тогда имеет смысл заранее отсортировать массив (применить предварительную обработку данных), а затем, используя двоичный поиск, экономить время при каждом новом поисковом запросе.

Алгоритмы сортировки

Ранее мы проанализировали один из простых методов сортировки массивов — метод прямого выбора и выяснили, что его асимптотическая сложность $O(n^2)$. Повторим анализ для метода пузырька, который также изучался в 10 классе:

```

нц для i от 1 до n-1
  нц для j от n-1 до i шаг -1
    если A[j]>A[j+1] то
      c:= A[j]; A[j]:= A[j+1]; A[j+1]:= c;
    все
  кц
кц

```

На первом шаге основного цикла выполняется $n - 1$ шагов внутреннего цикла, т. е. $n - 1$ сравнений. Далее количество сравнений уменьшается до 1, так что общее количество сравнений равно:

$$T_c(n) = (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n,$$

так же как и у алгоритма прямого выбора. В то же время в худшем случае при каждом сравнении выполняется перестановка, что требует

$$T_p(n) = 3 \cdot \frac{n(n-1)}{2} = \frac{3}{2}n^2 - \frac{3}{2}n$$

операций присваивания. Таким образом, этот алгоритм имеет асимптотическую сложность $O(n^2)$ как по числу сравнений, так и по числу присваиваний.

Существуют ли более эффективные сортировки, имеющие, например, линейную сложность? Да, для некоторых особых случаев существуют. Например, если известно, что все значения исходно-

го массива находятся в интервале от 1 до некоторого значения MAX , можно использовать **сортировку подсчётом**. Для этого выделяется дополнительный массив счётчиков:

целтаб $C[1:MAX]$

который предварительно обнуляется:

```
нц для i от 1 до MAX
  C[i] := 0
кц
```

Затем в цикле проходим весь массив с данными и для каждого элемента $A[i]$ увеличиваем счётчик $C[A[i]]$. Например, если $A[i]=20$, счётчик $C[20]$ увеличивается на 1. После окончания цикла в каждом счётчике $C[i]$ находится количество значений исходного массива, равных i .

```
нц для i от 1 до n
  C[A[i]] := C[A[i]] + 1
кц
```

Теперь остаётся расставить числа в массиве A в нужном количестве. Например, если $C[20] = 5$, в массив A записываются последовательно 5 значений, равных 20:

```
k := 1
нц для i от 1 до MAX
  нц для j от 1 до C[i]
    A[k] := i
    k := k + 1
  кц
кц
```

Попробуем подсчитать количество операций для этого алгоритма. Заполнение массива C нулями требует MAX присваиваний. Цикл подсчёта элементов содержит n сложений и присваиваний, т. е. его сложность — линейная, $O(n)$. Наконец, последний вложенный цикл выполняет также n сложений и присваиваний (по числу элементов массива A), поэтому алгоритм в целом имеет линейную сложность по n .

Однако нужно учитывать, что принципиальное ускорение алгоритма в сравнении с предыдущими получено за счёт того, что:

- все значения — целые числа в ограниченном диапазоне;
- есть возможность использовать дополнительный массив размером MAX , который может значительно превышать размер исходного массива.

Здесь проявляется компромисс «скорость — память», который присутствует во многих задачах: ускорение алгоритма возможно за счёт использования дополнительной памяти и наоборот, экономия памяти приводит к замедлению работы алгоритма.

Доказано, что в общем случае вычислительная сложность сортировки, основанной только на использовании операций «сравнить» и «переставить», не может быть меньше, чем $O(n \log n)$. Именно такую сложность имеют, например, **сортировка слиянием** (англ. *merge sort*) и **пирамидальная сортировка** (англ. *heap sort*), которые применяются при работе с большими наборами данных. **Быстрая сортировка** (англ. *quick sort*), которая изучалась в 10 классе, в среднем тоже имеет сложность $O(n \log n)$, однако в худшем случае (когда на каждом шаге массив делится на две части, одна из которых состоит из одного элемента) требуется $O(n^2)$ обменов.

Вопросы и задания

1. Какие критерии используются для оценки качества алгоритмов?
2. Почему скорость работы алгоритма оценивается не временем выполнения, а количеством элементарных операций?
3. Как учитывается размер данных при оценке скорости алгоритма?
4. Что означают записи $O(1)$, $O(n)$, $O(n^2)$ и $O(2^n)$?
5. В каких случаях алгоритм, имеющий асимптотическую сложность $O(n^2)$, может работать быстрее, чем алгоритм с асимптотической сложностью $O(n)$?

Задачи

1. Оцените количество операций для алгоритмов:
 - а) поиска всех делителей числа;
 - б) нахождения минимального и максимального элементов массива;
 - в) определения количества положительных элементов массива;
 - г) поиска в квадратной матрице прямоугольной области с наибольшей суммой элементов;
 - д) поиска всех простых чисел среди элементов квадратной матрицы;
 - е) поиска в массиве пары элементов с наибольшим произведением;
 - ж) поиска в массиве самой длинной цепочки подряд идущих элементов с одинаковыми значениями;
 - з) вычисления суммы элементов квадратной матрицы;
 - и) поиска всех простых чисел в заданном интервале;
 - к) поиска всех чисел Фибоначчи среди элементов квадратной матрицы.
 В каждом случае опишите набор используемых элементарных операций. Определите асимптотическую сложность этих алгоритмов.

- *2. Предложите алгоритм, позволяющий найти и вывести на экран те символы, которые встречаются в строке более одного раза. Оцените его асимптотическую сложность.
- *3. Алфавит языка племени «тумба-юмба» содержит k символов. Предложите алгоритм построения всех возможных слов этого языка, имеющих длину n символов, и оцените его асимптотическую сложность.

§ 37

Доказательство правильности программ

Как доказать правильность программы?

Как правило, программист разрабатывает программу на заказ, и от него требуется не только написать код, но и убедиться, что код работает правильно, т. е. в соответствии с требованиями заказчика.

Очевидно, что если программа выдаёт неверный результат хотя бы для одного варианта входных данных, можно сразу сказать, что она некорректна, т. е. содержит ошибки.

Сложнее доказать правильность программы — убедиться, что она выдаёт верные результаты при любых допустимых входных данных. Программисты-практики для решения этой задачи используют **тестирование**: проверяют работу программы с помощью набора тестовых данных, для которых известен правильный результат. Если полученный результат не совпадает с заданным, выполняется **отладка** программы, т. е. поиск и исправление ошибок.

Однако, как писал нидерландский учёный, один из создателей современного программирования Эдсгер Дейкстра, «отладка может показать лишь наличие ошибок и никогда — их отсутствие». В результате можно гарантировать верную работу программы только при тех данных, которые использовались в контрольных тестах. Кроме того, неясно, как определить, что все ошибки выявлены и нужно завершить отладку.



Э. В. Дейкстра
(1930–2002)

Пример 1. Рассмотрим следующую программу для выбора максимального из трёх значений, записанных в переменных a , b и c :

```
если  $a > b$  то  $M := a$  иначе  $M := b$  все
если  $b > c$  то  $M := b$  иначе  $M := c$  все
```

Проверяя её на тестах

$(a, b, c) = (1, 2, 3), (1, 3, 2), (2, 1, 3)$ и $(2, 3, 1)$,

мы во всех этих случаях получаем в переменной M верный ответ 3. Однако это не означает, что программа правильная, так как существует контрпример $(3, 2, 1)$: для этого набора входных данных в переменной M в результате оказывается число 2.

Чтобы быть уверенными в том, что программа работает правильно при *любых* допустимых исходных данных, применяют методы **доказательного программирования**: для каждого блока программы составляют требования к входным и выходным данным и строго доказывают, что программа всегда работает верно.

К сожалению, доказывать правильность программ не так просто, и в таких доказательствах тоже возможны ошибки. Однако при этом автор должен глубоко разобраться в алгоритме и его «подводных камнях», и часто при этом обнаруживаются ошибки, которые могли бы проявиться уже после выпуска программы в свет.

На практике редко доказывают правильность всей программы в целом. В то же время очень полезно доказывать правильность отдельных блоков (циклов, процедур и функций) для уменьшения количества «необъяснимых» ошибок и сокращения времени отладки.

Покажем метод доказательства правильности программы на простом примере.

Пример 2. Требуется доказать, что после выполнения следующей программы значения переменных a и b меняются местами:

```
 $b := a + b$       | 1
 $a := b - a$       | 2
 $b := b - a$       | 3
```

Предполагается, что сумма исходных чисел не приводит к переполнению разрядной сетки. Для удобства операторы программы пронумерованы.

Обозначим начальные значения переменных a и b через a_0 и b_0 . После выполнения оператора 1 в переменной b будет записано значение $a_0 + b_0$. Оператор 2 записывает в переменную a значение

$$b - a = a_0 + b_0 - a_0 = b_0.$$

В результате выполнения оператора 3 получаем новое значение переменной b , равное

$$b - a = a_0 + b_0 - b_0 = a_0.$$

Таким образом, в результате выполнения программы переменные a и b будут равны b_0 и a_0 соответственно, что и требовалось доказать. Поэтому приведённая программа правильная.

Пример 3. Попробуем доказать или опровергнуть правильность уже встречавшейся ранее программы для выбора максимального из трёх значений, записанных в переменных a , b и c :

```
если a>b то M:=a иначе M:=b все      | 1
если b>c то M:=b иначе M:=c все      | 2
```

Анализируя строку 2, выясняем, что в ней значение переменной M всегда будет изменено, т. е. результат работы первой строки программы стирается, и

$$M = \begin{cases} b, & \text{если } b > c, \\ c, & \text{если } c \geq b. \end{cases}$$

Конечно, эта величина не совпадает с определением максимального значения из a , b и c . Таким образом, программа неправильная: она выдает неверное значение, если максимальное из трёх чисел хранилось в переменной a . Контрпример мы уже приводили: (3, 2, 1).

Алгоритм Евклида

Теперь докажем, что один из древнейших известных алгоритмов — алгоритм Евклида — действительно вычисляет наибольший общий делитель (НОД) двух натуральных чисел (мы рассматривали его в 10 классе).

Алгоритм Евклида. Пусть заданы два натуральных числа m и n , причём $m \geq n$. Для вычисления $\text{НОД}(m, n)$ следует многократно заменять большее число остатком от деления большего на меньшее до тех пор, пока меньшее число не станет равным нулю. Тогда оставшееся ненулевое число и есть $\text{НОД}(m, n)$.

Программа, основанная на алгоритме Евклида, может выглядеть, например, так (здесь a , b и r — целочисленные переменные):

```
a:=m; b:=n      | 1  НОД(a,b)=НОД(m,n)
нц пока b<>0    | 2
  r:=mod(a, b)   | 3
  a:=b; b:=r     | 4  НОД(a,b)=НОД(m,n)
кц              | 5
вывод a         | 6  НОД(a,b)=НОД(m,n), b=0
```

Докажем, что в результате этого алгоритма в переменной a находится $\text{НОД}(m, n)$.

В строке 1 исходные значения копируются из переменных m и n соответственно в переменные a и b . Очевидно, что при этом выполнено условие $\text{НОД}(a, b) = \text{НОД}(m, n)$.

На каждом шаге цикла (в строках 3–4) вычисляется остаток r от деления a на b и пара (a, b) заменяется на пару (b, r) . Какими свойствами обладают полученные значения b и r ?

Поскольку r — это остаток от деления a на b , до выполнения строк 3–4 было справедливо равенство $a = bp + r$, где p — некоторое целое число. Тогда, если a и b имеют общий делитель, то такой же делитель имеет и r . Следовательно, $\text{НОД}(b, r) = \text{НОД}(a, b) = \text{НОД}(m, n)$. Это значит, что условие $\text{НОД}(a, b) = \text{НОД}(m, n)$ по-прежнему выполняется после каждого шага цикла.

Поскольку остаток r с каждым шагом строго уменьшается, в конце концов он станет равным нулю и запишется в переменную b при выполнении строки 4. Цикл сразу же закончится, поскольку нарушится условие его выполнения. После завершения работы цикла условие $\text{НОД}(a, b) = \text{НОД}(m, n)$ по-прежнему выполняется, но, кроме того, $b = 0$. Отсюда следует, что $a = \text{НОД}(m, n)$.

Инвариант цикла

Таким образом, для алгоритма Евклида существует условие $\text{НОД}(a, b) = \text{НОД}(m, n)$, которое остаётся справедливым на протяжении всего выполнения алгоритма: перед началом цикла, после каждого шага цикла и после окончания работы цикла. Такое условие называется инвариантом цикла (англ. *invariant* — неизменный).

Инвариант цикла — это соотношение между значениями переменных, которое остаётся справедливым после завершения любого шага цикла.

Выделив в явном виде инвариант каждого цикла, мы избегаем многих возможных ошибок на начальной стадии и делаем первый шаг к доказательству правильности всей программы. Как писал академик Андрей Петрович Ершов, один из первых теоретиков программирования в СССР, «программиста бьют по рукам, если он посмеет написать оператор цикла, не найдя перед этим его инварианта».



А. П. Ершов
(1931–1988)

Рассмотрим несколько примеров.

Пример 1. Двое играют в следующую игру: перед ними лежат в ряд $N + 1$ камней, сначала N белых, и в конце цепочки — один чёрный. За один ход каждый может взять от 1 до 3 камней. Проигрывает тот, кто берет чёрный («несчастливый») камень.

Начнём анализ с простейших случаев. Если $N = 0$, то первый игрок проиграл, он может взять только чёрный камень. Если $N = 1, 2, 3$, то, наоборот, при правильной игре проигрывает второй игрок, потому что первый может забрать все камни, кроме чёрного. Вариант $N = 4$ снова приводит к проигрышу первого игрока, потому что забрать все белые камни он не может, а после его хода второй оставит только чёрный камень. Также проигрышными будут позиции при $N = 8, 12, 16, \dots$, т. е. при любых значениях N , которые делятся на 4.

Таким образом, для своего выигрыша игрок должен каждым своим ходом восстанавливать *инвариант*: число оставшихся белых камней должно быть кратно 4. Если инвариант выполнен в начальной позиции, положение проигрышное и первый игрок может надеяться только на ошибку соперника.

Пример 2. Пусть задан массив A длины n . Найдём инвариант цикла в программе суммирования элементов массива:

```
Sum:=0
нц для i от 1 до n
  Sum:=Sum+A[i]
кц
```

Здесь на каждом шаге к переменной Sum добавляется элемент массива $A[i]$, так что при любом i после окончания очередного шага цикла в Sum накоплена сумма всех элементов массива с номерами от 1 до i . Это и есть инвариант цикла. Поэтому сразу можно сделать вывод о том, что после завершения цикла в переменной Sum будет записана сумма всех элементов массива.

Аналогично можно показать, что в алгоритме поиска наименьшего значения в массиве:

```
Min:=A[1]
нц для i от 2 до n
  если A[i]<Min то
    Min:=A[i]
все
кц
```

инвариант формулируется так: после выполнения каждого шага цикла в переменной Min записан минимальный из первых i элементов. Отсюда сразу следует, что после завершения цикла (при $i = n$) в этой переменной будет минимальный из всех элементов.

Пример 3. Для того же массива найдем инвариант цикла в программе сортировки элементов массива методом пузырька:

```
нц для i от 1 до n-1
  нц для j от n-1 до i шаг -1
    если A[j]>A[j+1] то
      c:=A[j]; A[j]:=A[j+1]; A[j+1]:=c;
    все
  кц
кц
```

До начала алгоритма элементы расположены произвольно. На каждом шаге внешнего цикла на свое место «всплывает» один элемент массива. Поэтому инвариант этого цикла можно сформулировать так: «После выполнения i -го шага цикла первые i элементов массива отсортированы и установлены на свои места».

Теперь построим инвариант внутреннего цикла. В этом цикле очередной «лёгкий» элемент поднимается вверх к началу массива. Перед первым шагом внутреннего цикла элемент, который будет стоять на i -м месте в отсортированном массиве, может находиться в любой ячейке от $A[i]$ до $A[n]$. После каждого шага его «зона нахождения» сужается на одну позицию, так что инвариант внутреннего цикла можно сформулировать так: «Элемент, который будет стоять на i -м месте в отсортированном массиве, может находиться в любой ячейке от $A[i]$ до $A[j]$ ». Очевидно, что когда в конце этого цикла $j = i$, элемент $A[i]$ встаёт на своё место.

В предыдущих примерах мы определяли инвариант готового цикла. Теперь покажем, как можно строить цикл с помощью заранее выбранного инварианта.

Пример 4. Рассмотрим алгоритм быстрого возведения в степень, основанный на использовании операций возведения в квадрат и умножения. Он использует две очевидные формулы:

- (1) $a^k = a^{k-1} \cdot a$ при нечётной степени k и
- (2) $a^k = (a^2)^{k/2}$ при чётной степени k .

Покажем, как работает алгоритм, на примере возведения числа a в степень 7:

$$a^7 = a^6 \cdot [a] = (a^2)^3 \cdot [a] = (a^2)^2 \cdot [a^2 \cdot a] = (a^4)^1 \cdot [a^2 \cdot a] = \\ = (a^4)^0 \cdot [a^4 \cdot a^2 \cdot a] = [a^4 \cdot a^2 \cdot a].$$

Здесь поочерёдно применяются первая и вторая формулы. Заметим, что на каждом этапе выражение a^n можно представить в виде $a^n = b^k \cdot p$, где через p обозначена часть, взятая выше в квадратных скобки. Если нам каким-то образом удастся уменьшить k до нуля, сохранив это равенство, то мы получим $a^n = p$, т. е. задача будет решена, а результат будет находиться в переменной p .

Таким образом, равенство $a^n = b^k \cdot p$ можно использовать как инвариант цикла. Для того чтобы обеспечить выполнение этого равенства в начальный момент, можно принять, например, $b = a$, $k = n$ и $p = 1$. Далее в цикле применяются формулы (1) и (2) (в зависимости от чётности k на данном шаге). Цикл заканчивается, когда $k = 0$. В результате получаем следующее решение:

```
b:=a; k:=n; p:=1
нц пока k<>0
  если mod(k,2)=0 то
    k:=div(k,2)
    b:=b*b
  иначе
    k:=k-1
    p:=b*p
все
кц
вывод p
```

Заметим, что инвариант цикла $a^n = b^k \cdot p$ выполняется до начала цикла, после каждого шага, а также после завершения цикла. Таким образом, мы написали код программы и одновременно доказали правильность этого блока.

Спецификация

Для доказательства правильности программы необходимо иметь спецификацию — точное описание того, что должно быть сделано в результате работы программы.

Спецификация — точная и полная формулировка задачи, содержащая информацию, необходимую для построения алгоритма её решения.

На практике спецификации программ обычно формулируют на естественном языке, в котором слова могут иметь несколько разных значений. Для строгого доказательства желательно, чтобы спецификация была задана в формальном виде, с помощью формул или соотношений между величинами.

По предложению английского ученого Ч. Хоара, спецификация записывается в форме $\{Q\}S\{R\}$, где Q — начальное условие, S — программа и R — утверждения, описывающие конечный результат. Запись $\{Q\}S\{R\}$ означает следующее: «Если выполнение программы S началось в состоянии, удовлетворяющем Q , то гарантируется, что оно завершится через конечное время в состоянии, удовлетворяющем R ».

Корректная программа — это программа, соответствующая спецификации.

Если для исходных данных не выполняется условие Q , программа должна сообщать об этом пользователю и закончить работу. Это говорит о надёжности программы.

Например, для алгоритма Евклида условия Q и R могут выглядеть так:

$$Q: m \geq n > 0, \quad R: a = \text{НОД}(m, n),$$

а для программы суммирования элементов массива $A[1:n]$ (см. пример 2 на стр. 40) — так:

$$Q: n > 0, \quad R: \text{Sum} = \sum_{i=1}^n A[i] = A[1] + A[2] + \dots + A[n].$$

Спецификации могут (и должны) быть составлены не только для программы в целом, но и для её отдельных блоков (процедур, функций, циклов и т. д.). Полезно вносить утверждения Q и R прямо в текст программы. Построенная таким образом *аннотированная программа* — это ещё один шаг к доказательному программированию.

Ч. Хоар разработал специальный аппарат, позволяющий доказывать правильность программы на основе спецификаций отдельных блоков. Приведём простейшие правила преобразования:

- если $\{Q\}S\{P\}$ и $P \Rightarrow R$ (из истинности P следует истинность R), то $\{Q\}S\{R\}$;
- если $\{Q\}S\{P\}$ и $R \Rightarrow Q$, то $\{R\}S\{P\}$;
- если программа S — это последовательное выполнение блоков S_1 и S_2 , для которых выполняются спецификации $\{Q\}S_1\{P\}$ и $\{P\}S_2\{R\}$, то выполняется спецификация $\{Q\}S\{R\}$.

Доказательство правильности программ используют в двух ситуациях:

- доказывают правильность готовых программ (верификация программ);
- строят программы одновременно с доказательством их правильности (синтез программ).

Как правило, верификация — это очень трудоёмкий и сложный процесс, и оказывается значительно проще использовать доказательства правильности во время разработки программы. При этом программы получаются проще, эффективнее и значительно надёжнее.



Вопросы и задания

1. Зачем нужно доказывать правильность программ?
2. Расскажите о двух подходах к проверке правильности программ.
3. Почему с помощью тестирования сложно доказать правильность программы? В каких случаях это всё же можно сделать? Приведите примеры.
4. Что изменится в доказательстве алгоритма Евклида, если m и n — это произвольные натуральные числа (неравенство $m \geq n$ может не выполняться)?
5. Что такое инвариант цикла?
6. Зачем нужно определять инвариант цикла?
7. Что такое спецификация? Почему желательно формулировать её в виде формальных утверждений, а не на естественном языке?
8. Объясните запись $\{Q\}S\{R\}$.
9. Какая программа называется корректной?
10. Как вы думаете, можно ли назвать корректной программу, которая «зависает» при неверных входных данных? Обсудите этот вопрос в классе.
11. Что такое верификация программы?

12. Как вы думаете, что сложнее — доказывать правильность готовой программы или сразу писать программу, доказывая правильность отдельных блоков? Почему? Обсудите этот вопрос в классе.

Задачи

1. Докажите, что следующие операторы дают одинаковый результат при любых значениях L и R :

$c := \text{div}(L+R, 2)$ $c := L + \text{div}(R-L, 2)$

Какие достоинства и недостатки есть у каждого метода вычисления этой величины?

2. Докажите, что в результате выполнения следующего фрагмента программы в переменной M не всегда будет записано максимальное из трёх чисел (a , b и c):

```
M := a
если b > a то M := b все
если c > b то M := c все
```

Приведите контрпример, т. е. такие значения a , b и c , при которых значение M будет отличаться от $\max(a, b, c)$. Как можно исправить эту программу, заменив в ней всего один символ?

3. Докажите или опровергните правильность программы для выбора максимального из трёх значений, записанных в переменных a , b и c :

```
если a > b то M := a
иначе если b > c то M := b
иначе если c > a то M := c
все; все; все
```

Если эта программа некорректная, приведите контрпример. Может ли быть, что при каких-то входных данных значение переменной M будет неопределённым?

4. Докажите, что следующий фрагмент программы правильно сортирует значения в переменных a , b и c по возрастанию, т. е. всегда получается $a \leq b \leq c$:

```
если a > b то поменять (a, b) все
если b > c то поменять (b, c) все
если a > b то поменять (a, b) все
```

Алгоритм поменять меняет местами значения переменных-аргументов.

5. В игре «ним» двое игроков по очереди берут камни из двух кучек. За один ход можно взять любое ненулевое количество камней, но только из одной кучки. Тот, кому не осталось камней, проигрывает. Как определить, кто выиграет при правильной игре? Какой инвариант обеспечивает выигрыш?

6. Определите инвариант цикла для следующего алгоритма двоичного поиска (предполагается, что элементы массива A отсортированы по неубыванию):

```
L:=1; R:=n+1
нц пока L<R-1
  c:=div(L+R, 2)
  если X<A[c] то
    R:=c
  иначе
    L:=c
все
кц
```

Используя найденный инвариант, определите, какой именно элемент массива будет найден, если в массиве есть несколько элементов, равных X . Как нужно изменить инвариант (и цикл), чтобы найти *первый* элемент, равный X ?

7. Определите инварианты для следующих циклов. Что будет вычислено в переменной b ?

а)

```
k:=0; b:=1;
нц пока k<n
  k:=k+1;
  b:=b*a;
кц
```

б)

```
k:=n; b:=1;
нц пока k>0
  k:=k-1;
  b:=b*a;
кц
```

8. Определите условия Q и R для алгоритмов:

- нахождения суммы всех делителей числа;
- проверки числа на простоту;
- определения количества слов в символьной строке;
- двоичного поиска элемента в отсортированном массиве;
- перестановки элементов массива в обратном порядке;
- преобразования числа из символьной записи в значение целого типа.

9. Предложите другие начальные значения переменных b , k и p в алгоритме быстрого возведения в степень. Инвариант цикла должен сохраниться.

10. Оцените сложность алгоритма быстрого возведения в степень при $n = 2^m$, где m — натуральное число.

Практические работы к главе 5

Работа № 36 «Машина Тьюринга»

Работа № 37 «Машина Поста»

Работа № 38 «Нормальные алгоритмы Маркова»

Работа № 39 «Вычислимые функции»

Работа № 40 «Инвариант цикла»

ЭОР к главе 5 на сайте ФЦИОР (<http://fcior.edu.ru>)

- Алгоритмически неразрешимые задачи

Самое важное в главе 5

- Интуитивное понятие алгоритма, которое мы использовали ранее, непригодно для математического доказательства неразрешимости задач.
- Входные и выходные данные любого алгоритма можно закодировать в виде последовательностей символов некоторого алфавита (и даже двоичного алфавита).
- Про любой алгоритм можно сказать следующее:
 - алгоритм получает на вход дискретный объект (например, слово);
 - алгоритм обрабатывает входной объект по шагам (дискретно), строя на каждом шаге промежуточные дискретные объекты; этот процесс может закончиться или не закончиться;
 - если выполнение алгоритма заканчивается, его результат — это объект, построенный на последнем шаге;
 - если выполнение алгоритма не заканчивается (алгоритм закликивается) или заканчивается аварийно, то результат его работы при данном входе не определен.

- Алгоритм — это программа для некоторого исполнителя.
- Универсальный исполнитель — это исполнитель, который может моделировать работу любого другого исполнителя. Это значит, что для любого алгоритма, написанного для любого исполнителя, существует эквивалентный алгоритм для универсального исполнителя.
- Все универсальные исполнители эквивалентны между собой.
- Каждый алгоритм задаёт (вычисляет) функцию, которая преобразует входное слово в результат (выходное слово). Алгоритмы называются эквивалентными, если они задают одну и ту же функцию.
- Вычислимая функция — это функция, для вычисления которой существует алгоритм. Любая вычислимая функция может задаваться разными алгоритмами.
- Алгоритмически неразрешимая задача — это задача, соответствующая невычислимой функции.
- Говорят, что алгоритм имеет асимптотическую сложность $O(f(n))$, если найдётся такая постоянная c , что, начиная с некоторого $n = n_0$, выполняется условие $T(n) \leq c \cdot f(n)$.
- Задачи оптимизации, которые решаются только полным перебором вариантов, могут иметь, например, асимптотическую сложность $O(2^n)$ или $O(n!)$. Эти функции при больших n возрастают быстрее, чем многочлен любой степени.
- Чтобы обеспечить надёжность программы, используют методы доказательного программирования: разработка программы ведётся одновременно с доказательством её правильности.
- Инвариант цикла — это соотношение между величинами, которое остаётся справедливым после завершения любого шага цикла.

Глава 6

Алгоритмизация и программирование

§ 38

Целочисленные алгоритмы

Во многих задачах все исходные данные и необходимые результаты — целые числа. При этом всегда желательно, чтобы все промежуточные вычисления тоже проводились только с целыми числами. На это есть, по крайней мере, две причины:

- процессор, как правило, выполняет операции с целыми числами значительно быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними также выполняются без погрешностей (если, конечно, не происходит переполнение разрядной сетки).

Решето Эратосфена

Во многих прикладных задачах, например, при шифровании с помощью алгоритма RSA, используются простые числа (вспомните материал главы 10 учебника для 10 класса). Основные задачи при работе с простыми числами — это проверка числа на простоту и нахождение всех простых чисел в заданном диапазоне.

Пусть задано некоторое натуральное число N и требуется найти все простые числа в диапазоне от 2 до N . Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все числа от 2 до N , и каждое из них отдельно проверяется на простоту. Например, можно проверить, есть ли у числа k делители в диапазоне от 2 до $\sqrt{k}$. Если ни одного такого делителя нет, то число k простое.

Описанный метод при больших N работает очень медленно, он имеет асимптотическую сложность $O(N\sqrt{N})$. Греческий математик Эратосфен Киренский (275–194 гг. до н. э.) предложил другой алгоритм, который работает намного быстрее (сложность $O(N \log \log N)$):

- 1) выписать все числа от 2 до N ;
- 2) начать с $k = 2$;
- 3) вычеркнуть все числа, кратные k ($2k, 3k, 4k$ и т. д.);

- 4) найти следующее невычеркнутое число и присвоить его переменной k ;
- 5) повторять шаги 3 и 4, пока $k < N$.

Покажем работу алгоритма при $N = 16$:

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

Первое невычеркнутое число — это 2, поэтому вычёркиваем все чётные числа:

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

Далее вычёркиваем все числа, кратные 3:

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

Все числа, кратные 5 и 7, уже вычеркнуты. Таким образом, получены простые числа 2, 3, 5, 7, 11 и 13.

Классический алгоритм можно улучшить, уменьшив количество операций. Заметьте, что при вычёркивании чисел, кратных трём, нам не пришлось вычёркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим вычеркнуть все числа, кратные некоторому k , например $k = 5$. При этом числа $2k$, $3k$ и $4k$ уже были вычеркнуты на предыдущих шагах, поэтому нужно начать не с $2k$, а с k^2 . Тогда получается, что при $k^2 > N$ вычёркивать уже будет нечего, что мы и увидели в примере. Поэтому можно использовать улучшенный алгоритм:

- 1) выписать все числа от 2 до N ;
- 2) начать с $k = 2$;
- 3) вычеркнуть все числа, кратные k , начиная с k^2 ;
- 4) найти следующее невычеркнутое число и присвоить его переменной k ;
- 5) повторять шаги 3 и 4, пока $k^2 \leq N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения данных — массив логических величин с индексами от 2 до N . Как и в учебнике 10 класса, слева будем писать программу на школьном алгоритмическом языке, а справа — на языке Паскаль.

Объявление переменных в программе будет выглядеть так (для $N = 100$):

```
цел i, k, N=100
логтаб A[2:N]
```

```
const N=100;
var i, k: integer;
    A: array[2..N] of boolean;
```

Если число i не вычеркнуто, будем хранить в элементе массива $A[i]$ истинное значение, если вычеркнуто — ложное. В самом начале нужно заполнить массив истинными значениями:

```
нц для i от 2 до N      for i:=2 to N do
    A[i]:=да             A[i]:=True;
кц
```

В основном цикле выполняется описанный выше алгоритм:

```
к:=2                    к:=2;
нц пока k*k<=N          while k*k<=N do begin
    если A[k] то         if A[k] then begin
        i:=k*k           i:=k*k;
        нц пока i<=N     while i<=N do begin
            A[i]:=нет     A[i]:=False;
            i:=i+k        i:=i+k
        кц               end
        все              end;
        k:=k+1           k:=k+1
    кц                   end;
кц
```

Обратите внимание, что для того, чтобы вообще не применять вещественную арифметику, мы заменили условие $k \leq \sqrt{N}$ на равносильное условие $k^2 \leq N$, в котором используются только целые числа.

После завершения этого цикла невычеркнутыми остались только простые числа, для них соответствующие элементы массива содержат истинные значения. Эти числа нужно вывести на экран:

```
нц для i от 2 до N      for i:=2 to N do
    если A[i] то         if A[i] then
        вывод i,нс       writeln(i);
    все
кц
```

«Длинные» числа

Современные алгоритмы шифрования используют достаточно длинные ключи, которые представляют собой числа длиной 256 битов и больше. С ними нужно выполнять разные операции:

складывать, умножать, находить остаток от деления. Вопрос состоит в том, как хранить такие числа в памяти, где для целых чисел отводится память значительно меньших размеров (обычно до 64 битов). Ответ достаточно очевиден: нужно разбить длинное число на части так, чтобы оно занимало несколько ячеек памяти.

«Длинное» число — это число, которое не помещается в переменную стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «длинной арифметикой».

Для хранения «длинного» числа удобно использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 9 таким образом:

	0	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	0	0

Такой способ имеет ряд недостатков:

- 1) нужно где-то хранить длину числа, иначе числа 12345678, 123456780 и 1234567800 будет невозможно различить;
- 2) неудобно выполнять арифметические операции, которые начинаются с младшего разряда;
- 3) память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд — число от 0 до 9.

Чтобы избавиться от первых двух проблем, достаточно «развернуть» массив наоборот, так чтобы младший разряд находился в $A[0]$. В этом случае на рисунках удобно применять обратный порядок элементов:

	9	8	7	6	5	4	3	2	1	0
A	0	0	1	2	3	4	5	6	7	8

Теперь нужно найти более экономичный способ хранения длинного числа. Например, разместим в одном элементе массива три разряда числа, начиная справа:

	9	8	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	0	12	345	678

Здесь использовано равенство

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0.$$

Фактически мы представили исходное число в системе счисления с основанием 1000.

Сколько разрядов можно хранить в одном элементе массива? Это зависит от размера элемента. Например, если переменная занимает 4 байта и число хранится со знаком, допустимый диапазон его значений:

$$\text{от } -2^{31} = -2\,147\,483\,648 \text{ до } 2^{31} - 1 = 2\,147\,483\,647.$$

В таком элементе можно хранить до 9 разрядов десятичного числа, т.е. использовать систему счисления с основанием 1 000 000 000. Однако нужно учитывать, что с такими числами будут выполняться арифметические операции, результат которых должен помещаться в переменную некоторого типа. Например, если надо умножать разряды этого числа на число $k < 100$ и в языке программирования нет 64-битных целочисленных типов данных, то в элементе массива можно хранить не более 7 разрядов.

Покажем на примере, как можно использовать систему счисления с основанием 1 000 000 для выполнения операций с «длинными» числами.

Задача. Вычислить точно значение факториала $100! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100$ и вывести его на экран в десятичной системе счисления (это число состоит более чем из сотни цифр и явно не помещается в одну переменную).

Для хранения «длинного» числа будем использовать целочисленный массив A. Определим необходимую длину массива. Заметим, что

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot 99 \cdot 100 < 100^{100}.$$

Число 100^{100} содержит 201 цифру, поэтому число $100!$ содержит не более 200 цифр. Если в каждом элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 элементов:

```
цел N=33          const N=33;
целтаб A[0:N]     var A: array[0..N] of integer;
```

Чтобы найти $100!$, нужно сначала присвоить «длинному» числу значение 1, а затем последовательно умножать его на все числа от 2 до 100. Запишем эту идею на псевдокоде, обозначив через A «длинное» число, находящееся в массиве A:

```
[A]:=1
нц для k от 2 до 100
  [A]:=[A]*k
кц
```

Записать в «длинное» число единицу — значит присвоить элементу $A[0]$ значение 1, а в остальные переменные записать нули:

```
A[0]:=1
нц для i от 1 до N
  A[i]:=0
кц
```

Таким образом, остаётся научиться умножать «длинное» число на «короткое» ($k \leq 100$). «Короткими» обычно называют числа, которые помещаются в переменную одного из стандартных типов данных.

Попробуем сначала выполнить такое умножение на примере. Предположим, что в каждом элементе массива хранятся 6 цифр «длинного» числа, т. е. используется система счисления с основанием $d = 1\,000\,000$. Тогда число $[A] = 12345678901734567$ хранится в трёх элементах:

	2	1	0
A	12345	678901	734567

Пусть $k = 3$. Начинаем умножать с младшего разряда: $734567 \cdot 3 = 2203701$. В нулевом разряде могут находиться только 6 цифр, значит, старшая двойка перейдёт в перенос в следующий разряд. В программе для выделения переноса r можно использовать целочисленное деление на основание системы счисления d . Остаток от деления — это то, что остаётся в текущем разряде. Поэтому получаем:

```
s:=A[0]*k
A[0]:=mod(s,d)
r:=div(s,d)
```

```
s:=A[0]*k;
A[0]:=s mod d;
r:=s div d;
```

Для следующего разряда будет всё то же самое, только в первой операции к произведению нужно добавить перенос из предыдущего разряда, который был записан в переменную r . Приняв в самом начале $r = 0$, запишем умножение «длинного» числа на «короткое» в виде цикла по всем элементам массива, от $A[0]$ до $A[N]$:

```
r:=0
нц для i от 0 до N
  s:=A[i]*k+r
  A[i]:=mod(s,d)
  r:=div(s,d)
кц
```

```
r:=0;
for i:=0 to N do begin
  s:=A[i]*k+r;
  A[i]:=s mod d;
  r:=s div d
end;
```

В свою очередь, эти действия нужно выполнить в другом (внешнем) цикле для всех k от 2 до 100:

```
нц для k от 2 до 100
  ...
кц
```

```
for k:=2 to 100 do begin
  ...
end;
```

После этого в массиве A будет находиться искомое значение $100!$, остаётся вывести его на экран. Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому в массиве

	2	1	0
A	1	2	3

хранится значение 1000002000003 , а не 123. Кроме того, старшие нулевые разряды выводить на экран не надо. Поэтому при выводе требуется:

- 1) найти первый (старший) ненулевой разряд числа;
- 2) вывести это значение без лидирующих нулей;
- 3) вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Поскольку мы знаем, что число не равно нулю, старший ненулевой разряд можно найти в таком цикле¹:

```
i:=N
нц пока A[i]=0
  i:=i-1
кц
```

```
i:=N;
while A[i]=0 do
  i:=i-1;
```

Старший разряд выводим обычным образом (без лидирующих нулей):

```
вывод A[i]
```

```
write(A[i]);
```

¹ Подумайте, что изменится, если выводимое число может быть нулевым.

Для остальных разрядов будем использовать специальную процедуру Write6:

```
нц для k от i-1 до 0 шаг -1   for k:=i-1 downto 0 do
  Write6(A[k])                Write6(A[k]);
кц
```

Эта процедура последовательно выводит цифры десятичного числа, начиная с сотен тысяч и кончая единицами:

<pre>алг Write6 (цел x) нач цел M, xx xx:=x M:=100000 нц пока M>0 вывод div(xx,M) xx:=mod(xx,M) M:=div(M, 10) кц кон</pre>	<pre>procedure Write6(x: integer); var M: integer; begin M:=100000; while M>0 do begin write(x div M); x:=x mod M; M:=M div 10 end end;</pre>
---	--

Для того чтобы разобраться, как она работает, выполните «ручную прокрутку» при различных значениях x (например, возьмите $x = 1$, $x = 123$ и $x = 123456$).



Вопросы и задания

1. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» по сравнению с проверкой каждого числа на простоту?
2. Что такое «длинные» числа?
3. В каких случаях необходимо применять «длинную арифметику»?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения «длинного» числа символьную строку? Какие проблемы при этом могут возникнуть?
6. Почему неудобно хранить «длинное» число, записывая первую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов десятичной записи числа можно хранить в 16-битном элементе массива?
9. Объясните, какие проблемы возникают при выводе длинного числа. Как их можно решать?
10. Объясните работу процедуры Write6.



Задачи

1. Докажите, что если у числа k нет ни одного делителя в диапазоне от 2 до $\sqrt{k}$, то оно простое.
2. Напишите две программы, которые находят все простые числа в диапазоне от 2 до N двумя разными способами:
 - а) проверкой каждого числа из этого интервала на простоту;
 - б) используя решето Эратосфена.
 Сравните число шагов цикла (время работы) этих программ для разных значений N . Постройте для каждого варианта зависимость количества шагов от N , сделайте выводы о сложности алгоритмов.
3. Докажите, что в приведённой в параграфе программе вычисления $100!$ не будет переполнения при использовании 32-битных целых переменных.
4. Можно ли в программе вычисления $100!$ в одной ячейке массива хранить 9 цифр «длинного» числа?
5. Без использования программы определите, сколько нулей стоит в конце числа $100!$
6. Соберите всю программу и вычислите $100!$. Сколько цифр входит в это число?
7. Оформите вывод «длинного» числа на экран в виде отдельной процедуры. Учтите, что число может быть нулевым.
- *8. Придумайте другой способ вывода «длинного» числа, использующий символьные строки.
9. Напишите процедуру для ввода «длинных» чисел из файла.
10. Напишите процедуры для сложения и вычитания длинных чисел.
- *11. Напишите процедуры для умножения и деления «длинных» чисел.
- *12. Напишите процедуру для извлечения квадратного корня из «длинного» числа.

§ 39

Структуры (записи)

Зачем нужны структуры?

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляров и т. д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но в массиве используются элементы одного типа, тогда как информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т. д.) так, чтобы i -й элемент каждого массива относился к книге с номером i . Но такой подход оказывается слишком неудобным и ненадежным. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

Возникает естественная идея — объединить все данные, относящиеся к книге, в единый блок памяти, который в программировании называется структурой.

Структура — это тип данных, который может включать несколько полей — элементов разных типов (в том числе и другие структуры).

В Паскале структуры по традиции называют *записями* (англ. *record* — запись). Далее в этой главе мы будем использовать возможности свободно распространяемого компилятора *FreePascal*.

Объявление структур

Как и любые переменные в Паскале, структуры необходимо объявлять. До этого мы работали с простыми типами данных (целыми, вещественными, логическими и символьными), а также с массивами этих типов. Вы знаете, что при объявлении переменных и массивов указывается их тип, поэтому для того, чтобы работать со структурами, нужно ввести новый тип данных.

Построим структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только¹:

- фамилию автора (строка не более 40 символов);
- название книги (строка не более 80 символов);
- имеющееся в библиотеке количество экземпляров (целое число).

¹ Конечно, в реальной ситуации данных больше, но принцип не меняется.

Объявление такого составного типа имеет вид:

```
type
  TBook = record
    author: string[40]; {автор, строка}
    title: string[80];  {название, строка}
    count: integer      {количество, целое}
  end;
```

Объявления типов данных начинаются с ключевого слова **type** (в переводе с англ. — тип) и располагаются выше блока объявления переменных. Имя нового типа — **TBook** — это удобное сокращение от английских слов **Type Book** (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования. Слово **record** означает, что этот тип данных — **структура (запись)**; далее перечисляются поля и указываются их типы. Объявление структуры заканчивается ключевым словом **end**.

Обратите внимание, что для строк *author* и *title* указан максимальный размер. Это сделано для того, чтобы точно определить, сколько места нужно выделить на них в памяти.

В результате такого объявления никаких структур в памяти не создается: мы просто описали новый тип данных, чтобы транслятор знал, что делать, если мы захотим его использовать.

Теперь можно использовать тип **TBook** так же, как и простые типы, для объявления переменных и массивов:

```
const N = 100;
var B: TBook;
    Books: array[1..N] of TBook;
```

Здесь введена переменная *B* типа **TBook** и массив *Books*, состоящий из элементов того же типа.

Иногда бывает нужно определить размер одной структуры. Для этого используется стандартная функция **sizeof**, которой можно передать имя типа, а также переменную или массив:

```
Writeln(sizeof(TBook));
Writeln(sizeof(B));
Writeln(sizeof(Books));
```

Первые две команды выведут на экран размер одной структуры (124 байта), а последняя — размер выделенного массива из 100 структур. Размер структуры вызывает некоторые вопросы: каждый элемент строки занимает 1 байт, а целое число — 2 бай-

та¹, поэтому простой подсчёт дает значение $40 + 80 + 2 = 122$. Откуда появились ещё 2 байта? Дело в том, что строка *author* из 40 символов фактически занимает 41 байт, а строковое поле *title* — 81 байт: 1 дополнительный байт расходуется на хранение фактического размера строки².

Обращение к полю структуры

Для того чтобы работать не со всей структурой, а с отдельными полями, используют так называемую **точечную запись**, разделяя точкой имя структуры и имя поля. Например, *B.author* обозначает «поле *author* структуры *B*», а *Books[5].count* — «поле *count* элемента массива *Books[5]*». Для определения размера полей в байтах, можно снова использовать функцию *sizeof*:

```
writeln(sizeof(B.author));
writeln(sizeof(B.title));
writeln(sizeof(B.count));
```

и мы увидим на экране числа 41, 81 и 2.

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
readln(B.author);
readln(B.title);
readln(B.count);
```

присваивать новые значения:

```
B.author:='Пушкин А.С.';
B.title:='Полтава';
B.count:=1;
```

использовать при обработке данных:

```
p:=Pos(' ', B.author);
fam:=Copy(B.author, 1, p-1); { только фамилия }
B.count:=B.count - 1;        { взяли одну книгу }
if B.count=0 then
  writeln('Этих книг больше нет!');
```

¹ Здесь приведены данные для FreePascal со стандартными настройками. В других системах программирования эти величины могут быть иными, что связано с различными способами хранения данных.

² Поэтому в классическом варианте языка Паскаль строки не могут быть длиннее 255 символов.

и выводить на экран:

```
writeln(B.author, ' ', B.title, '. ', B.count, ' шт.');
```

Работа с файлами

В программах, которые работают с данными на диске, бывает нужно читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет выполнять файловые операции проще и надёжнее (с меньшей вероятностью ошибки). Для этого нужно использовать файлы специального типа, которые называются **типизированными**. Все записываемые в них данные должны иметь одинаковый тип. В отличие от текстовых файлов данные в типизированных файлах хранятся во внутреннем формате, т. е. так, как они представлены в памяти компьютера во время работы программы. Например, можно сделать файл целых чисел или логических величин.

В данном случае нас интересует файл структур типа *TBook*, так что файловая переменная *F* для работы с типизированным файлом должна быть объявлена так:

```
var F: file of TBook;
```

Запись структуры в файл выполняется стандартным способом:

```
Assign(F, 'books.dat');
Rewrite(F);
B.author:='Тургенев И.С.';
B.title:='Муму';
B.count:=2;
write(F, B);
Close(F);
```

Напомним, что процедура *Assign* связывает файловую переменную с файлом на диске, процедура *Rewrite* открывает файл на запись, а процедура *Close* — завершает запись на диск и закрывает файл.

Процедура *Write*, определив, что файловая переменная *F* связана с типизированным файлом структур, записывает в файл одну структуру во внутреннем формате. При попытке передать этой процедуре переменную другого типа произойдёт ошибка и аварийный останов программы.

С помощью цикла можно записать в файл весь массив структур:

```
for i:=1 to N do
  write(F, Books[i]);
```

Прочитать из файла одну структуру и вывести её поля на экран можно следующим образом:

```
Assign(F, 'books.dat');
Reset(F);
read(F, B);
Writeln(B.author, ', ', B.title, ', ', B.count);
Close(F);
```

Процедура `read`, получив ссылку `F` на типизированный файл, может принимать в качестве следующих параметров только структуры типа `TBook`.

Если заранее известно, сколько структур записано в файле, при чтении их в массив можно применить цикл с переменной:

```
for i:=1 to N do
  read(F, Books[i]);
```

Если же число структур неизвестно, нужно выполнять чтение до тех пор, пока файл не закончится, т. е. функция `Eof` не вернёт истинное значение:

```
i:=0;
while not Eof(F) do begin
  i:=i+1;
  read(F, Books[i])
end;
```

Здесь целая переменная `i` играет роль счётчика: в ней в конце каждого шага цикла хранится количество фактически прочитанных структур.

Сортировка

Для сортировки массива структур применяют те же методы, что и для сортировки массива простых переменных. Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют **ключевым полем** или **ключом**, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей (**составной ключ**).

Отсортируем массив `Books` (типа `TBook`) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле `author`. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка *методом пузырька* выглядит так:

```
for i:=1 to N-1 do
  for j:=N-1 downto i do
    if Books[j].author > Books[j+1].author then
      begin
        B:=Books[j]; Books[j]:=Books[j+1];
        Books[j+1]:=B;
      end;
```

Здесь `i` и `j` — целочисленные переменные, а `B` — вспомогательная структура типа `TBook`.

Как вы знаете из курса 10 класса, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые различающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией «Волк» окажется выше в отсортированном списке, чем строка с более длинной фамилией «Волков», даже с учётом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем — по второй букве.

Возможно, что структуры требуется отсортировать так, чтобы не перемещать их в памяти. Например, они очень большие, и многократное копирование целых структур занимает много времени; или по каким-то другим причинам перемещать структуры нельзя. При таком ограничении нужно вывести на экран или в файл отсортированный список. В этом случае применяют **сортировку по указателям**, в которой используется дополнительный массив переменных специального типа — указателей.

Указатель — это переменная, в которой можно сохранить адрес любой переменной заданного типа.

То есть содержимое указателя — это адрес памяти. Чтобы избежать случайных ошибок, каждому указателю при объявлении присваивается тип данных, адреса которых он может хранить. Например, объявление

```
type PBook = ^TBook;
```

вводит новый тип данных — указатель на структуру типа `TBook`. Адреса переменных других типов в такой указатель записывать

нельзя. Имя типа-указателя удобно начинать с буквы «P» от английского слова *pointer* — указатель.

Для сортировки массива *Books* нужно разместить в памяти массив таких указателей и одну вспомогательную переменную, которая будет использована при сортировке:

```
var p: array[1..N] of PBook;
    p1: PBook;
```

Следующий этап — расставить указатели так, чтобы *i*-й указатель был связан с *i*-й структурой из массива *Books*:

```
for i:=1 to N do p[i]:=@Books[i];
```

Знак @ обозначает операцию взятия адреса, т. е. в указатель записывается адрес структуры.

Для того чтобы от указателя перейти к объекту, на который он ссылается, используют операцию ^. Например, в нашем случае (после показанной выше начальной установки указателей) запись $p[i]^$ обозначает то же самое, что и $Books[i]$, а $p[i]^title$ — то же самое, что и $Books[i].title$.

Теперь можно перейти к сортировке. Рассмотрим идею на примере массива из трёх структур. Сначала указатели стоят по порядку (рис. 6.1, а). В результате сортировки нужно переставить их так, чтобы $p[1]$ указывал на первую структуру в отсортированном списке, $p[2]$ — на вторую и т. д. (рис. 6.1, б).

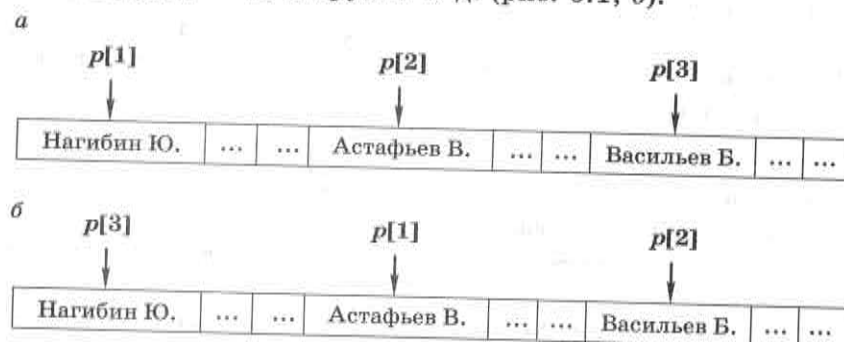


Рис. 6.1

Обратите внимание, что при этом сами структуры в памяти не перемещаются.

При сортировке обращение к полям структур идёт через указатели, и меняются местами тоже указатели, а не сами структуры:

```
for i:=1 to N-1 do
  for j:=N-1 downto i do
    if p[j]^author > p[j+1]^author then
      begin
        p1:=p[j]; p[j]:=p[j+1];
        p[j+1]:=p1;
      end;
```

Здесь использован *метод пузырька*, а $p1$ — это временная переменная типа *PBook*, которая служит для перестановки указателей.

Теперь можно вывести отсортированные данные, обращаясь к ним через указатели (а не через массив *Books*):

```
for i:=1 to N do
  writeln(p[i]^author, ' ',
          p[i]^title, ' ',
          p[i]^count);
```

Вопросы и задания

1. Что такое структура? В чём её отличие от массива?
2. В каких случаях использование структур даёт преимущества? Какие именно? Приведите примеры.
3. Как объявляется новый тип данных в Паскале? Выделяется ли при этом память?
4. Как обращаются к полю структуры? Расскажите о точечной записи.
5. Как определить, сколько байтов памяти выделяется на структуру?
6. Что такое типизированный файл? Чем он отличается от текстового?
7. Как работать с типизированными файлами?
8. Как можно сортировать структуры?
9. В каких случаях при сортировке желательно не перемещать структуры в памяти?
10. Что такое указатель?
11. Как записать в указатель адрес переменной?
12. Как обращаться к полям структуры через указатель?
13. Как используются указатели при сортировке?

Подготовьте сообщение

- а) «Структуры в языке Си»
- б) «Структуры в языке Javascript»
- в) «Структуры в языке Python»



Задачи

- Опишите структуру, в которой хранится информация о:
 - видеозаписи;
 - сотруднике фирмы;
 - самолёте;
 - породистой собаке.
- Постройте программу, которая работает с базой данных в виде типизированного файла. Ваша СУБД (система управления базой данных) должна иметь следующие возможности:
 - просмотр записей;
 - добавление записей;
 - удаление записей;
 - сортировка по одному из полей (через указатели).

§ 40

Множества

Множество — это некоторый набор элементов (см. § 4 учебника для 10 класса). В языке Паскаль есть специальный тип данных для работы с множествами. Покажем применение множеств в Паскале на простых примерах.

Пусть требуется определить количество знаков препинания в символьной строке *s*. Для решения этой задачи нужно в цикле проверить каждый символ строки *i*, если он совпадает с одним из знаков препинания, увеличить переменную-счётчик. Стандартное решение выглядит так:

```
var s: string;
    i, count: integer;
begin
  readln(s);
  count:=0;
  for i:=1 to Length(s) do
    if (s[i]='.') or (s[i]=',') or (s[i]=';') or
       (s[i]=':') or (s[i]='!') or (s[i]='?') then
      count:=count+1;
  writeln(count);
end.
```

Условный оператор, содержащий длинное перечисление всех возможных вариантов, выглядит некрасиво и затрудняет понимание программы. Вместо этого можно использовать *множество*,

состоящее из нужных знаков. В Паскале множество заключается в квадратные скобки, его элементы перечисляются через запятую:

```
['.', ',', ';', ':', '!', '?']
```

Теперь в условном операторе остаётся проверить (с помощью специального оператора *in*), принадлежит ли символ *s[i]* этому множеству:

```
if s[i] in ['.', ',', ';', ':', '!', '?'] then
  count:=count+1;
```

Запись получилась намного короче и понятнее. Аналогично можно подсчитать количество цифр:

```
if s[i] in ['0'..'9'] then count:=count+1;
```

Здесь через две точки обозначается диапазон, в данном случае он включает все символы-цифры десятичной системы.

При перечислении элементов множества можно использовать как диапазоны, так и отдельные элементы. Например, проверим, можно ли использовать символ *s[i]* в адресе электронной почты:

```
if s[i] in ['a'..'z', '0'..'9', '.', '-', '_'] then
  { символ правильный }
```

В программе можно использовать переменные специального типа, который описывает множество. Например, переменная

```
var digits: set of '0'..'9';
```

может хранить множество символов-цифр.

Составим программу, которая выводит на экран все различные цифры, присутствующие в символьной строке *s*:

```
var s: string;
    i: integer;
    c: char;
    digits: set of '0'..'9';
begin
  readln(s);
  digits:=[];
  for i:=1 to Length(s) do
    if s[i] in ['0'..'9']
    then digits:=digits+[s[i]];
  for c:='0' to '9' do
    if c in digits then writeln(c);
  end.
```

Здесь оператор

```
digits:=[];
```

записывает в переменную *digits* пустое множество, не содержащее ни одного элемента. Если найдена цифра, она добавляется к множеству (квадратные скобки вокруг *s[i]* превращают символ в множество, состоящее из одного элемента):

```
digits:=digits+[s[i]];
```

Заметим, что если данная цифра уже входит во множество, никаких изменений не происходит.

Порядок элементов во множестве не определен, так что элементы множества не различаются ни по номерам (как элементы массива), ни по именам (как поля структуры). Поэтому вывести множество на экран сразу нельзя. В последнем цикле мы перебираем все цифры, и если очередная цифра входит во множество *digits*, выводим её на экран.

Множество в Паскале может содержать до 256 элементов, поэтому допустимы, например, такие объявления:

```
var cs: set of char;
    bs: set of byte;
```

Переменная *cs* может хранить любое множество однобайтных символов, а переменная *bs* — любое множество целых чисел от 0 до 255.

Для элементов множества можно вводить свои обозначения (имена). Например, введём новый тип данных (так называемый *перечислимый тип*), который описывает различные свойства шрифта:

```
type TFontStyles=(fsBold, fsItalic, fsUnderline);
```

Здесь *fsBold* обозначает жирный шрифт, *fsItalic* — курсив, а *fsUnderline* — подчёркивание. Шрифт может иметь все эти характеристики в любых комбинациях, поэтому переменная, описывающая свойства шрифта, должна содержать множество элементов типа *fontStyles*:

```
var fs: set of TFontStyles;
```

Ей можно присваивать новые значения:

```
fs:=[fsBold,fsItalic];
```

Можно добавлять элементы к множеству:

```
fs:=fs+[fsBold,fsUnderline];
```

и удалять их из множества:

```
fs:=fs-[fsItalic];
```

Для того чтобы вывести такое множество на экран, нужно как-то перебрать в цикле все возможные элементы множества, т. е. все возможные значения перечислимого типа *TFontStyles*. Для этого введём новую переменную:

```
var style: TFontStyles;
```

Перечислимый тип *TFontStyles* — это порядковый тип, т. е. для каждого значения можно указать предыдущее и следующее. Значение *fsBold*, записанное на первом месте списка при объявлении типа *TFontStyles* — это первый элемент, имеющий минимальное значение; последний элемент *fsUnderline* имеет максимальное значение. Поэтому для перебора всех элементов множества можно использовать такой цикл:

```
fs:=[fsBold,fsItalic];
for style:=fsBold to fsUnderline do
  if style in fs then
    write(1)
  else write(0);
```

Этот фрагмент программы выводит битовую строку, которая обозначает присутствие элементов в множестве. В данном случае это будет строка 110, которая говорит о том, что в множестве есть элементы *fsBold* и *fsItalic*, а элемента *fsUnderline* нет.

Для множеств можно также использовать операцию пересечения, которая обозначается знаком «*». Эта операция выделяет общие элементы двух множеств.

Множества можно сравнивать с помощью операций «=» (равно), «<>» (не равно), «<=» (первое множество является подмножеством второго) и «>=» (второе множество является подмножеством первого).

В памяти каждый элемент множества занимает 1 бит. Поскольку в Паскале множество может содержать не более 256 элементов, переменная такого типа занимает не более 32 байтов. Рассмотрим снова множество цифр

```
var digits: set of '0'..'9';
```

Когда мы записали в эту переменную пустое множество, все биты были обнулены:

```
digits:=[]; {0000000000}
```

Теперь запишем непустое множество:

```
digits:=['1','3','5','7']; {0101010100}
```

В этом случае биты, соответствующие всем цифрам, входящим в множество, равны 1.

Операция объединения множеств сводится к выполнению логической операции «ИЛИ» для битовых полей:

```
digits:=['1','3','5','7'] + {0101010100}
        ['2','3','8'];      {0011000010}
                        {Результат: 0111010110}
```

а операция пересечения — к выполнению логической операции «И»:

```
digits:=['1','3','5','7'] * {0101010100}
        ['2','3','8'];      {0011000010}
                        {Результат: 0001000000}
```

Эти операции поддерживаются процессорами на аппаратном уровне, поэтому выполняются быстро. Подумайте, как можно представить операцию вычитания множеств с помощью стандартных логических операций.



Вопросы и задания

1. Что такое множество? Чем оно отличается от массива, символьной строки и структуры?
2. Обсудите, в каких задачах удобно использовать множества.
3. Какие ограничения существуют при использовании данных типа «множество» в языке Паскаль?
4. Почему нельзя объявить множества так?

```
var iSet: set of integer;
    sSet: set of string;
```
5. Объявите множество, в которое входят прописные и строчные латинские буквы.
6. Какие операции можно выполнять с множеством?
7. Как вы думаете, почему нельзя вывести множество на экран одной командой?
8. Как представлено множество в памяти компьютера? Что позволяет быстро выполнять операции с множествами?

9. Как можно выполнить операцию вычитания множеств, используя битовые логические операции?

10. Переменные a , b , c и x имеют тип **set of char**. Для переменных a , b и c заданы начальные значения:

```
a:=['a','b','c'];
b:=['b','c','d'];
c:=['a','c','e'];
```

Что будет записано в переменной x после выполнения следующего оператора?

- a) $x:=a*(b+c);$
- б) $x:=a*b+c;$
- в) $x:=a*b*c;$
- г) $x:=a*(b-c);$
- д) $x:=a*b-c;$
- е) $x:=a-b*c;$

Подготовьте сообщения

- а) «Множества в языке C++ (библиотека STL)»
- б) «Множества в языке Python»

Задачи

1. Введите с клавиатуры целое число и определите, есть ли в нём повторяющиеся цифры. Программа должна вывести ответ «да» или «нет».
2. Введите с клавиатуры символьную строку и определите, верно ли, что она представляет собой правильную запись числа в восьмеричной системе счисления.
3. Введите с клавиатуры символьную строку и определите, верно ли, что она представляет собой правильное имя переменной в языке Паскаль.
4. Введите с клавиатуры символьную строку и удалите в ней повторяющиеся символы.
5. Введите с клавиатуры символьную строку и определите количество различных символов в ней.
6. Введите с клавиатуры символьную строку и выведите все символы, которые встречаются в ней только один раз.
7. Введите с клавиатуры символьную строку и выведите в порядке убывания все цифры, которых *нет* в этой строке.
8. Введите с клавиатуры две символьные строки и определите, какие символы встречаются в обеих строках.
9. Введите с клавиатуры две символьные строки и определите, какие латинские буквы *не* встречаются ни в одной из них.

10. Допустимый набор символов включает строчные и прописные латинские буквы, а также цифры. Заполните массив из 20 элементов случайными символами из этого набора так, чтобы все символы в нём были различными.
11. Перепишите программу поиска случайных чисел в диапазоне от 2 до 255 с помощью решета Эратосфена так, чтобы для хранения простых чисел использовалась переменная типа «множество».

§41

Динамические массивы

Что это такое?

Когда мы объявляем массив, место для него выделяется во время трансляции, т. е. до выполнения программы. Такой массив называется **статическим**.

В то же время иногда размер данных заранее неизвестен. Например, пусть в файле записан массив чисел, которые нужно отсортировать. Их количество неизвестно, но известно, что такой массив помещается в оперативную память. В этом случае есть два варианта: 1) выделить заранее максимально большой блок памяти и 2) выделять память уже *во время выполнения программы* (т. е. **динамически**), когда станет известен необходимый размер массива.

Другой пример — задача составления *алфавитно-частотного словаря*. В файле находится список слов. Нужно вывести в другой файл все различные слова, которые встречаются в файле, и определить, сколько раз встречается каждое слово. Здесь проблема состоит в том, что нужный размер массива можно узнать только тогда, когда все различные слова будут найдены и, таким образом, задача решена. Поэтому нужно сделать так, чтобы массив мог «расширяться» в ходе работы программы.

Эти задачи приводят к понятию **динамических структур данных**, которые позволяют во время выполнения программы:

- создавать новые объекты в памяти;
- изменять их размер;
- удалять их из памяти, когда они не нужны.

Память под эти объекты выделяется в специальной области, которую обычно называют «**кучей**» (англ. *heap*).

Размещение в памяти

Задача 1. Требуется ввести с клавиатуры целое значение N , затем N целых чисел и вывести на экран эти числа в порядке возрастания.

Поскольку для сортировки все числа необходимо удерживать в памяти, нужно заводить массив, в который будут записаны все элементы. Поэтому алгоритм решения задачи на псевдокоде выглядит так:

```
прочитать данные из файла в массив
отсортировать их по возрастанию
вывести массив на экран
```

Все эти операции для обычных массивов подробно рассматривались в курсе 10 класса, поэтому здесь мы остановимся только на главной проблеме: как разместить в памяти массив, размер которого до выполнения программы неизвестен.

Для подобных случаев в версии языка Паскаль, которая поддерживается в среде *FreePascal*, существуют **динамические массивы**, которые объявляются без указания размера:

```
var A: array of integer;
```

Использовать сразу такой массив нельзя, поскольку его размер неизвестен. Попытка обращения к элементу, например $A[1]$, вызывает ошибку и аварийный останов программы.

Когда значение переменной N введено, можно фактически разместить массив в памяти, используя процедуру `SetLength` (англ. *set length* — установить длину):

```
SetLength(A, N);
```

Далее массив A используется так же, как и обычный (статический) массив. Остаётся один вопрос: в каком диапазоне находятся его индексы?

Вы помните, что границы изменения индексов обычного (статического) массива задаются при его объявлении, причём начальный индекс может быть любым. Индексы динамического массива *всегда начинаются с нуля*, так что к начальному элементу нужно обращаться как $A[0]$, а к последнему — как $A[N-1]$. Например, чтение данных с клавиатуры выполняется в цикле:

```
for i:=0 to N-1 do read(A[i]);
```

Кроме того, массив «знает» свою длину, которая вычисляется с помощью стандартной функции `Length` (в переводе с англ. — длина), и максимальный индекс, который возвращает функция `High` (в переводе с англ. — высокий). Поэтому предыдущий цикл можно заменить на такой:

```
for i:=0 to High(A) do read(A[i]);
```

Размер массива (количество элементов в нём) можно вычислить как `Length(A)` или `High(A)+1`.

Как только массив станет не нужен, можно удалить его из памяти, установив нулевую длину:

```
SetLength(A, 0);
```

Для такого (удалённого) массива нулевой длины функция `Length(A)` вернёт значение 0.

Таким же образом можно работать и с динамическими матрицами. Они объявляются как «массив массивов»:

```
var A: array of array of integer;
```

Для определения её размеров в процедуре `SetLength` нужно указать два параметра — количество строк и количество столбцов:

```
SetLength(A, 4, 3);
```

Функция `High` возвращает максимальный индекс строки (минимальный индекс всегда равен 0):

```
writeln(High(A));           (= 3)
```

Для определения границ изменения второго индекса (максимального номера столбца) нужно вызывать эту функцию для отдельной строки:

```
writeln(High(A[0]));        (= 2)
```

Использование в подпрограммах

Динамические массивы можно передавать как параметры подпрограмм (процедур и функций). Например, процедуру для вывода на экран целочисленного массива можно написать так:

```
procedure printArray(X: array of integer);
begin
  for i:=0 to High(X) do write(X[i], ' ');
end;
```

Динамические массивы можно передать в подпрограмму как изменяемые параметры (с помощью ключевого слова `var`). В этом

случае все изменения, сделанные в подпрограмме, применяются к массиву, переданному вызывающей программой, а не к его копии.

Расширение массива

Задача 2. С клавиатуры вводятся натуральные числа, ввод заканчивается числом 0. Нужно вывести на экран эти числа в порядке возрастания.

Как и в предыдущей задаче, для сортировки нужно предварительно сохранить все числа в оперативной памяти (в массиве). Но проблема в том, что размер этого массива становится известен только тогда, когда будут введены все числа. Что же делать?

В первую очередь приходит в голову такой вариант: при вводе каждого следующего ненулевого числа расширять массив на 1 элемент и записывать введённое число в последний элемент массива:

```
read(x);
while x<>0 do begin
  SetLength(A, Length(A)+1);
  A[High(A)]:=x;
  read(x)
end;
```

Здесь `x` — это целая переменная. К счастью, при таком расширении массива значения всех существующих элементов сохраняются.

Чем плох такой подход? Дело в том, что память в «куче» выделяется блоками. Поэтому при каждом увеличении длины массива последовательно выполняются три операции:

- 1) выделение блока памяти нового размера;
- 2) копирование в этот блок всех «старых» элементов;
- 3) удаление «старого» блока памяти из «кучи».

Видно, что «накладные расходы» очень велики, т. е. мы заставляем компьютер делать слишком много вспомогательной работы.

Ситуацию можно немного исправить, если увеличивать массив не каждый раз, а, скажем, после каждых 10 введённых элементов. То есть когда все свободные элементы массива заполнены, к нему добавляется ещё 10 новых элементов¹. При этом нужно считать фактическое количество записанных в массив значений, потому что определить их, как в предыдущей программе, через функцию `Length(A)`, будет невозможно:

¹ Можно также при каждом расширении увеличивать размер массива в два раза.

```

N:=0;
read(x);
while x<>0 do begin
  if N>High(A) then
    SetLength(A, Length(A)+10);
  A[N]:=x;
  N:=N+1;
  read(x);
end;

```

Здесь целая переменная N — это счётчик введённых чисел.

Теперь, когда все числа записаны в массив, можно отсортировать их любым известным методом, например методом пузырька или с помощью быстрой сортировки (эти алгоритмы изучались в 10 классе). Закончить программу вы можете самостоятельно.

Как это работает?

Чтобы грамотно применять динамические массивы, необходимо разобраться в том, как они работают. Для этого выведем на экран размер массива из 100 элементов (целых чисел):

```

SetLength(A, 100);
write(sizeof(A));
write(100*sizeof(integer));

```

Вы с удивлением обнаружите, что программа выводит числа 4 и 200, т. е. функция `sizeof` считает, что размер массива равен 4 байтам, хотя на самом деле 100 целых переменных должны занимать 200 байтов. Более того, величина `sizeof(A)` не зависит от фактического размера массива. Поэтому можно сделать вывод, что размер элементов тут вообще не учитывается.

На самом деле, в переменной A хранится *адрес* массива в памяти¹ (рис. 6.2), который действительно занимает 4 байта. Таким образом, фактически A — это *указатель*. Если массив имеет



Рис. 6.2

¹ Кроме того, непосредственно перед массивом хранится его фактический размер.

нулевой размер, этот указатель равен нулю (нулевой адрес в Паскале обозначается `nil`).

Что из этого следует? Представьте, например, что мы построили структуру, которая состоит из массива (поле *data*) и количества используемых элементов в нём (поле *size*):

```

type TArray=record
  data: array of integer;
  size: integer;
end;

```

Что будет, если мы попытаемся сохранить такую структуру в типизированном файле? Несложно понять, что в файл запишутся только элементы структуры, т. е. адрес массива *data* и количество элементов *size*. Сами элементы не входят в структуру, поэтому сохранены не будут. Если после этого мы прочитаем из файла такую структуру, адрес *data* будет недействителен и использовать его нельзя. Поэтому при сохранении в файле структур с динамическими полями нужно принимать специальные меры по сохранению содержимого массивов.

Кроме обычных (линейных) массивов, можно использовать и динамические многомерные массивы, в том числе матрицы (двумерные массивы).

Динамическая матрица — это указатель на массив указателей:

```
var A: array of array of integer;
```

Если применить к ней процедуру `SetLength` с одним параметром

```
SetLength(A, 10);
```

то в памяти будет выделен массив указателей на строки, причём память под сами строки не выделяется. То есть $A[1]$ (строка матрицы с индексом 1) — это указатель, к которому можно «привязать» динамический массив любого размера, например так:

```

for i:=0 to High(A) do
  SetLength(A[i], i+1);

```

Таким образом, мы получили матрицу, где все строки имеют разную длину:

```

writeln(Length(A[0]));  {= 1}
writeln(Length(A[9])); {= 10}

```



Вопросы и задания

1. Приведите примеры задач, в которых использование динамических массивов даёт преимущества (какие именно?).
2. Что такое динамические структуры данных? Где выделяется память под эти данные?
3. Как объявить в программе динамический массив и задать его размер?
4. Как расширить массив в ходе работы программы? Не потеряются ли при этом уже записанные в нём данные?
5. Как определить границы изменения индексов динамического массива? Нужно ли хранить его размер в отдельной переменной?
6. Как удалить массив из памяти?
7. Как разместить в памяти динамическую матрицу?
8. Как передать динамический массив в подпрограмму?
9. Какие проблемы могут возникнуть при сохранении динамических массивов и матриц в файлах? Как вы предлагаете их решать?



Подготовьте сообщение

- а) «Динамические массивы в языке Си»
- б) «Динамические массивы в языке Javascript»
- в) «Списки в языке Python как динамические массивы»



Задачи

1. Напишите полные программы для решения задач, рассмотренных в тексте параграфа.
2. Введите с клавиатуры число N и вычислите все простые числа в диапазоне от 2 до N , используя решето Эратосфена.
3. Введите с клавиатуры число N и запишите в массив первые N простых чисел.
4. Введите с клавиатуры число N и запишите в массив первые N чисел Фибоначчи (напомним, что они задаются рекуррентной формулой $F_n = F_{n-1} + F_{n-2}$, $F_1 = F_2 = 1$).
5. Напишите функцию, которая находит максимальный элемент переданного ей динамического массива.
6. Напишите подпрограмму, которая находит максимальный и минимальный элементы переданного ей динамического массива (используйте изменяемые параметры).
7. Напишите рекурсивную функцию, которая считает сумму значений элементов переданного ей динамического массива.
8. Напишите функцию, которая сортирует значения переданного ей динамического массива, используя алгоритм «быстрой сортировки» (см. главу 8 учебника для 10 класса).

§ 42

Списки

Что такое список?

Задача. В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Требуется построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова указано, сколько раз оно встречается в исходном файле.

Для решения задачи нам нужно составить *список*, в котором хранить пары «слово — количество». Список составляется по мере чтения файла, т. е. это динамическая структура.

Список — это упорядоченный набор элементов одного типа, для которого введены операции вставки (включения) и удаления (исключения).



Обычно используют **линейные списки**, в которых для каждого элемента (кроме первого) можно указать предыдущий, а для каждого элемента, кроме последнего, — следующий.

Вернёмся к нашей задаче построения алфавитно-частотного словаря. Алгоритм, записанный в виде псевдокода, может выглядеть так:

```

нц пока есть слова в файле
    прочитать очередное слово
    если оно есть в списке то
        увеличить на 1 счётчик для этого слова
    иначе
        добавить слово в список
        записать 1 в счётчик слова
все
кц
  
```

Теперь нужно записать все шаги этого алгоритма с помощью операторов языка программирования.

Использование динамического массива

В нашем случае каждый элемент списка должен содержать пару значений: слово (символьную строку) и счётчик этих слов (целое число). Поэтому элементы списка — это структуры, тип которых можно описать так:

```
type TPair = record
    word: string;      {слово}
    count: integer     {счётчик}
end;
```

В этой задаче размер массива становится известен только в конце работы программы. Поэтому требуется динамический массив, состоящий из описанных выше структур. С ним нужно выполнять следующие операции:

- искать заданное слово в списке;
- увеличивать счётчик заданного слова на 1;
- вставлять слово в определённое место списка (так, чтобы сохранить алфавитный порядок).

Как и в предыдущем параграфе, будем расширять размер массива сразу на 10 элементов¹, чтобы не выделять память слишком часто.

Объявим структуру-список:

```
type TWordList = record
    data: array of TPair; {динамический массив}
    size: integer          {количество элементов}
end;
```

Напомним, что количество фактически используемых элементов массива *size* может быть меньше, чем количество элементов, размещённых в памяти.

Введём переменную *L* типа *TWordList*:

```
var L: TWordList;
```

В начале основной программы очистим список и установим для него нулевую длину:

```
SetLength(L.data, 0);
L.size := 0;
```

¹ Возможны и другие варианты, например можно увеличивать размер массива в 2 раза.

Основной цикл (чтение данных из файла и построение списка) можно записать так (для последующего объяснения строки в теле цикла пронумерованы):

```
while not Eof(F) do begin
    readln(F, s);           {1}
    p := Find(L, s);        {2}
    if p >= 0 then          {3}
        Inc(L.data[p].count) {4}
    else begin              {5}
        p := FindPlace(L, s); {6}
        InsertWord(L, p, s)  {7}
    end                     {8}
end;
```

Здесь используются две вспомогательные переменные: символьная строка *s* (типа *string*) и целая переменная *p*. В строке 1 программы очередное слово читается из файла в строку *s*. Затем с помощью функции *Find* определяется, есть ли оно в списке (строка 2). Если есть (функция *Find* вернула существующий индекс), увеличиваем счётчик этого слова на 1 (строки 3–4) с помощью стандартной процедуры *Inc*.

Если в списке слова ещё нет (функция *Find* вернула -1), нужно найти место, куда его вставить, так чтобы не нарушился алфавитный порядок. Это делает функция *FindPlace*, которая должна возвращать номер элемента массива, *перед* которым нужно вставить прочитанное слово. Вставку выполняет процедура *InsertWord*.

Здесь встретилось обозначение с двумя точками: *L.data[p].count*. Вспомним, что *L* — это структура-список, у него есть поле-массив *data*. В этом массиве происходит обращение к элементу с номером *p*. Этот элемент — структура типа *TPair*, в составе которой есть поле *count*. Таким образом, *L.data[p].count* означает: «Поле *count* в составе *p*-го элемента массива *data*, который входит в структуру *L*».

Когда список готов, остаётся вывести его в выходной файл:

```
Assign(F, 'output.dat');
Rewrite(F);
for p := 0 to L.size - 1 do
    writeln(F, L.data[p].word, ': ', L.data[p].count);
Close(F);
```

Для каждого элемента списка в файл выводится хранящееся в нём слово и через двоеточие — сколько раз оно встретилось в тексте.

Таким образом, нам остаётся написать функции `Find` и `FindPlace`, а также процедуру `InsertWord`.

Функция `Find` принимает список и слово, которое нужно искать. Из курса 10 класса вы знакомы с двумя алгоритмами поиска: линейным и двоичным. Здесь для простоты будем использовать линейный поиск. В цикле проходим все элементы (не забывая, что их нумерация в динамическом массиве начинается с нуля). Как только очередное слово списка совпадёт с образцом, возвращаем в качестве результата функции номер этого элемента. Если просмотрены все элементы и совпадения не было, функция вернёт `-1`.

```
function Find(L: TWordList; word: string): integer;
var i: integer;
begin
    Find:=-1;
    for i:=0 to L.size-1 do
        if L.data[i].word=word then begin
            Find:= i;
            break
        end
    end;
end;
```

Функция `FindPlace` также принимает в качестве параметров список и слово. Она находит место вставки нового слова в список, при котором сохраняется алфавитный порядок расположения слов. Результат функции — номер слова, перед которым нужно вставить заданное. Для этого нужно найти в списке слово, которое «больше» заданного. Если такое слово не найдено, новое слово вставляется в конец списка:

```
function FindPlace(L: TWordList; word: string):
    integer;
var i, p: integer;
begin
    p:=-1;
    for i:=0 to L.size-1 do
        if L.data[i].word > word then begin
            p:=i;
            break
        end;
    end;
```

```
if p < 0 then p:=L.size;
FindPlace:=p
end;
```

Процедура `InsertWord` вставляет слово *word* в позицию *k* в список *L*:

```
procedure InsertWord(var L: TWordList; k: integer;
    word: string);
var i: integer;
begin
    IncSize(L);
    for i:=L.size-1 downto k+1 do
        L.data[i]:= L.data[i-1];
    L.data[k].word:= word;
    L.data[k].count:= 1
end;
```

Поскольку в список добавляется новый элемент, его размер увеличивается. Для этого введена процедура `IncSize`, которая вызывается в строке 1 (мы напишем её позже). Далее в цикле сдвигаем все последние элементы, включая элемент с номером *k*, на одну ячейку к концу массива (строки 2–3). Таким образом, элемент с номером *k* освобождается. В строке 4 в него записывается новое слово, а в строке 5 счётчик этого слова устанавливается равным 1.

Процедура `IncSize` увеличивает размер списка на 1 элемент. Когда нужный размер становится больше, чем размер динамического массива, массив расширяется сразу на 10 элементов.

```
procedure IncSize(var L: TWordList);
begin
    Inc(L.size);
    if L.size > Length(L.data) then
        SetLength(L.data, Length(L.data)+10)
end;
```

Процедура `IncSize` в программе должна располагаться выше вызывающей её процедуры `InsertWord`.

Приведём окончательную структуру программы:

```
program AlphaList;
{ объявления типов TPair и TWordList }
var F: text;
```

```

s: string;
L: TWordList;
p: integer;
{ процедуры и функции }
begin
  SetLength(L.data, 0);
  L.size := 0;
  Assign(F, 'input.dat');
  Reset(F);
  { основной цикл: составление списка слов }
  Close(F);
  { вывод результата в файл }
end.

```

Блоки, выделенные серым фоном, уже были написаны ранее в этом параграфе.

Заметим, что если известно максимальное количество разных слов в файле (скажем, не более 1000), то же самое можно сделать и на основе обычного (статического) массива, в котором память выделена заранее на максимальное число элементов.

Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (модуль). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями. В нашей программе в отдельный модуль можно вынести все операции со списком слов.

Модуль в языке Паскаль, в отличие от основной программы, начинается со слова `unit`, после которого ставится название модуля.

```

unit WordList;
interface
...
implementation
...
end.

```

В модуле два основных раздела: `interface` (интерфейс, общедоступная часть) и `implementation` (реализация, недоступная другим модулям). В разделе `interface` обычно размещают объявления типов данных, функций и процедур, а в разделе

`implementation` — программный код. В нашей программе модуль может выглядеть так:

```

unit WordList;
interface
  type TPair = record
    word: string;
    count: integer;
  end;
  TWordList = record
    data: array of TPair;
    size: integer;
  end;
  function Find(L: TWordList; word: string): integer;
  function FindPlace(L: TWordList;
    word: string): integer;
  procedure InsertWord(var L: TWordList; k: integer;
    word: string);
implementation
  { процедуры и функции }
end.

```

В секции `interface` мы расположили объявление типов данных, которые будут нужны основной программе, и заголовки подпрограмм этого модуля, которые могут вызываться извне. Всё, что находится в секции `implementation`, скрыто от «внешнего мира». В частности, там могут быть внутренние подпрограммы, которые «видны» только внутри модуля (в нашем случае это процедура `IncSize`).

Структура модуля в чём-то подобна айсбергу: видна только «надводная часть» (`interface`), а значительно более весомая «подводная часть» (`implementation`) скрыта. За счёт этого все, кто используют модуль, могут не думать о том, как именно он выполняет свою работу. Это один из приёмов, которые позволяют справляться со сложностью больших программ.

Модуль подключается к основной программе (или к другому модулю) с помощью ключевого слова `uses`. Если программа использует несколько модулей, все они перечисляются через запятую после слова `uses`.

Наша основная программа, использующая модуль `WordList`, выглядит так:

```

program AlphaList;
uses WordList; { подключение модуля }
var F: text;
    s: string;
    L: TWordList;
    p: integer;
begin
  {тело основной программы}
end.

```

Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других. Кроме того, такой подход ускоряет трансляцию больших программ, так как каждый модуль транслируется отдельно, причём только в том случае, если он был изменён.

Связные списки

Линейный список иногда представляется в программе в виде **связного списка**, в котором каждый элемент может быть размещён в памяти в произвольном месте, но должен содержать ссылку (указатель) на следующий элемент. У последнего элемента эта ссылка нулевая (в Паскале — `nil`), она показывает, что следующего элемента нет. Кроме того, нужно хранить где-то (в указателе `Head`) адрес первого элемента («головы») списка, иначе список будет недоступен (рис. 6.3).

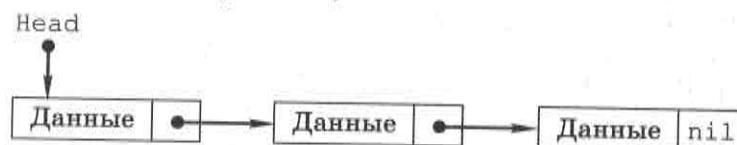


Рис. 6.3

Если замкнуть связный список в кольцо, так чтобы последний элемент содержал ссылку на первый, получается **циклический список** (рис. 6.4).

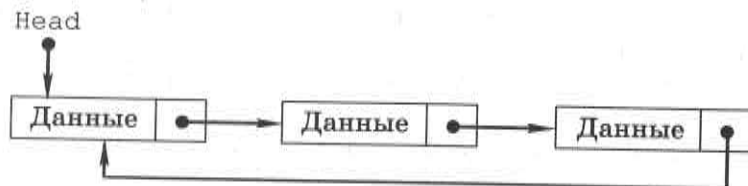


Рис. 6.4

Поскольку элементы связного списка содержат ссылки только на следующий элемент, к предыдущему перейти нельзя. Поэтому перебор возможен только в одном направлении. Этот недостаток устранён в **двусвязном списке**, где каждый элемент хранит адрес как следующего, так и предыдущего элемента (рис. 6.5).

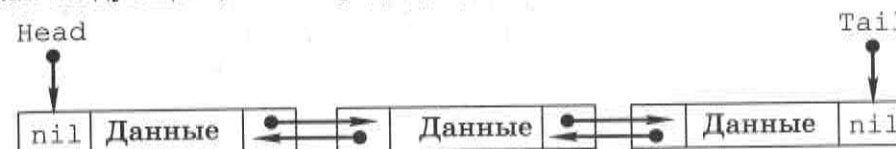


Рис. 6.5

Для такого списка обычно хранятся два адреса: «голова» списка (указатель `Head`) и его «хвост» (указатель `Tail`). Можно организовать и **циклический двусвязный список**. Использование двух указателей для каждого элемента приводит к дополнительному расходу памяти и усложнению всех операций со списком, потому что при добавлении и удалении элемента нужно правильно расставить оба указателя.

Преимущество связных списков состоит в том, что для них очень быстро выполняются операции вставки и удаления элементов, в то время как поиск нужного элемента требует перебора (прохода по списку).

Применение связных списков приводит к более сложным алгоритмам, чем работа с динамическими массивами; рассматривать соответствующие программы мы не будем.

Вопросы и задания

1. Что такое список? Какие операции он допускает?
2. Верно ли, что элементы в списке упорядочены?
3. Какой метод поиска в списке можно использовать? Обсудите разные варианты.
4. Как добавить элемент в линейный список, сохранив заданный порядок сортировки?
5. Как можно представить список в программе? В каких случаях для этого можно использовать обычный массив?
6. Объясните запись `L.data[i].word`.
7. Что такое модуль? Зачем используют модули?
8. Как оформляется текст модуля? Как по нему отличить модуль от основной программы?
9. Что размещается в секциях `interface` и `implementation`?

10. Можно ли все переменные и подпрограммы поместить в секцию `interface`? Чем это плохо?
11. Как подключается модуль к основной программе или другому модулю?
12. Что такое связный список?
13. Что такое циклический список? Попробуйте придумать задачу, где после завершения просмотра списка нужно начать просмотр заново.
14. Сравните односвязный и двусвязный списки. Покажите на примерах. В чём достоинства и недостатки одного и второго типов?

Подготовьте сообщение

- а) «Списки в языке Си»
- б) «Ассоциативные массивы в языке Javascript»
- в) «Словари в языке Python»

Задачи

1. Постройте программу, которая составляет алфавитно-частотный словарь для заданного файла со списком слов. Используйте модуль, содержащий все операции со списком.
- *2. В программе из задачи 1 измените функцию `Find` так, чтобы в ней использовался двоичный поиск.
3. В программе из задачи 2 объедините функции `Find` и `FindPlace`, заменив их на одну функцию. Если слово найдено в списке, функция работает так же, как `Find`: возвращает номер слова в списке. Если слово не найдено, функция должна вернуть отрицательное число: номер элемента массива, перед которым нужно вставить слово, со знаком минус.
- *4. В программе из задачи 3 выведите все найденные слова в файл в порядке убывания частоты, т. е. в начале списка должны стоять слова, которые встречаются в файле чаще всех.
- *5. Используя литературу и ресурсы Интернета, изучите технику работы со связными списками. Напишите программу, которая строит алфавитно-частотный словарь с помощью связных списков.

§ 43

Стек, очередь, дек

Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т. п.). С точки зрения информатики, её можно воспринимать как список элементов, расположенных в определённом порядке. Этот список имеет одну особенность — удалять и добавлять элементы можно только с одной («верхней») стороны. Действительно, для того

чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину тоже нельзя.

Стек (англ. *stack* — стопка) — это линейная структура данных, в которой элементы добавляются и удаляются только с одного конца (англ. **LIFO**: *Last In – First Out* — последним пришёл — первым ушёл).

На рисунке 6.6 показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка.

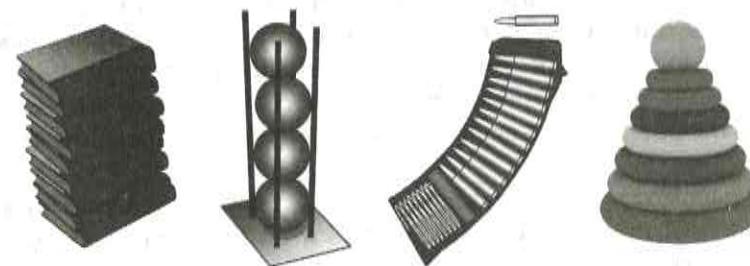


Рис. 6.6

Как вы знаете из главы 8 учебника для 10 класса, стек используется при выполнении программ: в этой специальной области оперативной памяти хранятся адреса возврата из подпрограмм, параметры, передаваемые функциям и процедурам, а также локальные переменные.

Задача 1. В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.

В этой задаче очень удобно использовать стек. Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* — втолкнуть);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* — вытолкнуть).

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
нц пока файл не пуст
    прочитать x
    добавить x в стек
кц
```

Теперь верхний элемент стека — это последнее число, прочитанное из файла. Поэтому остаётся «вытолкнуть» все записанные в стек числа, они будут выходить в обратном порядке:

```

нц пока стек не пуст
  вытолкнуть число из стека в x
  записать x в файл
кц

```

Использование динамического массива

Поскольку стек — это линейная структура данных с переменным количеством элементов, для создания стека в программе мы можем использовать динамический массив. Конечно, можно организовать стек из обычного (статического) массива, но его будет невозможно расширить сверх размера, выделенного при трансляции.

Для рассмотренной выше задачи 1 структура-стек содержит динамический целочисленный массив и количество используемых в нём элементов:

```

type TStack = record
  data: array of integer;
  size: integer
end;

```

Будем считать, что стек «растёт» от начала к концу массива, т. е. вершина стека — это последний элемент.

Для работы со стеком нужны две подпрограммы:

- процедура Push, которая добавляет новый элемент на вершину стека;
- функция Pop, которая возвращает верхний элемент стека и убирает его из стека.

Приведём эти подпрограммы:

```

procedure Push(var S: TStack; x: integer);
begin
  if S.size > High(S.data) then
    SetLength(S.data, Length(S.data) + 10);
  S.data[S.size] := x;
  S.size := S.size + 1;
end;

```

```

function Pop(var S: TStack): integer;
begin
  S.size := S.size - 1;
  Pop := S.data[S.size];
end;

```

Обратите внимание, что здесь структура типа TStack изменяется внутри подпрограмм, поэтому этот параметр должен быть изменяемым (описан с помощью var).

Заметим, что если нам понадобится стек, который хранит данные другого типа (например, символы, символьные строки или структуры), в объявлении типа и в приведённых подпрограммах нужно просто заменить **integer** на нужный тип.

Введём процедуру InitStack, которая заполняет поля структуры начальными значениями (выполняет инициализацию стека):

```

procedure InitStack(var S: TStack);
begin
  SetLength(S.data, 0);
  S.size := 0;
end;

```

Теперь несложно написать цикл ввода данных в стек из файла:

```

InitStack(S);
while not eof(F) do begin
  read(F, x);
  Push(S, x);
end;

```

Здесь *S* — переменная типа TStack; *F* — файловая переменная, связанная с файлом, открытым на чтение; *x* — целая переменная. Вывод результата в файл выполняется так:

```

while S.size > 0 do begin
  x := Pop(S);
  writeln(F, x);
end;

```

Здесь *i* — целая переменная, а *F* — файловая переменная, связанная с файлом, открытым на запись.

Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме: $(5+15)/(4+7-1)$?

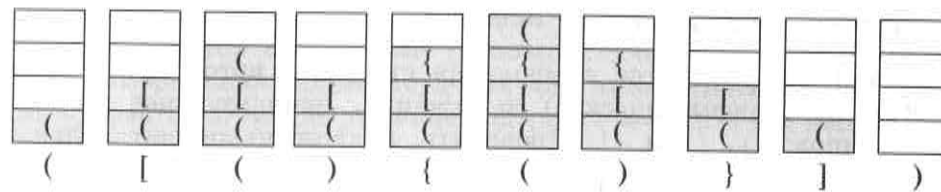


Рис. 6.8

Введём новый тип данных для работы со стеком:

```
type TStack=record
    data: array of char;
    size: integer
end;
```

Отличие этой структуры от стека из предыдущей задачи только в том, что он содержит не целые числа, а символы (типа `char`). Поэтому приводить подпрограммы `Push` и `Pop` мы не будем, вы можете их переделать самостоятельно. Для удобства добавим только логическую функцию, которая возвращает значение `True` (истина), если стек пуст:

```
function isEmpty(S: TStack): boolean;
begin
    isEmpty := (S.size=0)
end;
```

Введём строковые константы `L` и `R`, которые содержат все виды открывающих и соответствующих закрывающих скобок:

```
const L = '([{' ;
      R = ')]}' ;
```

Объявим переменные основной программы:

```
var S: TStack;
    p, i: integer;
    str: string;
    err: boolean;
    c: char;
```

Переменная `str` — это исходная строка. Логическая переменная `err` будет сигнализировать об ошибке. Сначала ей присваивается значение `False` («ложь»).

В основном цикле меняется целая переменная `i`, которая обозначает номер текущего символа, переменная `p` используется как вспомогательная:

```
for i:=1 to Length(str) do begin
    p:=Pos(str[i],L);           {1}
    if p>0 then Push(S,str[i]); {2}
    p:=Pos(str[i],R);           {3}
    if p>0 then begin           {4}
        if isEmpty(S) then err:=True {5}
        else begin
            c:=Pop(S);           {6}
            if p<>Pos(c,L) then err:=True {7}
        end;
        if err then break       {8}
    end
end;
```

Сначала мы ищем символ `str[i]` в строке `L`, т. е. среди открывающих скобок (строка 1). Если это действительно открывающая скобка, помещаем её в стек (2).

Далее ищем символ среди закрывающих скобок (3). Если нашли, то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная `err` принимает истинное значение.

Если в стеке что-то есть, снимаем символ с вершины стека в символьную переменную `c` (6). В строке (7) сравнивается тип (номер) закрывающей скобки `p` и номер открывающей скобки, найденной на вершине стека.

Если они не совпадают, выражение неправильное, и в переменную `err` записывается значение `True`.

Если при обработке текущего символа обнаружено, что выражение неверное (значение переменной `err` — `True`), цикл завершается досрочно с помощью оператора `break` (8).

После окончания цикла нужно проверить содержимое стека: если он не пуст, то в выражении есть незакрытые скобки, и оно ошибочно:

```
if not isEmpty(S) then err:=True;
```

В конце программы остаётся вывести результат на экран:

```
if not err
then writeln('Выражение правильное.')
else writeln('Выражение неправильное.');
```

Поскольку в этой задаче элементы стека — символы, решение можно значительно упростить, если использовать в качестве сте-

ка символьную строку. Попробуйте написать такой вариант программы самостоятельно.

Очереди, деки

Все мы знакомы с принципом очереди: первым пришёл — первым обслужен (англ. **FIFO**: *First In — First Out*). Соответствующая структура данных в информатике тоже называется очередью.

Очередь — это линейная структура данных, для которой введены две операции:

- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь — это не просто теоретическая модель. Операционные системы используют очереди для организации сообщений между программами: каждая программа имеет свою очередь сообщений. Контроллеры жёстких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создаётся очередь из пакетов данных, ожидающих отправки.

Задача 3. Рисунок задан в виде матрицы A , в которой элемент $A[y, x]$ определяет цвет пикселя (x, y) на пересечении строки y и столбца x . Требуется перекрасить в цвет 2 одноцветную область, начиная с пикселя (x_0, y_0) . На рисунке 6.9 показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой $(2, 1)$.

	1	2	3	4	5
1	0	1	0	1	1
2	1	1	1	2	2
3	0	1	0	2	2
4	3	3	1	2	2
5	0	1	1	0	0

$(2,1)$ $\rightarrow$

	1	2	3	4	5
1	0	2	0	1	1
2	2	2	2	2	2
3	0	2	0	2	2
4	3	3	1	2	2
5	0	1	1	0	0

Рис. 6.9

Эта задача актуальна для графических программ. Один из возможных вариантов решения использует очередь, элементы которой — координаты пикселей (точек):

добавить в очередь точку (x_0, y_0)
запомнить цвет начальной точки

```

нц пока очередь не пуста
  взять из очереди точку (x, y)
  если A[y, x] = цвету начальной точки то
    A[y, x] := 2;
    добавить в очередь точку (x-1, y)
    добавить в очередь точку (x+1, y)
    добавить в очередь точку (x, y-1)
    добавить в очередь точку (x, y+1)
  все
кц

```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы A). Заметим, что в этом алгоритме некоторые точки могут быть добавлены в очередь несколько раз (подумайте, когда это может случиться). Поэтому решение можно несколько улучшить, как-то помечая точки, уже добавленные в очередь, чтобы не добавлять их повторно (попробуйте сделать это самостоятельно).

Две координаты точки связаны между собой, поэтому в программе лучше объединить их в структуру **TPoint** (от англ. *point* — точка), а очередь составить из таких структур:

```

Type TPoint = record
  x, y: integer
end;
TQueue = record
  data: array of TPoint;
  size: integer
end;

```

Для удобства построим функцию **Point**, которая формирует структуру типа **TPoint** по заданным координатам:

```

function Point(x, y: integer): TPoint;
begin
  Point.x := x;
  Point.y := y
end;

```

Для работы с очередью, основанной на динамическом массиве, введём две подпрограммы:

- процедура **Put** добавляет новый элемент в конец очереди; если нужно, массив расширяется блоками по 10 элементов;
- функция **Get** возвращает первый элемент очереди и удаляет его из очереди (обработку ошибки «очередь пуста» вы мо-

жете сделать самостоятельно); все следующие элементы сдвигаются к началу массива.

```

procedure Put(var Q: TQueue; pt: TPoint);
begin
  if Q.size > High(Q.data) then
    SetLength(Q.data, Length(Q.data)+10);
  Q.data[Q.size]:=pt;
  Q.size:=Q.size+1;
end;
function Get(var Q:TQueue): TPoint;
var i: integer;
begin
  Get:=Q.data[0];
  Q.size:=Q.size-1;
  for i:=0 to Q.Size-1 do
    Q.data[i]:=Q.data[i+1]
end;

```

Остаётся написать основную программу. Объявляем константы и переменные:

```

const XMAX = 5; YMAX = 5;
      NEW_COLOR = 2;
var Q: TQueue;
    x0, y0, color: integer;
    A: array[1..YMAX,1..XMAX] of integer;
    pt: TPoint;

```

Предположим, что матрица *A* заполнена. Задаём исходную точку, с которой начинается заливка, запоминаем её «старый» цвет и добавляем эту точку в очередь:

```

x0:=2; y0:=1;
color:=A[y0,x0];
Put(Q, Point(x0,y0));

```

Основной цикл практически повторяет алгоритм на псевдокоде:

```

while not isEmpty(Q) do begin
  pt:=Get(Q);
  if A[pt.y,pt.x]=color then begin
    A[pt.y,pt.x]:=NEW_COLOR;
    if pt.x>1 then Put(Q, Point(pt.x-1,pt.y));
    if pt.x<XMAX then Put(Q, Point(pt.x+1,pt.y));
    if pt.y>1 then Put(Q, Point(pt.x,pt.y-1));
  end;
end;

```

```

if pt.y<YMAX then Put(Q, Point(pt.x,pt.y+1))
end
end;

```

Здесь функция *isEmpty* — такая же, как и для стека: возвращает *True*, если очередь пуста (поле *size* равно нулю).

В приведённом примере начало очереди всегда совпадает с первым элементом массива (имеющим индекс 0). При этом удаление элемента из очереди неэффективно, потому что требуется сдвинуть все оставшиеся элементы к началу массива. Существует и другой подход, при котором элементы очереди не передвигаются. Допустим, что мы знаем, что количество элементов в очереди всегда меньше *N*. Тогда можно выделить статический массив из *N* элементов (с индексами от 1 до *N*) и хранить в отдельных переменных номера первого элемента очереди («головы», англ. *head*) и последнего элемента («хвоста», англ. *tail*). На рисунке 6.10, а показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной *Head* (рис. 6.10, б).

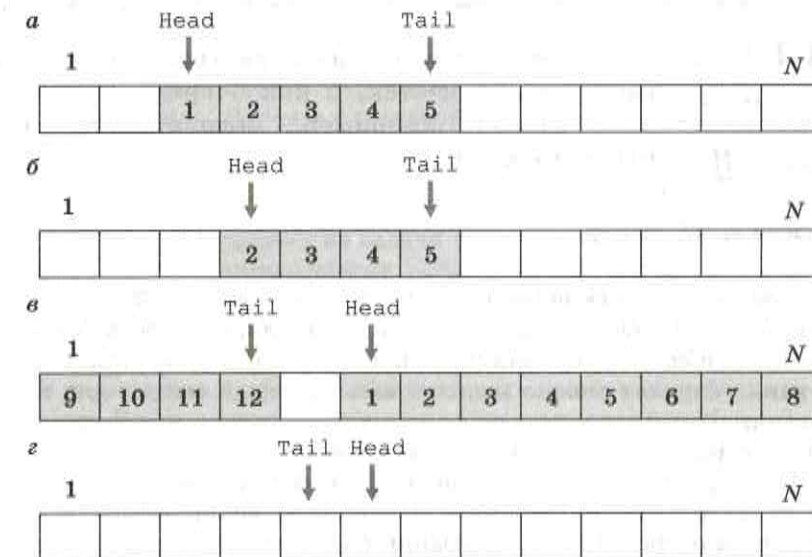


Рис. 6.10

При добавлении элемента в конец очереди переменная *Tail* увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной *Tail* присваивается значение 1. Таким образом, массив ока-