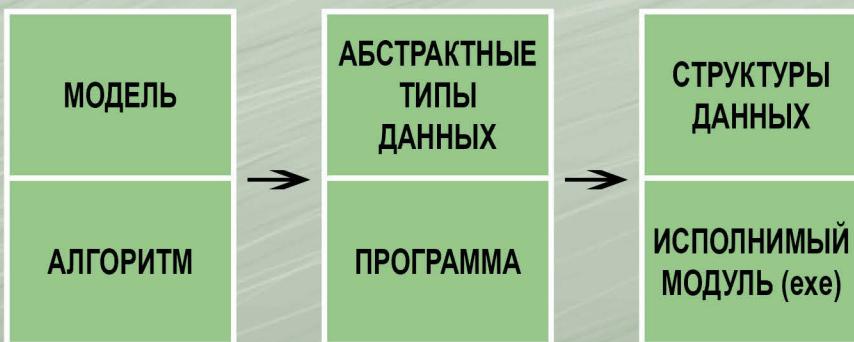


А.А. МОСКВИТИН

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ



МИНИСТЕРСТВО ОБРАЗОВАНИЯ СТАВРОПОЛЬСКОГО КРАЯ
ФИЛИАЛ ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО
ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«СТАВРОПОЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ ИНСТИТУТ»
В Г. ЕССЕНТУКИ

Москвитин А.А.

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ



РИА-КМВ
Пятигорск,
2021 г.

УДК: 519:517.12

ББК 22.3

М 822

*Печатается по решению
Совета Филиала СГПИ в г. Ессентуки*

Рецензенты:

И.М. Першин – доктор технических наук, профессор,
заведующий кафедрой систем управления и информационных
технологий ИСТИД СКФУ, филиал в г. Пятигорске.

Р.М. Ткаченко – кандидат физико-математических наук,
доцент кафедры физики и математики Пятигорского
медицинско-фармацевтического института – Филиала ГБОУ ВО
Волгоградского государственного медицинского университета

М 822

Структуры данных и алгоритмы. Учебное пособие для
проведения занятий по дисциплине «Программирование» со
студентами очной формы обучения / авт. А.А. Москвитин.
Пятигорск: РИА-КМВ; Ессентуки: СГПИ(ф), 2021. – 144 с.

ISBN 978-5-6046722-7-3

Учебное пособие рассчитано на студентов, специализирую-
щихся в области *computer science*, и особенно изучающих про-
граммирование.

Основу пособия составляют методы и средства работы с такими
понятиями, как данные и их структуры, а также методы и
средства разработки и оценки сложности алгоритмов.

Рассмотрены основные алгоритмы из различных разделов информатики.

УДК 519:517.12

ББК 22.3

ISBN 978-5-6046722-7-3

© Москвитин А.А., 2021

© Филиал СГПИ в г. Ессентуки, 2021.

© РИА-КМВ, 2021 г.

Оглавление

<i>Введение</i>	6
Данные.....	7
<i>Структуры данных</i>	7
Внутренние и внешние структуры данных	9
Простые базовые структуры	10
Числовые структуры	10
Символьные структуры	13
Логические структуры	15
Перечислимый тип	16
Интервалы.....	17
Указатели	18
<i>Статические структуры</i>	24
Вектор.....	24
Массивы	25
Множества	25
Записи.....	27
Таблицы.....	29
Обычные таблицы.....	30
Хэш-таблицы	30
Разрешение коллизий	32
Электронные таблицы	33
Таблицы решений	34
Таблицы в WEB-программировании.....	34
Мар	35
<i>Полустатические структуры</i>	35
Стеки	36
Очереди	37
Деки	38
Строки	39
<i>Динамические структуры</i>	40
Линейные связные списки	41
Связные списки	41
Разветвленные связные списки.....	42
Графы	43
Деревья	47
Префиксное дерево	49
<i>Файловые структуры.....</i>	50
Последовательные файлы.....	51

Файлы прямого доступа	53
Алгоритмические структуры.....	53
Вопросы для самоконтроля.....	54
Алгоритмы	56
От понятной задачи к алгоритму.....	57
Классификация алгоритмов	60
Стадии алгоритмизации	60
Разработка алгоритма.....	64
Понимание задачи.....	64
Выбор метода проектирования	65
Методы проектирования алгоритмов.....	65
Метод частных целей, подъема и обрабатывания назад	66
Эвристики	66
Программирование с отходом назад	66
Метод ветвей и границ	67
Рекурсия.....	68
Формы записи алгоритма.....	68
Аналитическая форма записи алгоритма	68
Описание алгоритма нахождения НОД делением	69
Графическая форма записи алгоритмов.....	70
Р-язык записи алгоритмов.....	72
Табличная форма записи алгоритма.....	74
Запись алгоритма в форме диаграммы.....	75
Псевдокод	76
Школьный алгоритмический язык.....	77
Исполнители алгоритмов	78
Некоторые именованные алгоритмы.....	81
Алгоритм Ньютона	81
Алгоритм Евклида – нахождение наибольшего общего делителя	82
Реализация алгоритма Евклида делением	82
Реализация алгоритма Евклида вычитанием	83
Алгоритм Дейкстры	84
Минимальное оставное дерево.....	90
Алгоритм Прима	91
Реализация алгоритма Прима	92
Алгоритм Крускала	94
Реализация алгоритма Крускала	95
Алгоритм Беллмана – Форда.....	97
Реализация алгоритма Форда-Беллмана	100
Алгоритм Хаффмана.....	100

Упражнения	103
Алгоритм Форда – Фалкерсона	103
Пример не сходящегося алгоритма	104
Сортировка Шелла	107
Алгоритм Брезенхэма	109
Алгоритм Эль-Гамаля	110
Алгоритм Гаусса – Лежандра	112
Генетические алгоритмы	113
Применение генетических алгоритмов	114
Методы тестирования алгоритмов	115
Анализ сложности алгоритма.....	116
Порядок роста.....	117
Константный – $O(1)$	117
Линейный – $O(n)$	118
Логарифмический – $O(\log n)$	118
Линеарифметический – $O(n \cdot \log n)$	118
Квадратичный – $O(n^2)$	119
Алгоритмически неразрешимые проблемы	119
Примеры анализа алгоритмов	123
Заключение	128
Вопросы для самопроверки.....	128
Приложения	131
Приложение 1	131
Пример проектирования алгоритмов.....	131
Пример спецификации осмысленной задачи	131
Приложение 2	136
Системы счисления	136
Перевод чисел из произвольной СС в десятичную и обратно.....	139
Перевод между основаниями, составляющими степень 2 ..	140
Приложение 3	142
Свойства логарифмов	142
Правила работы с суммами	142
Приложение 4	142
Языки программирования	143

ВВЕДЕНИЕ

Работа на компьютере требует от пользователя знаний и умений в области моделирования, алгоритмизации, выбора структур данных и языков программирования.

Любая работа пользователя начинается с того момента, когда он понял ту задачу, которую собирается решить с помощью компьютера.

При этом под задачей понимается неудовлетворенное чувство беспокойства.

Структура задачи состоит из описания входных данных, описания результатов вычислений, а также нечто такое, что позволит безошибочно отделять решения от нерешений. Заметим не ответов от не-ответов. Под этим *нечто* (критерием осмысленности задачи) будем понимать тройку:

1. *алгоритм*, реализуемый на компьютере;
2. *ограничения*, гарантирующие все решения задачи;
3. *исключительные ситуации*, определяющие все не-решения задачи.

Таким образом, понятную задачу в информатике можно представить следующей текстовой формулой

Понятная задача = Условие + Критерий осмысленности + Заключение

В свою очередь *критерий осмысленности* можно представить следующей формулой

$$\text{Критерий осмысленности} = \begin{cases} \text{Алгоритм} \\ \{ \text{Ограничения} \\ \} \text{ Исключительные ситуации} \end{cases}$$

Для задания входных данных и представления результатов вычислений требуется знания о структурах данных, которые позволяют по алгоритму разработать программу для компьютера.

Таким образом, общая схема решения задачи на компьютере (при знании структуры данных и алгоритмы) выглядит следующим образом.

Вначале человек вначале строит *модель*¹ предметной области (поскольку реальный объект имеет бесчисленное множество характеристик (параметров), не все из которых важны для данной задачи).

¹ Модель – абстракция реальности с наиболее ярко выраженными свойствами.

Затем он пытается разработать *алгоритм*¹ этой задачи, реализуемой на компьютере.

Разрабатывая алгоритм, мы обязаны учитывать структуры данных.

ДАННЫЕ

Что такое данные?

Данные – это представление фактов и идей в формализованном виде, пригодном для передачи или преобразовании в некотором процессе.

Данные в компьютере представляются некоторыми типами.

При реализации алгоритмов, понятие «структуре данных» тесно связано с понятием «типы данных». Любые данные, т.е. константы, переменные, значения функций или выражения, характеризуются своими типами.

Информация по каждому типу данных однозначно определяет:

- 1) *структуру хранения данных указанного типа*, т.е. выделение памяти и представление данных в ней, с одной стороны, и интерпретирование двоичного представления, с другой;
- 2) *множество допустимых значений*, которые может иметь тот или иной объект описываемого типа;
- 3) *множество допустимых операций*, которые применимы к объекту описываемого типа.

Таким образом, *структура данных* – это совокупность физически (типы данных) и логически (алгоритм, функции) взаимосвязанных переменных и их значений. Более подробно о структуре данных будет сказано далее.

СТРУКТУРЫ ДАННЫХ

Структура (запись) – конечное упорядоченное множество элементов, в общем случае имеющих различный тип. Элементы, входящие в состав структуры, называют полями структуры. Кажд-

¹ Алгоритм – понятное и точное предписание исполнителю совершить последовательность действий, направленных на достижение определенной цели или на решение поставленной задачи.

дое поле структуры имеет имя, по которому производится обращение к содержимому поля.

Пример структуры – совокупность сведений о некотором человеке: номер ИНН, фамилия, имя, отчество, дата рождения, место работы, трудовой стаж и т. п.

Структуры данных – это совокупность элементов данных и отношений между ними (рис. 1).

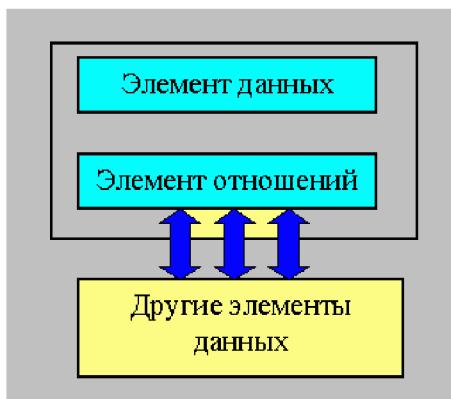


Рис. 1. Структуры данных

При этом под элементами данных может подразумеваться как простое данное, так и структура данных.

Элемент отношений – это совокупность всех связей элемента с другими элементами данных, рассматриваемой структуры.

Формально структуру данных можно представить следующим образом:

$S := (D, R)$,

где S – структура данных, D – данные и R – отношения.

Это общее понятие. Однако, в практике работы с данными обязательно учитываются особенности их организации и применения. Для этого принято классифицировать все существующие структуры данных по расположению и по типам взаимосвязей (рис. 2).



Рис. 2. Классификация структур данных

При этом важной характеристикой структур данных являются ее изменчивость, то есть изменение числа элементов и/или связей между составными частями структуры.

Одна из классификаций структур данных предполагает выделение внутренней и внешней структур данных. Разделение структур данных на внешние и внутренние чисто условное, поскольку они имеют одинаковый смысл и внутри и вне компьютера.

Внутренние и внешние структуры данных

Внутренняя структура данных определяет данные, расположенные в оперативной памяти компьютера, а *внешняя структура* – на внешних устройствах компьютера.

Обычно **внутренние структуры данных** надстраиваются над базовой структурой памяти машины (оперативной памяти) путем использования соответствующих программных средств.

Характерным для данных, размещенных на внешней памяти, является то, что они могут сохраняться там даже после завершения создавшей их программы и могут быть впоследствии много-кратно использованы той же или другой программой при их обработке.

Линейные и нелинейные структуры данных характеризуются взаимосвязями данных и будут рассмотрены далее.

Таким образом, имеет смысл рассмотреть только те основные структуры данных, с которыми работает программист, т.е. те структуры данных, которые нам потребуются при разработке алгоритмов и программ, а именно: простые базовые структуры; статические структуры; полустатические структуры; динамические структуры; файловые структуры и алгоритмические структуры (рис. 3).

Простые базовые структуры

К простым базовым структурам относятся:

- числовые структуры;
- символьные структуры;
- логические структуры;
- перечисления;
- интервалы;
- указатели.

Рассмотрим представленные на рис. 3 структуры данных подробнее.



Рис. 3. Структуры данных

Числовые структуры

Цифра (или Первоцифра) – одно из N имеющихся (в данной системе счисления) изображений уникальных символов, которые применимы для отображения только в одном разряде этой системы счисления.

Число – основное понятие математики, используемое для количественной характеристики, сравнения, нумерации объектов и их частей. Письменными знаками для обозначения чисел служат цифры, а также символы математических операций.

Число (в десятичной системе счисления) – это цифровая структура, элементами которой являются цифры (от 0 до 9), размещаемые в своей уникальной последовательности и на своих позициях (разрядах) в соответствии с принятой системой счисления.

Структура Числа – это некое исследуемое число, где элементами числа являются цифры, стоящие в присущей им последовательности, но, где, тем не менее, допускается осуществление различных манипуляций с имеющимися цифрами для углублённого постижения внутренних особенностей и свойств таких чисел, как в целом, так и в частностях.

Система счисления – это способ представления чисел и соответствующие ему правила действий над числами. Система счисления – это знаковая система, в которой числа записываются по определенным правилам с помощью символов некоторого алфавита, называемых цифрами.

Для представления чисел используются *непозиционные* и *позиционные* системы счисления. Система счисления называется *позиционной*, если одна и та же цифра имеет различное значение, которое определяется ее местом в числе. Десятичная система счисления является позиционной. *Например*: 999. Римская система счисления является *непозиционной*. Значение цифры X в числе XXI остается неизменным при вариации ее положения в числе. Количество различных цифр, употребляемых в позиционной системе счисления, называется *основанием* системы счисления.

Развернутая форма числа – это запись числа в позиционной системе счисления, которая представляют собой сумму произведений цифр числа на значение позиций.

$$\text{Например: } 8527_{10} = 8 \cdot 10^3 + 5 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0$$

Ниже приведены примеры форматов числовых типов данных.

Формат для представления чисел с плавающей точкой, приведенный на рис. 4 а, содержит поля мантиссы, порядка и знаков мантиссы и порядка фиксированной длины. Однако чаще вместо порядка используется характеристика, полученная путем прибавления к порядку смещения, так чтобы характеристика была всегда положительной. При этом имеет место формат представления вещественных чисел такой, как на рис. 4 б, в.



Рис. 4 а. Формат машинного представления беззнаковых чисел:
а – тип byte; б – тип word

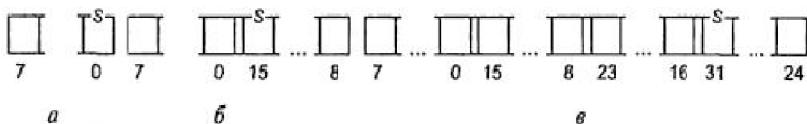


Рис. 4 б. Формат машинного представления чисел со знаком:
а – shortint; б – integer; в – longint

здесь s -знаковый бит числа. При этом, если $s=0$, то число положительное, если $s=1$ – число отрицательное. Цифры определяют номера разрядов памяти.

Знак числа	Порядок	Знак порядка	Мантисса
------------	---------	--------------	----------

Знак числа	Характеристика	Мантисса
------------	----------------	----------

Рис. 4 в. Формат представления вещественного числа с порядком и характеристикой

Формулы для вычисления характеристики и количество бит, необходимых для ее хранения, приведены в следующей таблице 1.

Таблица 1

Тип	Характеристика	Кол-во бит на хар-ку
real	$x = 2^7 + p + 1$	8
single	$x = 2^7 + p - 1$	8
double	$x = 2^{10} + p - 1$	11
extended	$x = 2^{14} + p - 1$	15

В разных языках программирования простые типы отображаются различными форматами. Так, *например*, в языке PASCAL это выглядит следующим образом (рис. 5).



Рис. 5. Структуры простых типов PASCAL

Символьные структуры

Символьная (текстовая) информация – самый простой тип данных с точки зрения его представления в памяти ЭВМ. Каждому символу текста в памяти соответствует байт с 8-разрядным кодом этого символа в том или ином стандарте. Буквам латинского алфавита, цифрам, знакам операций и различным разделителям (скобки, точки, запятые и т. п.) в некотором смысле повезло больше, т. к. их кодировка практически универсальна. Она предложена фирмой IBM и составляет первую половину большинства 8-разрядных кодировочных таблиц, используемых в разных странах. В терминологии IBM PC такие таблицы принято называть кодовыми страницами. Вторая половина кодовых страниц отведена под национальную символику.

Так представленная кодировка символов обеспечивает практически все потребности в программировании, т.к. одним байтом ($2^8 = 256$ символам) можно закодировать все элементы, размещенные на клавиатуре, она является основным устройством ввода данных.

В отечественной практике чаще всего приходится иметь дело либо с символикой MS-DOS в стандарте ASCII (American Standard Code for Information Interchange), либо с одной из кодовых страниц ANSI (American National Standard Institute), применяемых в среде Windows. Между этими таблицами существует принципиальное различие, связанное с кодировкой букв русского алфавита.

В таблице *ASCII* (кодовая страница 866) заглавные русские буквы начинаются со 128-й позиции. Вплотную вслед за ними располагается цепочка строчных букв от буквы а (код – 160) до буквы п (код – 175). Далее следуют символы псевдографики, используемые для формирования таблиц с одинарными и двойными линиями (диапазон кодов от 176 до 223). Начиная с 224-й позиции располагаются остальные буквы от р до я. И наконец, вслед за ними, выбиваясь из алфавитного порядка, размещены буквы ё (код -240) и ё (код – 241).

В таблице *ANSI* (кодовая страница 1251) коды русских букв начинаются с позиции 192 (код буквы А) и расположены сплошным интервалом до 255-й позиции (код буквы я). Символы псевдографики в графической оболочке Windows смысла не имеют и поэтому в таблице ANSI отсутствуют. Буквы ё и ё здесь тоже не включены в общепринятую алфавитную последовательность и имеют, соответственно, коды 168 и 184. Более того, их обозначение на большинстве русифицированных клавиатур отсутствует, и для включения таких букв в набираемый текст приходится нажимать самую левую клавишу в верхнем ряду (на латинском регистре этой клавише соответствует символ «~»).

Все достаточно развитые алгоритмические языки включают серию процедур по обработке символьных (каждый объект – отдельный символ) и строковых (цепочки символов) данных. В системе QBasic текстовые данные находятся в символьных переменных, имена которых либо заканчиваются знаком \$, либо начинаются с одной из букв, упомянутых в объявлении вида DEFSTR c-J, либо описаны в операторе DIM сочетаниемAS STRING. Базовым текстовым элементом в Си являются объекты типа char, значением каждого из которых является единственный символ. Объединение таких объектов в одномерный массив позволяет работать с цепочками символов. PASCAL обеспечивает работу как с

одиночными символами (данные типа `char`), так и с их последовательностями (данные типа `string`).

Каждый из рассматриваемых языков обеспечивает возможность доступа к отдельному символу строки.

В языке Си это вытекает из способа представления текстовой информации. Начальное присвоение (`char name [5] = «Вася»;`) или копирование одной строки в другую (`strcpy (name, «Вася»);`) эквивалентно 5 обычным операторам присваивания:

```
name[0] = 'В';
name[1]= 'а';
name[ 2 ] = 'с';
name[ 3 ] = 'я';
name[4]= "\0";
```

Иногда символ, не имеющий графического эквивалента (например, пробел) называют *литерой*. Следует помнить, что *литера* – это старинное название буквы.

Логические структуры

Значениями логического типа `BOOLEAN` может быть одна из предварительно объявленных констант *false* (ложь) или *true* (истина).

Данные логического типа занимают один байт памяти. При этом значению *false* соответствует нулевое значение байта, а значению *true* соответствует любое ненулевое значение байта. Например: *false* всегда в машинном представлении: 00000000; *true* может выглядеть таким образом: 00000001 или 00010001 или 10000000.

Однако следует иметь в виду, что при выполнении операции присваивания переменной логического типа значения *true*, в соответствующее поле памяти всегда записывается код 00000001.

Над логическими типами возможны операции булевой алгебры – НЕ (*not*), ИЛИ (*or*), И (*and*), исключающее ИЛИ (*xor*) – последняя реализована для логического типа не во всех языках. В этих операциях операнды логического типа рассматриваются как единое целое – вне зависимости от битового состава их внутреннего представления.

Кроме того, следует помнить, что результаты логического типа получаются при сравнении данных любых типов.

Интересно, что в языке С данные логического типа отсутствуют, их функции выполняют данные числовых типов, чаще всего – типа *int*. В логических выражениях операнд любого числового типа, имеющий нулевое значение, рассматривается как «ложь», а ненулевое – как «истина». Результатами логического типа являются целые числа 0 (ложь) или 1 (истина).

Перечислимый тип

Перечислимый тип представляет собой упорядоченный тип данных, определяемый программистом, т.е. программист перечисляет все значения, которые может принимать переменная этого типа. Значения являются неповторяющимися в пределах программы идентификаторами, количество которых не может быть больше 256, например,

```
type color=(red,blue,green);
work_day=(mo,tu,we,th,fr);
winter_day=(december,january,february);
```

Для переменной перечислимого типа выделяется один байт, в который записывается порядковый номер присваиваемого значения. Порядковый номер определяется из описанного типа, причем нумерация начинается с 0. Имена из списка перечислимого типа являются константами, например,

```
var B,C:color;
begin B:=blue; (* B=1 *)
      C:=green; (* C=2 *)
      Write(ord(B):4,ord(C):4); end.
```

После выполнения данного фрагмента программы на экран будут выданы цифры 1 и 2. Содержимое памяти для переменных B и C при этом следующее:

B – 00000001; C – 00000010.

На физическом уровне над переменными перечислимого типа определены операции создания, уничтожения, выбора, обновле-

ния. При этом выполняется определение порядкового номера идентификатора по его значению и, наоборот, по номеру идентификатора – его значение.

На логическом уровне переменные перечислимого типа могут быть использованы только в выражениях булевского типа и в операциях сравнения; при этом сравниваются порядковые номера значений.

Интервалы

Одним из способов образования новых типов из уже существующих является *ограничение допустимого диапазона значений* некоторого стандартного скалярного типа или рамок описанного перечислимого типа. Это ограничение определяется заданием минимального и максимального значений диапазона. При этом изменяется диапазон допустимых значений по отношению к базовому типу, но представление в памяти полностью соответствует базовому типу.

При машинном представлении это выглядит следующим образом.

Данные интервального типа могут храниться в зависимости от верхней и нижней границ интервала независимо от входящего в этот предел количества значений в виде, представленном в таблице 2. Для данных интервального типа требуется память размером один, два или четыре байта, например,

```
var A: 220..250; (* Занимает 1 байт *)
  B: 2221..2226; (* Занимает 2 байта *)
  C: 'A'..'K'; (* Занимает 1 байт *)
begin A:=240; C:='C'; B:=2222; end.
```

Таблица 2

Базовый тип	Максимально допустимый диапазон	Размер требуемой памяти
ShortInt	-128..127	1 байт
Integer	-32768..32767	2 байта
LongInt	-2147483648..2147483647	4 байта

Byte	0..255	1 байт
Word	0..65535	2 байта
Char	chr(ord(0))..chr(ord(255))	1 байт
Boolean	False..True	1 байт

После выполнения данной программы содержимое памяти будет следующим:

A – 11110000; C – 01000011; B – 10101110 00001000.

На физическом уровне над переменными *интервального типа* определены операции *создания, уничтожения, выбора, обновления*. Дополнительные операции определены базовым типом элементов интервального типа.

Примечание: запись chr(ord(0)) в таблице 2 следует понимать как: символ с кодом 0.

А) Интервальный тип от символьного: определение кода символа и, наоборот, символа по его коду.

Пусть задана переменная типа tz:'d'..'h'. Данной переменной присвоено значение 'e'. Байт памяти, отведенный под эту переменную, будет хранить ASCII-код буквы 'e' т.е. 01100101 (в 10-ом представлении 101).

Б) Интервальный тип от перечислимого: определение порядкового номера идентификатора по его значению и, наоборот, по номеру идентификатора – его значение.

На логическом уровне все операции, разрешенные для данных базового типа, возможны и для данных соответствующих интервальных типов.

Указатели

Тип указателя представляет собой адрес ячейки памяти (в подавляющем большинстве современных вычислительных систем размер ячейки – минимальной адресуемой единицы памяти – составляет один байт). При программировании на низком уровне – в машинных кодах, на языке Ассемблера и на языке С, который специально ориентирован на системных программистов, работа с адресами составляет значительную часть программных кодов. При решении прикладных задач с использованием языков высокого уровня (C++, Pascal, Visual Basic и др.) указатели не используются.

кого уровня наиболее частые случаи, когда программисту могут понадобиться следующие указатели:

- 1) При необходимости представить одну и ту же область памяти, а следовательно, одни и те же физические данные, как данные разной логической структуры. В этом случае в программе вводятся два или более указателей, которые содержат адрес одной и той же области памяти, но имеют разный тип (см. далее). Обращаясь к этой области памяти по тому или иному указателю, программист обрабатывает ее содержимое как данные того или иного типа.
- 2) Еще более важным является применение указателей при работе с динамическими структурами данных. Память под такие структуры выделяется в ходе выполнения программы, стандартные процедуры/функции выделения памяти возвращают адрес выделенной области памяти – указатель на нее. К содержимому динамически выделенной области памяти программист может обращаться только через такой указатель.

Физическое представление адреса существенно зависит от аппаратной архитектуры вычислительной системы.

Рассмотрим в качестве примера структуру адреса в микропроцессоре i8086.

Машинное слово этого процессора имеет размер 16 двоичных разрядов. Если использовать представление адреса в одном слове, то можно адресовать 64 Кбайт памяти, что явно недостаточно для сколько-нибудь серьезного программного изделия. Поэтому адрес представляется в виде двух 16-разрядных слов – сегмента и смещения. Сегментная часть адреса загружается в один из специальных сегментных регистров (в i8086 таких регистров 4). При обращении по адресу задается идентификатор сегментного регистра и 16-битное смещение. Полный физический (эффективный) адрес получается следующим образом. Сегментная часть адреса сдвигается на 4 разряда влево, освободившиеся слева разряды заполняются нулями, к полученному таким образом коду прибавляется смещение, как показано на рис. 6.

Сегмент	0111 0010 1111 0011 0000
+	
Смещение	0001 1001 1110 0110
=	
Полный физический адрес	0111 0100 1001 0001 0110

Рис 6. Вычисление полного адреса в микропроцессоре i8086

Полученный эффективный адрес имеет размер 20 двоичных разрядов, таким образом, он позволяет адресовать до 1 Мбайт памяти.

Еще раз повторим, что физическая структура адреса принципиально различна для разных аппаратных архитектур. Так, например, в микропроцессоре i386 обе компоненты адреса 32-разрядные; в процессорах семейства S/390 адрес представляется в виде 31-разрядного смещения в одном из 19 адресных пространств, в процессоре Power PC 620 одним 64-разрядным словом может адресоваться вся как оперативная, так и внешняя память.

Операционная система MS DOS была разработана именно для процессора i8086 и использует описанную структуру адреса даже, когда выполняется на более совершенных процессорах. Однако, это сегодня единственная операционная система, в среде которой программист может работать с адресами в реальной памяти и с физической структурой адреса. Все без исключения современные модели процессоров аппаратно выполняют так называемую динамическую трансляцию адресов и совместно с современными операционными системами обеспечивают работу программ в виртуальной (каждущейся) памяти. Программа разрабатывается и выполняется в некоторой виртуальной памяти, адреса в которой линейно изменяются от 0 до некоторого максимального значения. Виртуальный адрес представляет собой число – номер ячейки в виртуальном адресном пространстве. Преобразование виртуального адреса в реальный производится аппаратно при каждом обращении по виртуальному адресу. Это преобразование выполняется совершенно незаметно (прозрачно) для программиста, по-

этому в современных системах программист может считать физической структурой адреса структуру виртуального адреса. Виртуальный же адрес представляет собой целое число без знака. В разных вычислительных системах может различаться разрядность этого числа. Большинство современных систем обеспечивают 32-разрядный адрес, позволяющий адресовать до 4 Гбайт памяти, но уже существуют системы с 48 и даже 64-разрядными адресами.

В программе на языке высокого уровня указатели могут быть типизированными и нетипизированными.

При объявлении типизированного указателя определяется и тип объекта в памяти, адресуемого этим указателем. Так, например, объявления в языке PASCAL:

```
Var ipt : ^integer; cpt : ^char;
```

или в языке С:

```
int *ipt; char *cpt;
```

означают, что переменная *ipt* представляет собой адрес области памяти, в которой хранится целое число, а *cpt* – адрес области памяти, в которой хранится символ. Хотя физическая структура адреса не зависит от типа и значения данных, хранящихся по этому адресу, компилятор считает указатели *ipt* и *cpt* имеющими разный тип, и в Pascal оператор:

```
cpt := ipt;
```

будет расценен компилятором как ошибочный (компилятор С для аналогичного оператора присваивания ограничивается предупреждением). Таким образом, когда речь идет об указателях типизированных, правильнее говорить не о едином типе данных «указатель», а о целом семействе типов: «указатель на целое», «указатель на символ» и т.д. Могут быть указатели и на более сложные, интегрированные структуры данных, и указатели на указатели.

Нетипизированный указатель – тип *pointer* в Pascal или *void ** в С – служит для представления адреса, по которому содержатся данные неизвестного типа. Работа с нетипизированными указателями существенно ограничена, они могут использоваться только для сохранения адреса, обращение по адресу, задаваемому нетипизированным указателем, невозможно.

Основными операциями, в которых участвуют указатели, являются *присваивание, получение адреса, выборка*.

Присваивание является двухместной операцией, оба операнда которой – указатели. Как и для других типов, операция присваивания копирует значение одного указателя в другой, в результате оба указателя будут содержать один и тот же адрес памяти. Если оба указателя, участвующие в операции присваивания типизированные, то оба они должны указывать на объекты одного и того же типа.

Операция *получения адреса* – одноместная, ее operand может иметь любой тип, результатом является типизированный (в соответствии с типом operand) указатель, содержащий адрес объекта-operand.

Операция *выборки* – одноместная, ее operandом является типизированный (обязательно!) указатель. Результат – данные, выбранные из памяти по адресу, заданному operandом. Тип результата определяется типом указателя-operand.

Перечисленных операций достаточно для решения задач прикладного программирования, поэтому набор операций над указателями, допустимых в языке PASCAL, этим и ограничивается. Системное программирование требует более гибкой работы с адресами, поэтому в языке Си доступны также операции адресной арифметики, которые описываются ниже.

К указателю можно прибавить целое число или вычесть из него целое число. Поскольку память имеет линейную структуру, прибавление к адресу числа даст нам адрес области памяти, смешенной на это число байт (или других единиц измерения) относительно исходного адреса. Результат операций «указатель + целое», «указатель – целое» имеет тип «указатель».

Можно вычесть один указатель из другого (оба указателя-operand при этом должны иметь одинаковый тип). Результат такого вычитания будет иметь тип целого числа со знаком. Его значение показывает, на сколько байт (или других единиц измерения) один адрес отстоит от другого в памяти.

Отметим, что сложение указателей не имеет смысла. Поскольку программа разрабатывается в относительных адресах и при разных своих выполнениях может размещаться в разных областях

памяти, сумма двух адресов в программе будет давать разные результаты при разных выполнениях. Смещение же объектов внутри программы друг относительно друга не зависит от адреса загрузки программы, поэтому результат операции вычитания указателей будет постоянным, и такая операция является допустимой.

Операции адресной арифметики выполняются только над типизированными указателями. Единицей измерения в адресной арифметике является размер объекта, который указателем адресуется. Так, если переменная `ipt` определена как указатель на целое число (`int *ipt`), то выражение `ipt+1` даст адрес, больший не на 1, а на количество байт в целом числе (в MS DOS – 2). Вычитание указателей также дает в результате не количество байт, а количество объектов данного типа, помещающихся в памяти между двумя адресами. Это справедливо как для указателей на простые типы, так и для указателей на сложные объекты, размеры которых составляют десятки, сотни и более байт.

В связи с имеющимися в языке Си расширенными средствами работы с указателями, следует упомянуть и о разных представлениях указателей в этом языке. В Си указатели любого типа могут быть близкими (*near*) и дальними (*far*) или (*huge*). Эта дифференциация связана с физической структурой адреса в i8086, которая была рассмотрена выше. Близкие указатели представляют собой смещение в текущем сегменте, для представления такого указателя достаточно одного 16-разрядного слова.

Дальние указатели представляются двумя 16-разрядными словами – сегментом и смещением. Разница между *far* или *huge* указателями состоит в том, что:

- для первых адресная арифметика работает только со смещением, не затрагивая сегментную часть адреса, таким образом, операции адресной арифметики могут изменять адрес в диапазоне не более 64 Кбайт;
- для вторых – в адресной арифметике участвует и сегментная часть, таким образом, предел изменения адреса – 1 Мбайт.

Впрочем, это различие в представлении указателей имеется только в системах программирования, работающих в среде MS DOS, в современных же операционных системах, поддерживаю-

щих виртуальную адресацию, различий между указателями нет, все указатели можно считать гигантскими.

Статические структуры

Статические структуры представляют собой структурированное множество примитивных структур. Например, вектор может быть представлен упорядоченным множеством чисел.

Изменчивость не свойственна статическим структурам, т. е. размер памяти компьютера, отводимый для таких данных, постоянен и выделяется на этапе компиляции или выполнения программы.

Вектор

С логической точки зрения *вектор*, как *статическая структура* (одномерный массив) представляет собой структуру данных с фиксированным числом элементов одного и того же типа. Каждый элемент вектора:

- имеет свой уникальный номер (индекс). Обращение к элементу вектора выполняется по имени вектора и номеру элемента. С физической точки зрения элементы вектора размещаются;
- в памяти в подряд расположенных ячейках памяти (рис. 7).

Под элемент вектора выделяется количество байт памяти, определяемое базовым типом элемента этого вектора. Тогда размер памяти, отводимой для размещения вектора, будет определяться следующим соотношением: $S = k * \text{Sizeof}(\text{Тип})$, где k – количество элементов (длина) вектора, а $\text{Sizeof}(\text{Тип})$ – размер памяти, необходимой для хранения одного элемента вектора.

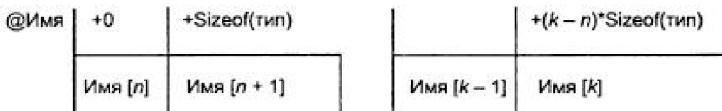


Рис. 7. Представление вектора памяти.
@Имя – адрес вектора или адрес первого элемента

Массивы

Массив – это тоже статическая структура данных с фиксированным и упорядоченным набором однотипных элементов (компонентов).

Доступ к какому-либо из элементов массива осуществляется по имени и номеру (индексу) этого элемента. Количество индексов определяет размерность массива. Так, *например*, чаще всего встречаются одномерные (вектора) и двумерные (матрицы) массивы. Первые имеют один индекс, вторые – два.

Двумерный массив (матрица) – это вектор, каждый элемент которого вектор. Поэтому то, что справедливо для вектора, справедливо и для матрицы (аналогично для n -мерных массивов).

Обязательный критерий *статического массива* – это однородность данных, единовременно хранящихся в нем. Когда же данное условие не выполняется, то массив является *гетерогенным*.

Множества

Множеством является статическая структура, представляющая собой набор неповторяющихся данных одного и того же типа. Множество может принимать все значения базового типа. Поскольку базовый тип не должен превышать 256 возможных значений, типом элементов множества могут быть *byte*, *char* и производные от них типы.

Множества хранят данные без определенного порядка и без повторяющихся значений. Помимо возможности *добавления* и *удаления* элементов, есть несколько других важных функций, которые работают с двумя наборами одновременно. Рассмотрим их.

- *Union* (Объединение). Объединяет все элементы из двух разных множеств и возвращает результат, как новый набор (без дубликатов).
- *Intersection* (Пересечение). Если заданы два множества, эта функция вернет другое множество, содержащее элементы, которые имеются и в первом и во втором множестве.

- *Difference* (Разница). Вернет список элементов, которые находятся в одном множестве, но НЕ повторяются в другом.
- *Subset* (Подмножество) – возвращает булево значение, показывающее, содержит ли одно множество все элементы другого множества.

Множество в памяти (рис. 8) хранится как массив битов, в котором каждый бит указывает, является ли элемент принадлежащим объявленному множеству или нет. Таким образом, максимальное число элементов множества 256, а данные типа множество могут занимать не более 32 байт.

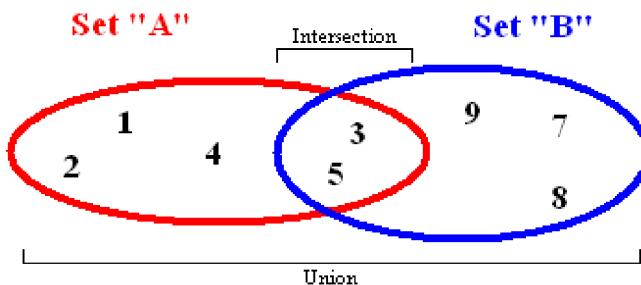


Рис. 8. Множества и операции над ними

Представление множества в памяти компьютера показано на рис. 9.

Смещение (байт)	Представление байта в машинной памяти (номера разрядов)																
@S + 0	<table border="1"> <tr> <td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td> </tr> </table>	7	6	5	4	3	2	1	0								
7	6	5	4	3	2	1	0										
@S + 31	<table border="1"> <tr> <td>255</td><td>254</td><td>253</td><td>252</td><td>251</td><td>250</td><td>249</td><td>248</td> </tr> <tr> <td>7</td><td></td><td></td><td></td><td></td><td></td><td></td><td>0</td> </tr> </table>	255	254	253	252	251	250	249	248	7							0
255	254	253	252	251	250	249	248										
7							0										

Рис. 9. Представление множества в памяти:
@S – адрес данного типа множества

Размер памяти (в байтах), выделяемых под множество, вычисляется по формуле: $S - (\max \text{ div } 8) - (\min \text{ div } 8) + 1$, где \max и \min – верхняя и нижняя границы базового типа данного множества, а div – целочисленное деление.

Множество как структура уровня представления является совокупностью различных объектов, которые могут либо сами являться множествами, либо представлять собой неделимые элементы, называемые атомами.

Множество как структура уровня хранения реализуется двумя способами:

- *простым* – в виде данных перечислимого типа; в языках высокого уровня этот тип данных реализуют с помощью типа «множество» (в Pascal) или констант перечислимого типа (в Си);
- *сложным* – в виде вектора или связного списка.

Отличие этих двух способов в том, что данные перечислимого типа ограниченно поддерживают понятие множества, представляя собой совокупность объектов, которым сопоставлены некоторые значения.

При реализации множеств с помощью вектора или связного списка становится возможным реализовать именно *математическое понятие множества*, согласно которому, над множествами определен ряд операций: объединение, пересечение, разность, слияние, вставка (удаление) элемента и т. д. С данными перечислимого типа эти операции невозможны.

Записи

Довольно часто вполне оправданным является представление некоторых элементов данных в качестве составных частей другой, более крупной логической единицы. Такой логической единицей может служить *запись*.

Запись – структура данных, состоящая из фиксированного числа компонент, называемых полями, каждая из которых может иметь свой тип. Записи позволяют в удобной форме представлять ведомости, таблицы, картотеки, каталоги и другие данные.

Запись – это комбинированный тип, значения которого представляют собой нетривиальную структуру данных. Они состоят

из нескольких полей разного типа, доступ к которым осуществляется по их именам.

Записи представляют собой средство для представления программных моделей реальных объектов предметной области, так как каждый такой объект обладает некоторыми свойствами, которые могут описываться данными различных типов.

Пример записи – набор сведений о сотруднике кафедры.

Объект (запись) «сотрудник» может обладать следующими свойствами:

- табельный номер – целое положительное число;
- фамилия-имя-отчество – строка символов и т. д.;
- пол – символ;
- ученая степень – строка символов;
- заработка плата – вещественное число;
- и др.

В памяти эта структура может быть представлена в одном из двух видов:

- в виде *последовательности полей*, занимающих непрерывную область памяти (рис. 10 а). Чтобы получить доступ к любому элементу записи, нужно знать адрес начала записи и смещение относительно начала. При этом достигается экономия памяти компьютера, но затрачивается лишнее время на вычисление адресов полей;
- в виде *связного списка* с указателями на значения полей записи. При такой организации имеет место быстрый доступ к элементам, но очень неэкономичный расход памяти для хранения (рис. 10 а, б).

@rec	+0	+1	+21	+29	+37	+38	+39
24	Иванов В. И.	АП	54	4	5	5	

Рис. 10 а. Представление в памяти переменной типа запись
в виде последовательности полей

rec	student	7	
byte	1	num	→
string	21	name	→
string	8	fac	→
string	8	group	→
byte	1	math	→
byte	1	comp	→
byte	1	lang	→

Указатели значений полей записи

Рис. 10 б. Размещение в памяти переменной типа запись в виде указателей

Приведем пример описания переменной в языке Pascal, имеющей структуру записи:

```

Var
  Address : Record
    HouseNumber : Integer;
    StreetName : String[20];
    CityName : String[20];
    PeopleName : String;
  End;

```

Таблицы

Табличная структура – чрезвычайно полезный инструмент для представления идей и выражения мыслей.

Таблица – это набор элементов одинаковой организации, каждый из которых можно представить в виде двойки $\langle K, V \rangle$, где K – ключ, а V – тело (информационная часть) элемента.

Таблицы бывают 4-х видов: *обычные; хэш-таблицы; электронные таблицы, таблицы решений, таблицы реляционных баз данных, табличное форматирование в WEB*.

Так, например, таблицы, как структуры данных – являются основными объектами реляционной базы данных.

Рассмотрим более подробно перечисленные табличные структуры данных.

Обычные таблицы

Таблицы в информатике рассматриваются как специально организованные структуры (контейнеры) для упорядоченного хранения данных.

Обычной таблицей называется перечень цифровых или информационных данных, которые располагаются в определенном порядке по графикам, столбцам и т. п.

Таблицы – это основные объекты реляционной базы данных. Еще одно применение таблиц это их применение при создан web-документов, так называемые разметочные таблицы. Разметочные таблицы лежат также в основе большинства web-страниц. Их главное достоинство – гибкость. Это особенно важно при создании динамических сайтов, например новостных.

Как физическая *структура данных таблица* представляет собой линейную последовательность ячеек памяти, число которых определяется количеством и размером полей каждого элемента, а также числом элементов таблицы.

Статистическая таблица – таблица, которая дает количественную характеристику статистической совокупности и представляет собой форму наглядного изложения полученных в результате статистической сводки и группировки числовых (цифровых) данных.

Хэш-таблицы

Хэш-таблица – это структура данных, реализующая интерфейс *map*, который позволяет хранить пары ключ / значение (рис. 11).

Она использует хеш-функцию для вычисления индекса в массиве, по которым можно найти желаемое значение.

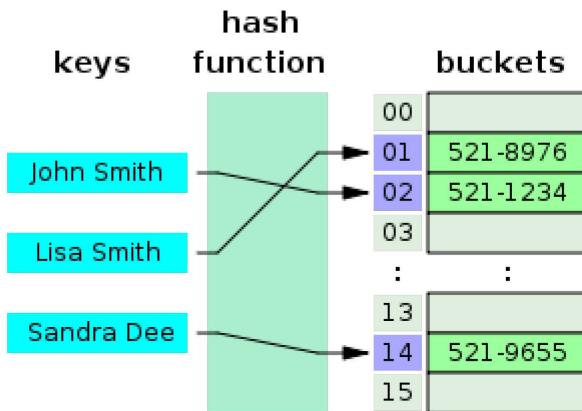


Рис. 11. Структура хэш-таблицы

Хеш-таблица – это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Хеш-таблица – это обычный массив с необычной адресацией, задаваемой хеш-функцией.

Важное свойство хеш-таблиц состоит в том, что при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время $O(1)$. Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: увеличить значение размера массива H и заново добавить в пустую хеш-таблицу все пары.

Учтите, что хеш-функция должна иметь следующие свойства:

- всегда возвращать один и тот же адрес для одного и того же ключа;
- не обязательно возвращает разные адреса для разных ключей;

- использует все адресное пространство с одинаковой вероятностью;
- быстро вычислять адрес.

Хеш-функция обычно принимает строку и возвращает числовое значение. Хеш-функция всегда должна возвращать одинаковое число для одного и того же ввода. Когда два ввода хешируются с одним и тем же цифровым выходом, это *коллизия* (рис. 12).

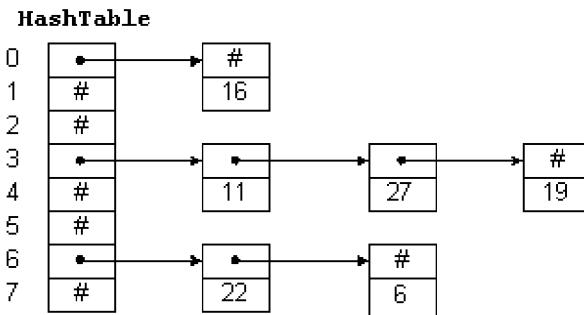


Рис. 12. Хеш-таблица с коллизиями

Разрешение коллизий

В информатике и криптографии *коллизия хеш-функции* – это равенство значений хеш-функции на двух различных блоках данных.

Разрешение коллизий (англ. collision resolution) в хеш-таблице, задача, решаемая несколькими способами: *метод цепочек*, *открытая адресация* и т.д. Очень важно сводить количество коллизий к минимуму, так как это увеличивает время работы с хеш-таблицами. Суть в том, чтобы их было как можно меньше. Поэтому, когда вы вводите пару ключ / значение в хеш-таблице, ключ проходит через хеш-функцию и превращается в число. Это числовое значение затем используется в качестве фактического ключа, в котором хранится значение. Когда вы снова попытаетесь получить доступ к тому же ключу, хеширующая функция обработает ключ и вернет тот же числовой результат. Затем число будет использовано для поиска связанного значения. Это обеспечивает очень эффективное время поиска со сложностью $O(1)$ в среднем.

Простейший способ обработки коллизий – сцепление элементов с одинаковыми значениями хеш-функции: все такие элементы сцепляются в список, а в хеш-таблицу помещается указатель на первый элемент этого списка. В пределах каждого такого списка осуществляется последовательный поиск.

Электронные таблицы

Электронные таблицы позволяют обрабатывать большие массивы числовых данных (рис. 13). В отличие от таблиц на бумаге электронные таблицы обеспечивают проведение динамических вычислений, т. е. пересчет по формулам при введении новых чисел. В математике с помощью электронных таблиц можно представить функцию в числовой форме и построить ее график, в физике – обработать результаты лабораторной работы, в географии или истории – представить статистические данные в форме диаграммы.

Вычисление цены устройства компьютера в рублях по заданному курсу доллара и евро.

	A	B	C	D	E	F
1	Устройство	Цена в у.е.	Цена в рублях	Цена в рублях	Курсы у.е.	
2	Системная плата	80	=\\$B2*E\$2	=\\$B2*F\$2	28	36
3	Процессор	70	=\\$B3*E\$2	=\\$B3*F\$2		
4	Оперативная память	15	=\\$B4*E\$2	=\\$B4*F\$2		
5	Жесткий диск	100	=\\$B5*E\$2	=\\$B5*F\$2		
6	Монитор	200	=\\$B6*E\$2	=\\$B6*F\$2		
7	Дисковод 3,5"	12	=\\$B7*E\$2	=\\$B7*F\$2		
8	Дисковод CD-ROM	30	=\\$B8*E\$2	=\\$B8*F\$2		
9	Корпус	25	=\\$B9*E\$2	=\\$B9*F\$2		
10	Клавиатура	10	=\\$B10*E\$2	=\\$B10*F\$2		
11	Мышь	5	=\\$B11*E\$2	=\\$B11*F\$2		
12	ИТОГО:	=СУММ(B2:B11)	=СУММ(C2:C11)	=СУММ(D2:D11)		

Рис. 13. Электронная таблица

Электронные таблицы – это работающее в диалоговом режиме приложение, хранящее и обрабатывающее данные в прямоугольных таблицах.

В отличие от обычных таблиц электронные таблицы – это специальные программы, работающие с динамическими структурами данных.

Следует помнить, что данные, представленные в таблице могут быть измерены в разных шкалах и поэтому над ними можно совершать только (допустимые) преобразования!!!

Таблицы решений

Таблица принятия решений – (таблица решений) способ компактного представления модели данных со сложной логикой. Аналогично условным операторам в языках программирования, они устанавливают связь между условиями и действиями. В отличие от традиционных языков программирования, таблицы решений в простой форме могут представлять связь между множеством независимых условий и действий.

Таблицы в WEB-программировании

Разметка web-страницы с помощью структуры «таблица» в языке HTML.:

```
<table border="1" cellpadding="10">
<tr>
  <td>Первая ячейка</td><td>Вторая ячейка</td><
  td>Третья ячейка</td>
</tr>
<tr>
  <td>Четвертая ячейка</td><td>Пятая ячейка</td><
  td>Шестая ячейка</td>
</tr>
</table>
```

Браузер покажет (рис. 14):

Первая ячейка	Вторая ячейка	Третья ячейка
Четвертая ячейка	Пятая ячейка	Шестая ячейка

Рис. 14. Разметка web-страницы тэгом «таблица»

Map

Map – это структура данных, которая хранит данные в парах ключ/значение, где каждый ключ уникален (рис. 15). *Map* иногда называется ассоциативным массивом или словарем.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Рис. 15. Структура МАР

Она часто используется для быстрого поиска данных. *Map*'ы позволяют сделать следующее:

- Добавление пары в коллекцию
- Удаление пары из коллекции
- Изменение существующей пары
- Поиск значения, связанного с определенным ключом

Полустатические структуры

Полустатические структуры данных характеризуются следующими признаками:

- имеют переменную длину и простые процедуры ее изменения;
- изменение длины структуры происходит в определенных пределах, не превышая какого-то максимального (пределного) значения.

Если полустатическую структуру рассматривать на *логическом уровне*, то это последовательность данных, связанная отношениями линейного списка. Доступ к элементу может осуществляться по его порядковому номеру.

Физическое представление полустатических структур данных в памяти – это обычно последовательность слотов в памяти, где каждый следующий элемент расположен в памяти в следующем слоте (т.е. вектор). Физическое представление может иметь, также, вид одностороннего связного списка (цепочки), где каждый следующий элемент адресуется указателем, находящимся в текущем элементе. В последнем случае ограничения на длину структуры гораздо менее строгие.

К полустатическим структурам относятся:

- стеки;
- очереди FIFO
- деки
- строки.

Рассмотрим их более подробно.

Стеки

Стек – линейный список, доступ к элементам которого осуществляется по принципу LIFO (Last In First Out).

Стек – это базовая структура данных, в которой вы можете только вставлять или удалять элементы в начале стека (рис. 16). Он напоминает стопку книг. Если вы хотите взглянуть на книгу в середине стека, вы сначала должны взять книги, лежащие сверху.

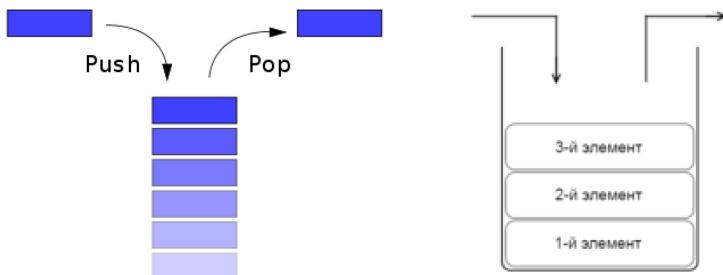


Рис. 16. Структура данных СТЕК

Стек считается LIFO (Last In First Out) – это означает, что последний элемент, который добавлен в стек, – это первый элемент, который из него выходит.

Основные операции над стеком:

- инициализация (Init);
- деструктизация (Destroy)
- помещение элемента в стек (Push);
- удаление элемента из стека (Pop);
- значение верхнего элемента (Top);
- проверка на пустоту (isEmpty);
- проверка на полноту (isFull).

Очереди

Вы можете думать об этой структуре, как об очереди людей в продуктовом магазине. Стоящий первым будет обслужен первым. Также как очередь.

Очередь – линейный список (рис. 17), доступ к элементам которого осуществляется по принципу FIFO (First In First Out).

Если рассматривать очередь с точки доступа к данным, то она является FIFO (First In First Out). Это означает, что после добавления нового элемента все элементы, которые были добавлены до этого, должны быть удалены до того, как новый элемент будет удален.

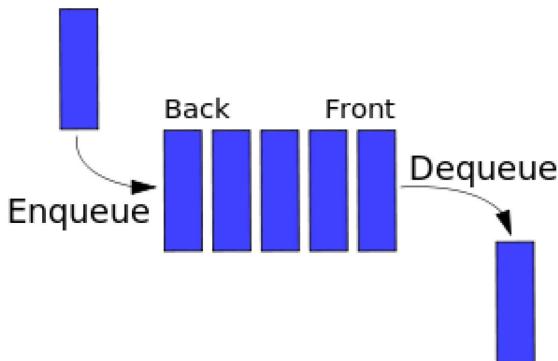


Рис. 17 а. Структура данных ОЧЕРЕДЬ

В очереди есть только две основные операции: *enqueue* и *dequeue*. Enqueue означает *вставить* элемент в конец очереди, а dequeue означает *удаление* переднего элемента.

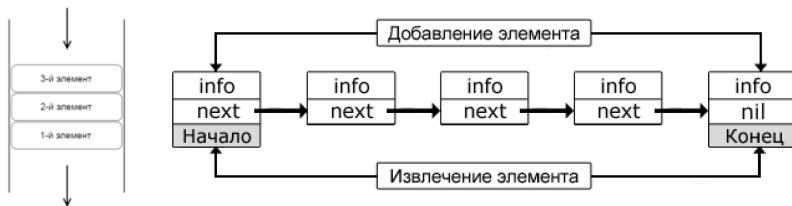


Рис. 17 б. Основные операции с ОЧЕРЕДЬЮ

Основные операции:

- инициализация (Init);
- деструктизация (Destroy);
- помещение элемента в очередь (Insert);
- удаление элемента из очереди (Remove);
- значение первого элемента (Head);
- значение последнего элемента (Tail);
- проверка на пустоту (isEmpty);
- проверка на полноту (isFull).

Деки

Дек – линейный список (рис. 18), доступ к элементам которого как с начала, так и с конца списка.

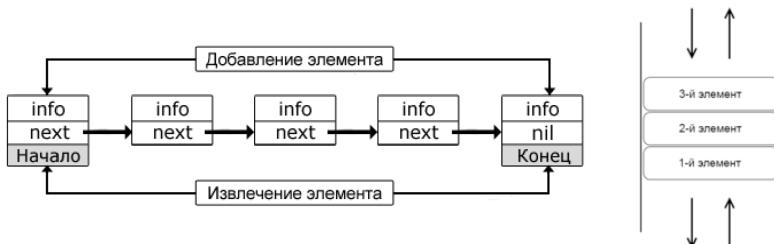


Рис. 18. Структура данных ДЕК

Основные операции:

- инициализация (Init);
- деструктизация (Destroy);
- помещение элемента в начало дека (InsertHead);
- помещение элемента в конец дека (InsertTail);
- удаление элемента из начала дека (RemoveHead);
- удаление элемента из конца дека (RemoveTail);
- значение первого элемента (Head);
- значение последнего элемента (Tail);
- проверка на пустоту (isEmpty);
- проверка на полноту (isFull).

Строки

Строка (*string*) – это структура данных (группа символов или их кодов), обрабатываемая как единый элемент.

В информатике под строкой символов понимают последовательность символов, заключенных в апострофы.

Пример: 'Адрес: улица Матвеева, 130'.

Под строкой в Прологе понимается последовательность символов, заключенная в двойные кавычки.

Пример:

S = «\84\117\114\98\111\32\80\82\79\76\79\71\»

S = «Turbo PROLOG».

Программы используют строки для хранения и передачи данных и команд.

Строка – это структура данных, состоящая из вектора символов и длины этого вектора. Вектор создается *конструктором* и уничтожается *деструктором*. Однако это может привести к не приятностям.

Например:

```
void f()
{
    string s1(10);
    string s2(20);
    s1 = s2;
}
```

будет размещать два вектора символов, а присваивание $s1 = s2$ будет портить указатель на один из них и дублировать другой.

Динамические структуры

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода *программа-компилятор* не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса.

Для решения проблемы адресации *динамических структур данных* используется метод, называемый *динамическим распределением памяти*, то есть память под отдельные элементы выделяется в момент, когда они «начинают существовать» в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.

Динамическая структура данных характеризуется тем что:

- она не имеет имени;
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

К динамическим структурам относят:

- однонаправленные (односвязные) списки;
- двунаправленные (двусвязные) списки;
- циклические списки;

- стек;
- дек;
- очередь;
- бинарные деревья.

Линейные связные списки

Список – набор элементов, каждый из которых состоит из двух полей. Одно поле содержит элемент данных или указатель на элемент данных, другое – указатель на следующий элемент списка, который, в свою очередь, тоже может быть начальным или промежуточным элементом другого списка.

Наличие явного указания на упорядоченность элементов списка позволяет достаточно легко манипулировать содержимым списка, включая новые и исключая старые элементы списка без их фактического перемещения в памяти. Это свойство позволяет размещать в памяти компьютера динамически изменяющиеся структуры данных.

Связные списки

Связный список (рис. 19) является одной из самых основных структур данных. Его часто сравнивают с массивом, поскольку многие другие структуры данных могут быть реализованы либо с помощью массива, либо с помощью связного списка. У каждого из них есть свои преимущества и недостатки.

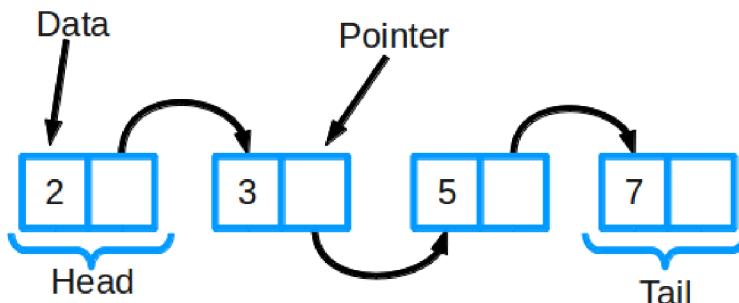


Рис. 19. Связный список

Связный список состоит из группы узлов, которые вместе представляют последовательность. Каждый узел содержит две вещи: фактические данные, которые хранятся (которые могут быть представлены любым типом данных), и указатель (или ссылка) на следующий узел в последовательности.

Существуют также дважды связанные списки, в которых каждый узел имеет указатель и на следующий, и на предыдущий элемент в списке.

Разветвленные связные списки.

Самые основные операции в связанном списке включают *добавление* элемента в список, *удаление* элемента из списка и *поиск* в списке для элемента.

Разветвленные связные списки

Нелинейным разветвленным списком является список, элементами которого могут быть тоже списки. Пример разветвленного списка представлен на рис. 20.

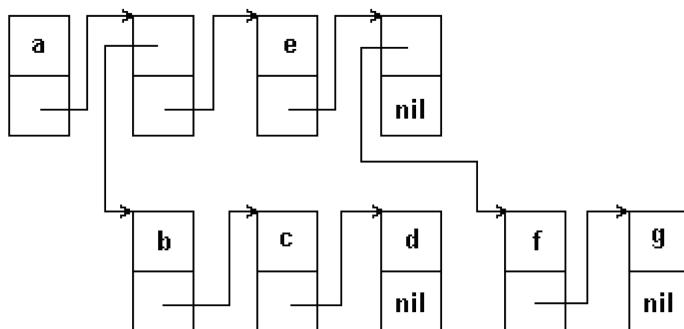


Рис. 20. Разветвленный список

Разветвленные списки описываются тремя характеристиками: *порядком*, *глубиной* и *длиной*.

Порядок. Над элементами списка задано транзитивное отношение, определяемое последовательностью, в которой элементы появляются внутри списка. В списке (x,y,z) атом x предшествует y , а y предшествует z . При этом подразумевается, что x предшествует z . Данный список не эквивалентен списку (y,z,x) . При

представлении списков графическими схемами порядок определяется горизонтальными стрелками. Горизонтальные стрелки истолковываются следующим образом: элемент из которого исходит стрелка, предшествует элементу, на который она указывает.

Глубина. Это максимальный уровень, приписываемый элементам внутри списка или внутри любого подсписка в списке. Уровень элемента предписывается вложенностью подсписков внутри списка, т.е. числом пар круглых скобок, окаймляющих элемент. В списке, изображенном на рис. А, элементы а и е находятся на уровне 1, в то время как оставшиеся элементы – б, с, д, ф и г имеют уровень 2. Глубина входного списка равна 2. При представлении списков схемами концепции глубины и уровня облегчаются для понимания, если каждому атомарному или списковому узлу приписать некоторое число l . Значение l для элемента x , обозначаемое как $l(x)$, является числом вертикальных стрелок, которое необходимо пройти для того, чтобы достичь данный элемент из первого элемента списка. На рис. А $l(a)=0$, $l(b)=1$ и т.д. Глубина списка является максимальным значением уровня среди уровней всех атомов списка.

Длина – это число элементов уровня 1 в списке. Например, длина списка на рис. А равна 3.

Графы

Граф – совокупность точек, соединенных линиями. Точки называются вершинами (узлами), а линии – ребрами (дугами) (рис. 21).

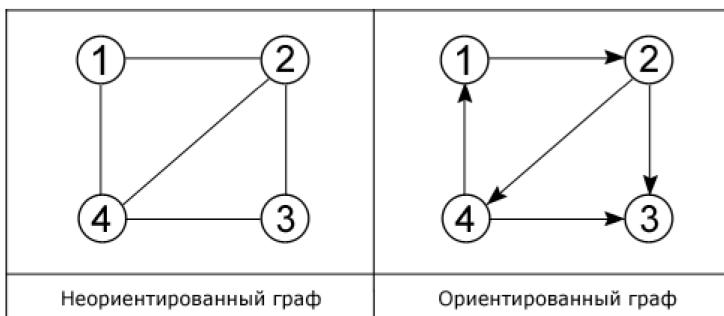


Рис. 21

Как показано на рисунке различают два основных вида графов: ориентированные и неориентированные. В первых ребра являются направленными, т. е. существует только одно доступное направление между двумя связными вершинами, например из вершины 1 можно пройти в вершину 2, но не наоборот. В неориентированном связном графе из каждой вершины можно пройти в каждую и обратно. Частный случай двух этих видов – смешанный граф. Он характерен наличием как ориентированных, так и неориентированных ребер.

Степень входа вершины – количество входящих в нее ребер, степень выхода – количество исходящих ребер.

Ребра графа необязательно должны быть прямыми, а вершины обозначаться именно цифрами, так как показано на рисунке. К тому же встречаются такие графы, ребрам которых присвоено в соответствие конкретное значение, они именуются взвешенными графиками, а это значение – весом ребра. Когда у ребра оба конца совпадают, т. е. ребро выходит из вершины F и входит в нее, то такое ребро называется петлей (рис. 22).

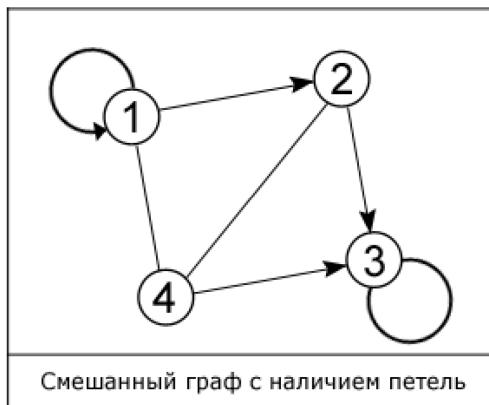


Рис. 22

Графы широко используются в структурах, созданных человеком, например в компьютерных и транспортных сетях, web-технологиях. Специальные способы представления позволяют использовать граф в информатике (в вычислительных машинах). Самые известные из них: «Матрица смежности», «Матрица инци-

дентности», «Список смежности», «Список рёбер». Два первых, как понятно из названия, для репрезентации графа используют матрицу, а два последних – список.

Графы представляют собой совокупности узлов (также называемых вершинами) и связей (называемых ребрами) между ними. Графы также известны как сети.

Одним из примеров графов является социальная сеть. Узлы – это люди, а ребра – дружба.

Существует два основных типа графов: ориентированные и неориентированные (рис. 23). Второй тип – это графы без какого-либо направления на ребрах между узлами. Ориентированные графы, напротив, представляют собой графы с направлением на них.

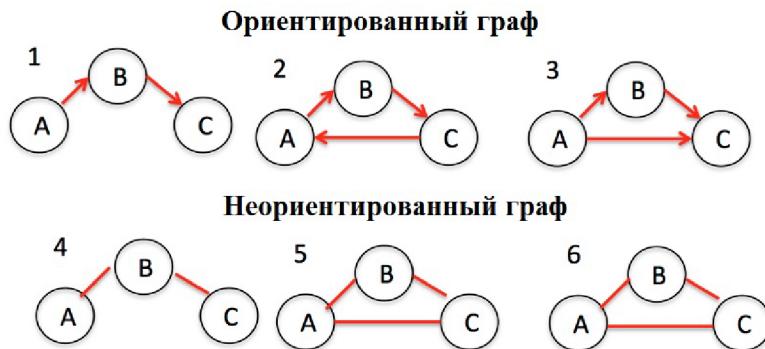
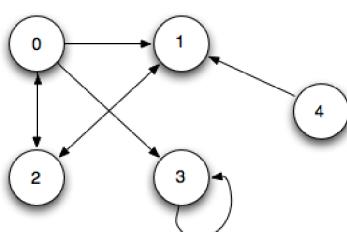


Рис. 23

Два частых способа представления графа – это список смежности и матрица смежности.



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

Список смежности может быть представлен как список, где левая сторона является узлом, а правая – списком всех других узлов, с которыми он соединен.

Матрица смежности представляет собой таблицу чисел, где каждая строка или столбец представляет собой другой узел на графе. На пересечении строки и столбца есть число, которое указывает на отношение. Нули означают, что нет ребер или отношений. Единицы означают, что есть отношения. Числа выше единицы могут использоваться для отображения разных весов.

Алгоритмы обхода – это алгоритмы для перемещения или посещения узлов в графе. Основными типами алгоритмов обхода являются поиск в ширину и поиск в глубину. Одно из применений заключается в определении того, насколько близко узлы расположены по отношению к корневому узлу.

Существует много алгоритмов на графах, в основе которых лежит систематический перебор вершин графа, такой что каждая вершина просматривается (посещается) в точности один раз. Поэтому важной задачей является нахождение хороших методов поиска в графе.

Под обходом графов (поиском на графах) мы будем понимать процесс систематического просмотра всех вершин графа с целью отыскания вершин, удовлетворяющих некоторому условию.

В. Липский называет метод поиска «хорошим»¹, если

- он позволяет алгоритму решения интересующей нас задачи легко «погрузиться» в этот метод
- каждое ребро графа анализируется не более одного раза (или, что существенно не меняет ситуации, число раз, ограниченное константой).

В данном разделе представлен алгоритм обхода графа, который называется поиском в ширину (Breadth First Search), и соответствующее этому алгоритму дерево. Об этом говорится на сайте <https://intellect.icu>. Также представлен алгоритм поиска в глубину(Depth First Search) и доказываются некоторые свойства этого вида обхода графа.

¹ Источник: <https://intellect.icu/algoritmy-obkhoda-grafa-4131>.

Эти алгоритмы можно эффективно использовать для поиска вершин, смежных с данной вершиной, построения простого пути между двумя вершинами, обнаружения циклов, проверки графа на связность и для многих других задач.

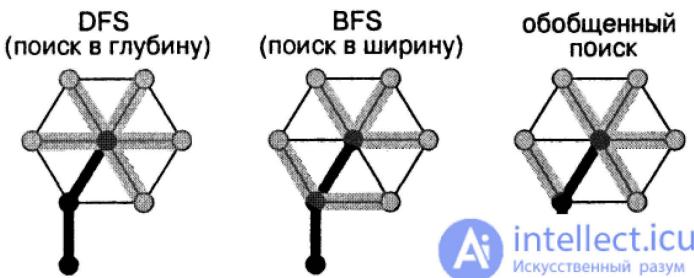


Рис. 24. Стратегии поиска вершины на графике
(серым отмечены использованные во время поиска вершины ребра)

Деревья

Дерево – это структура данных, состоящая из узлов. Она имеет следующие характеристики:

- 1) Каждое дерево имеет корневой узел (вверху).
- 2) Корневой узел имеет ноль или более дочерних узлов.
- 3) Каждый дочерний узел имеет ноль или более дочерних узлов и т. д.

Двоичное дерево поиска имеет + две характеристики:

- 1) Каждый узел имеет до двух детей (потомков).
- 2) Для каждого узла его левые потомки меньше текущего узла, что меньше, чем у правых потомков.

Двоичные деревья поиска позволяют быстро находить, добавлять и удалять элементы. Способ их настройки означает, что в среднем каждое сравнение позволяет операциям пропускать половину дерева, так что каждый поиск, вставка или удаление занимает времени, пропорциональное логарифму количества элементов, хранящихся в дереве.

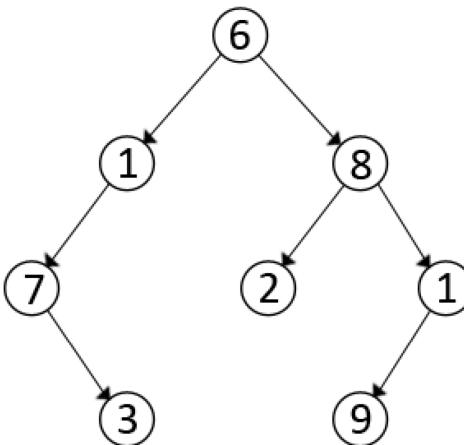


Рис. 25. Неупорядоченное дерево

Дерево как математический объект это абстракция из соименных единиц, встречающихся в природе. Схожесть структуры естественных деревьев с графиками определенного вида говорит о допущении установления аналогии между ними. А именно со связанными и вместе с этим ациклическими (не имеющими циклов) графиками. Последние по своему строению действительно напоминают деревья, но в чем то и имеются различия, например, принято изображать математические деревья с корнем расположенным вверху, т. е. все ветви «растут» сверху вниз. Известно же, что в природе это совсем не так.

Поскольку дерево это по своей сути граф, у него с последним многие определения совпадают, либо интуитивно схожи. Так, например, корневой узел (вершина 6) в структуре дерева – это единственная вершина (узел), характерная отсутствием предков, т. е. такая вершина, что на нее не ссылается ни какая другая вершина, а из самого корневого узла можно дойти до любой из имеющихся вершин дерева, что следует из свойства связности данной структуры.

Узлы, не ссылающиеся ни на какие другие узлы, иначе говоря, ни имеющие потомков называются листьями (2, 3, 9), либо терми-

нальными узлами. Элементы, расположенные между корневым узлом и листьями – промежуточные узлы (1, 1, 7, 8). Каждый узел дерева имеет только одного предка, или если он корневой, то не имеет ни одного.

Поддерево – часть дерева, включающая некоторый корневой узел и все его узлы-потомки. Так, например, на рисунке одно из поддеревьев включает корень 8 и элементы 2, 1, 9.

С деревом можно выполнять многие операции, например, находить элементы, удалять элементы и поддеревья, вставлять поддеревья, находить корневые узлы для некоторых вершин и др. Одной из важнейших операций является обход дерева. Выделяются несколько методов обхода. Наиболее популярные из них: симметричный, прямой и обратный обход. При прямом обходе узлы-предки посещаются прежде своих потомков, а в обратном обходе, соответственно, обратная ситуация. В симметричном обходе поочередно просматриваются поддеревья главного дерева.

Префиксное дерево

Бор, луч или дерево префикса – это своего рода дерево поиска. Оно хранит данные в шагах, каждый из которых является его узлом. Префиксное дерево из-за быстрого поиска и функции автоматического дописывания часто используют для хранения слов.

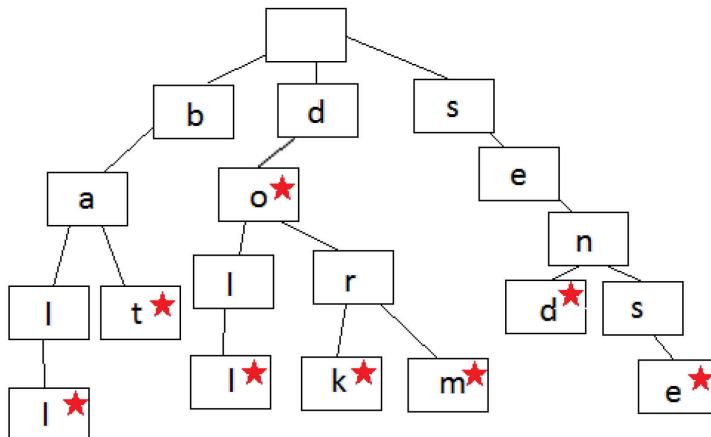


Рис. 26. Префиксное дерево

Каждый узел в префиксном дереве содержит одну букву слова. Вы следуйте ветвям дерева, чтобы записать слово, по одной букве за один раз. Шаги начинают расходиться, когда порядок букв отличается от других слов в дереве или, когда заканчивается слово. Каждый узел содержит букву (данные) и логическое значение, указывающее, является ли узел последним узлом в слове.

Посмотрите на рис. 26, и вы можете создавать слова. Всегда начинайте с корневого узла вверху и двигайтесь вниз. Показанное здесь дерево содержит слово ball, bat, doll, do, dork, dorm, send, sense.

Файловые структуры

Файловая структура – это совокупность файлов на диске и взаимосвязей между ними.

Файловая структура может быть одно– или многоуровневой. В одноуровневой структуре на носителе информации имена файлов образуют линейную последовательность, в многоуровневой, или иерархической, – древовидную структуру. Примером такой структуры может служить структура, приведенная на рис. 27.

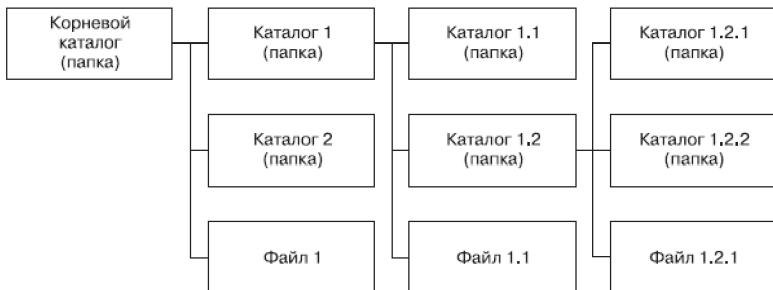


Рис. 27. Многоуровневая файловая структура

Корневой каталог (папка) содержит вложенные каталоги (папки) первого уровня, каждый из каталогов (папок) первого уровня может содержать вложенные каталоги (папки) второго уровня и т. д. В каталогах (папках) всех уровней могут храниться и файлы.

Иерархическую файловую структуру имеет ОС *Windows*. На верхнем уровне иерархии находится папка «Рабочий стол»; на втором уровне находятся системные папки «Мой компьютер»,

«Сетевое окружение», «Корзина» и т. д. На третьем уровне – диски, панель управления и т. д. Файловая система отображает внешние ЗУ в виде дисков. Каждому диску файловая система присваивает логическое имя: А: – гибкий диск (НГМД), С:, Д:, Е: и т. д. – жесткие диски (НЖМД), оптические диски (CD, DVD), флэш-память и т. д. В ОС *Windows* с помощью специальной системной программы можно искусственно провести условное разбиение жестких дисков в НЖМД на несколько логических дисков.

Последовательные файлы

Последовательный файл (sequential file) – это файл, доступ к которому производится последовательно от начала к концу, как если бы информация этого файла была организована в виде одной длинной строки. Примерами таких файлов могут служить звуковые файлы, видеофайлы, файлы, содержащие программы и текстовые документы. Фактически, большинство файлов, которые создает обычный пользователь ПК, являются последовательными.

Последовательный файл характеризуется последовательным размещением его физических записей, или блоков, в порядке их поступления в файл и доступом к блокам файла в порядке их физического расположения на соответствующем носителе данных. Типичным устройством внешней памяти, на котором организуются последовательные файлы, является НМЛ, для которого наиболее естественен последовательный способ обработки (записи, чтения и поиска) данных. Однако последовательные файлы можно создавать и обрабатывать и на запоминающих устройствах прямого доступа, а также на устройствах последовательного действия, которые существенно отличаются от НМЛ устройствах чтения перфокарт и перфолент, устройствах перфорации карт и лент, печатающих устройствах и т.п.

Логическая структура последовательного файла¹ представлена на рис. 28. Для каждой записи файла, кроме последней, имеется единственная следующая запись, которая на типичном последовательном носителе данных (таком, как магнитная лента) расположена

¹ <https://studfile.net/preview/4407052/page:16/>

жена физически вслед за предшествующей ей записью. В рассматриваемом типе файла, например, в общем случае обращение к записи i становится возможным лишь после обращения к записи $i-1$. Значит, если целью обращения к файлу является запись i , то при $i > 1$ предварительно необходимо пропустить $i-1$ предыдущих записей файла.



Рис. 28. Логическая структура последовательного файла

Операции над последовательными файлами имеют особенности в зависимости от типа внешних устройств, на которых создаются последовательные файлы. Основные операции доступа к последовательному файлу приведены в таблице 3.

Таблица 3

Операции при доступе к файлу	Тип устройства				
	НМЛ	ЗУПД	Чтения ПК	Перфорации	Печати
Чтение блока	Да	Да	Да	Нет	Нет
Чтение блока в обратном направлении	Да	Нет	Нет	Нет	Нет
Запись блока	Да	Да	Нет	Да	Да
Пропуск блока	Да	Да	*	*	*
Возврат на один блок	Да	Да	Нет	Нет	Нет
Пропуск файла	Да	Да	*	*	*
Возврат на один файл	Да	Да	Нет	Нет	Нет
Изменение блока на месте	Нет	Да	Нет	Нет	Нет
Определение текущей позиции файла	Да	Да	Нет	Нет	Нет
Установка в заданную позицию	Да	Да	Нет	Нет	Нет

Файлы последовательного доступа требуют значительных временных затрат при поиске необходимых данных. Поэтому в задачах, требующих минимизации времени поиска необходимых данных применяются файлы прямого доступа, которые в настоящее время применяются в подавляющем большинстве случаев.

Файлы прямого доступа

Файлы с постоянной длиной записи, расположенные на устройствах прямого доступа, называются *файлами прямого доступа*. В этих файлах физический адрес расположения нужной записи может быть вычислен по номеру записи.

Файлы прямого доступа могут быть форматными, двоичными, неформатными. Все записи в файле прямого доступа имеют одинаковую длину. Длина записи задается параметром RECL= в операторе открытия файла. Для открытия форматного файла прямого доступа в программе записывается оператор следующей структуры:

OPEN(N, FILE=fname, ACCESS='DIRECT', FORM='FORMATTED',

Оператор открытия двоичного файла прямого доступа:

OPEN(N, FILE=fname, ACCESS='DIRECT', FORM='BINARY', RECL=recl)

Здесь recl – целое число, задающее длину записи.

По умолчанию файл прямого доступа имеет опцию FORM='UNFORMATTED'.

Файлы комбинированно доступа и файлы организованные разделами оставлены для самостоятельного изучения

Алгоритмические структуры

Алгоритмы можно представлять в виде специального вида структур. Логическая структура любого алгоритма может быть представлена комбинацией трех базовых структур (рис. 29):

- 1) *Следование*. Предполагает последовательное выполнение команд сверху вниз. Если алгоритм состоит только из структур следования, то он является линейным.
- 2) *Ветвление*. Выполнение программы идет по одной из двух, нескольких или множества ветвей. Выбор ветви зависит от условия на входе ветвления и поступивших сюда данных.

- 3) **Цикл.** Предполагает возможность многократного повторения определенных действий. Количество повторений зависит от условия цикла.
- 4) **Функция (подпрограмма).** Команды, отделенные от основной программы, выполняются лишь в случае их вызова из основной программы (из любого ее места). Одна и та же функция может вызываться из основной программы сколько угодно раз.

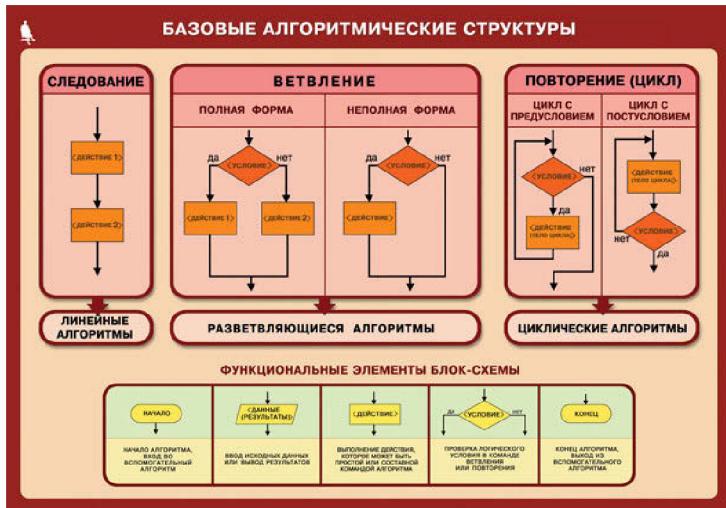


Рис. 29. Базовые алгоритмические структуры

Вопросы для самоконтроля

1. Какую структуру данных называют списком?
2. В чем отличие первого элемента одностороннего (дву направленного) списка от остальных элементов этого же списка?
3. В чем отличие последнего элемента одностороннего (дву направленного) списка от остальных элементов этого же списка?
4. Почему при работе с односторонним списком необходимо позиционирование на первый элемент списка?

5. Почему при работе с двунаправленным списком не обязательно позиционирование на первый элемент списка?
6. В чем принципиальные отличия выполнения добавления (удаления) элемента на первую и любую другую позиции в одностороннем списке?
7. В чем принципиальные отличия выполнения основных операций в односторонних и двунаправленных списках?
8. С какой целью в программах выполняется проверка на пустоту одностороннего (двунаправленного) списка?
9. С какой целью в программах выполняется удаление одностороннего (двунаправленного) списка по окончании работы с ним? Как изменится работа программы, если операцию удаления списка не выполнять?
10. Какую структуру данных называют стеком?
11. На базе каких структур может быть организован стек?
12. В чем преимущества и недостатки организации структур в виде стека?
13. В чем преимущества и недостатки организации структур в виде очереди?
14. Для моделирования каких реальных задач удобно использовать стек? А для каких очередь?
15. Какое значение хранит указатель на стек?
16. Какое значение хранит указатель на очередь?
17. Какие существуют ограничения на тип информационного поля стеки и очереди?
18. С какой целью в программах выполняется проверка на пустоту стека и очереди?
19. При работе со стеком или очередью доступны позиции ограниченного числа элементов. Возможна ли ситуация записи новых элементов стека или очереди на уже занятые собственными элементами участки памяти (запись себя поверх себя)? Ответ обоснуйте.
20. С какой целью в программах выполняется удаление стека и очереди по окончании работы с ними? Как изменится работа программы, если операцию удаления не выполнять?
21. Какую структуру данных называют очередью?

Алгоритмы

*«Если бы строили здания так же, как
программисты пишут программы, первый
затевавший дятел разрушил бы цивилизацию»*

Второй закон Вейлера

С самого рождения человек сталкивается с алгоритмами. Например, в виде алгоритма можно описать процесс похода в школу, в театр, университет; приобретение продуктов в магазине, на рынке, лекарств в аптеке; обучения езде на двухколесном велосипеде и т. п.

Алгоритм помогает человеку, слабо знакомому с каким-либо процессом, все же достичь желаемого результата.

Так что же это такое – алгоритм? Определим данное понятие.

Алгоритм¹ – точная конечная последовательность команд, приводящая от исходных данных к искомому результату, за конечное число шагов.

Более точно, под *алгоритмом* обычно понимают точное предписание, которое задает вычислительный процесс и представляет собой конечную последовательность обычных элементарных действий, четко определяющую процесс преобразования исходных данных в искомый результат, и удовлетворяющего следующим свойствам:

1. *алгоритм* – это процесс последовательного построения величин, идущий в дискретном времени таким образом, что в начальный момент задается исходная конечная система величин, а в каждый следующий момент система величин, получается, по определенному закону (программе) из системы величин, имевшихся в предыдущий момент времени (*дискретность алгоритма*).
2. Система величин, получаемых в какой-то (не начальный) момент времени, однозначно определяется системой вели-

¹ Слово «алгоритм» происходит от имени великого среднеазиатского ученого 8–9 вв. Аль-Хорезми Хорезм – историческая область на территории современного Узбекистана.

чин, полученных в предыдущие моменты времени (*детерминированность алгоритма*).

3. Если способ получения последующей величины из какой-нибудь заданной величины не дает результата, то должно быть указано, *что* надо считать результатом алгоритма (*направленность алгоритма*).
4. Начальная система величин может выбираться из некоторого потенциально бесконечного множества (*массовость алгоритма*).

Таким образом, алгоритм обладает следующими *свойствами*:

1. *Дискретностью* (в данном случае, разделенностью на части) и упорядоченностью. Алгоритм должен состоять из отдельных действий, которые выполняются последовательно друг за другом.
2. *Детерминированностью* (однозначная определенность). Многократное применение одного алгоритма к одному и тому же набору исходных данных всегда дает один и тот же результат.
3. *Формальностью* – алгоритм не должен допускать неоднозначности толкования действий для исполнителя.
4. *Результативностью и конечностью*. Работа алгоритма должна завершаться за определенное число шагов, при этом задача должна быть решена.
5. *Массовостью*. Определенный алгоритм должен быть применим ко всем однотипным задачам.

Алгоритмы разрабатывают обычно тогда, когда решают задачи, и чаще всего, когда задачи решают на компьютере. Рассмотрим место и роль алгоритма в решении задач на компьютере.

От понятной задачи к алгоритму

Во-первых, зафиксируем то, что мы будем понимать под термином «задача».

Задачу мы будем понимать, как неудовлетворенное чувство беспокойства. Отсюда – удовлетворение чувства беспокойства есть *решение задачи*.

Все задачи будем делить на три класса: осмысленные, полузадачи и томления.

Определим, какую задачу мы будем считать *понятной* (*осмысленной*), а какую не до конца осмысленной (*полузадачей* или *томлением*), а значит требующей еще каких-то дополнительных действий по её осмыслению. Это важно еще и потому, что при решении задачи на компьютере затрачиваются *существенные ресурсы* (материальные, финансовые, людские, временные и другие) и пренебрегать этим преступно.

Будем считать, что задача нам понятна (*осмысленна нами*) если существует *нечто* (будем называть это «*критерием осмысленности*»), позволяющее нам однозначно отделять процесс решения задачи от ее *не-решения* (заметим: не ответа от *не-ответа*).

Таким образом, будем считать задачу *осмысленной* (*понятной*) нам, если определено следующее:

Осмысленная задача = условие + критерий осмысленности + результат.

В свою очередь

Критерий осмысленности = алгоритм + ограничения + исключительные ситуации

В нашем случае (при решении задачи на компьютере), алгоритм такой задачи (если он существует) должен *иметь реализацию* на компьютере. В *ограничениях* должны быть перечислены все те ситуации, при которых задача *будет иметь решение*, а в *исключительных ситуациях* – указываются те условия, при которых задача *никогда не будет иметь решения*.

Пример такой осмысленной задачи, которая имеет алгоритм, реализуемый на компьютере, приведен в *приложении 1*.

Если мы можем указать все *решения*, а о *не-решениях* мы не знаем ничего, то в этом случае мы имеем дело с *полузадачами*. В этом случае, вначале, необходимо дополнить все *решения* еще и *не-решениями* задачи. (Это уже прерогатива ученых). А уже только после этого приступать к разработке алгоритма.

В случае *томления*, дело обстоит следующим образом. Поскольку мы не знаем ни одного *решения*, а только некоторые *не-*

решения, необходимо попытаться восполнить этот пробел – найти все решения задачи, что сделать практически невозможно (поскольку человек «сам не знает, чего он хочет»). Нас этот случай не интересует.

Заметим, что всякому алгоритму соответствует задача, для решения которой он был построен. Обратное утверждение в общем случае является *неверным* по двум причинам:

- *во-первых*, одна и та же задача может реализовываться различными алгоритмами;
- *во-вторых*, можно предположить, что имеются такие задачи, для которых алгоритм вообще не может быть построен.

Примером, подтверждающим сказанное выше, может служить так называемая «проблема останова». Рассмотрим её.

Пусть у нас есть строка с закодированной функцией A (на каком-то языке, это не принципиально). Требуется создать функцию F, которая определит, будет ли функция A выполняться вечно или нет (зависнет или не зависнет программы).

Задача *останова*¹ является неразрешимой задачей, т.к. для ее решения не существует алгоритма².

¹ Пусть у нас есть строка с закодированной функцией A (всё равно на каком языке, это не принципиально). Требуется создать функцию F, которая определит, будет ли функция A выполнять вечно или нет (зависнет или не зависнет). Естественно, работа A зависит от входных данных D. Соответственно F должна принимать на вход две переменных: код функции A и входные параметры D. А возвращать F(A, D) будет логическое значение: True – A(D) остановится, False – A(D) будет работать вечно.

² Зачем решать задачу останова? Всем, кто знаком с задачами криптографии, известно, какую важную роль в этой дисциплине играют простые числа Мерсенна. Человечество прилагает множество усилий по поиску этих чисел. Последнее (пока самое большое) было обнаружено 6 сентября 2008 года и насчитывало 11185272 десятичных знаков. И прямо сейчас десятки тысяч процессоров работают над поиском новых чисел. Числа Мерсенна напрямую связаны с совершенными числами. На сегодняшний день доказано, что первое нечётное совершенное число должно быть больше, чем 10^{300} . Перебор не обнаружил таких чисел, дойдя до 10^{500} . Неразрешимость проблемы остановки была впервые доказана в 1936 году Алланом Матисоном Тьюрингом (Alan Mathison Turing, 1912-1954). Это далеко не единственный вопрос, который легко решился бы сразу, как только мы получили бы решение проблемы остановки. Точно таким же способом можно было бы доказать или опровергнуть очень многие гипотезы современной

С другой стороны, если задача имеет алгоритм, да еще реализуемый на компьютере, то мы имеем реальный шанс достичь желаемого результата, т. е. – решения задачи.

Вначале определим, какими могут быть алгоритмы.

Классификация алгоритмов

Алгоритм для какой-либо задачи называется:

- *терминистическим (завершающимся, применимым)*, если он, для всех допустимых последовательностей шагов, *заканчивается* после *конечного* числа шагов;
- *детерминистическим*, если нет никакой *свободы* в выборе очередного шага обработки;
- *детерминированным*, если *результат* алгоритма определен однозначно;
- *последовательным*, если шаги обработки всегда выполняются *друг за другом*;
- *параллельным*, если некоторые шаги обработки могут выполняться *одновременно*.

Определив, какие бывают алгоритмы, мы можем приступить к их разработке – определим стадии алгоритмизации.

Стадии алгоритмизации

Решение задачи на компьютере начинается с разработки ее *модели*¹, на основе которой мы затем будем разрабатывать *алгоритм* и затем, *специфицировать задачу*. Разработка модели необходима нам для того, чтобы *конкретизировать*, только *основные моменты* решаемой задачи. В реальности каждая задача может обладать бесчисленным множеством характеристик (параметров),

математики. Например, гипотезу Биля, за которую даже назначена награда в 100000 долларов США. *Гипотеза Биля* формулируется следующим образом: неопределенное уравнение

$$Ax + By = Cz$$

не имеет решения в целых положительных, т.е. натуральных числах A, B, C, x, y и z при условии, что x, y и z больше 2.

¹ Модель – это абстракция реальности с выделенными наиболее важными свойствами.

не все из которых нам важны и существенны в данный момент времени.

После описания модели предметной области, в которой решается задача, переходят к разработке алгоритма, которая осуществляется в несколько этапов.

Вначале описывается *конечная последовательность* определенных действий, приводящих к результату. Затем эта последовательность действий *приводится в соответствие* с определением алгоритма, т.е. проверяются свойства алгоритма и, если необходимо, переопределяется указанная ранее последовательность действий. После этого *выбирается форма записи алгоритма, testeируется алгоритм или доказывается его правильность* и, наконец, алгоритм *реализуется* в виде программы (например, с помощью языка программирования и соответствующего инструментария).

Поскольку компьютер является автоматом, то в алгоритме не должно содержаться *непонятных предписаний* вида, изображенного на рис. 30.

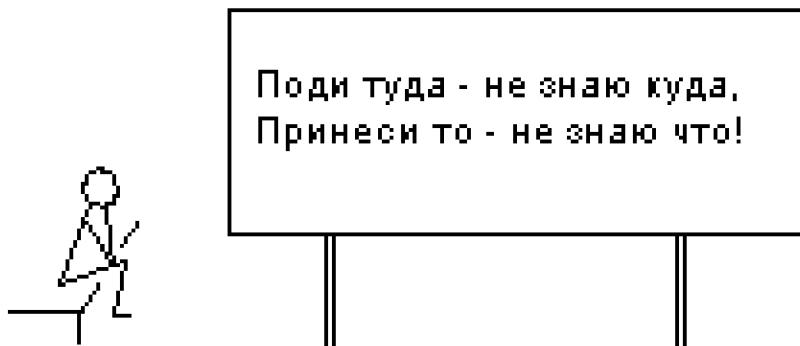


Рис. 30. Определенность алгоритма

Следует заметить, что *не каждое правило*, приводящее к решению задачи, является алгоритмом. Поясним это на примере.

Пример. Пусть требуется провести перпендикуляр к прямой MN в заданной точке A (рис. 31).

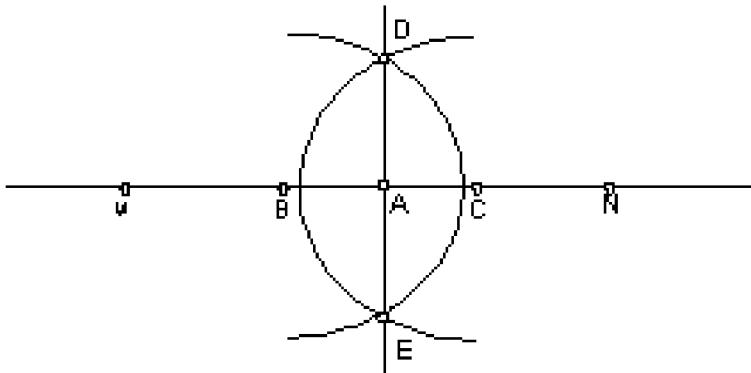


Рис. 31. Описание задачи

Решить данную задачу человек-исполнитель может следующим образом:

1. Отложить в обе стороны от точки A (на прямой MN) циркулем отрезки равной длины с концами B и C ,
2. Увеличить раствор циркуля до радиуса, в полтора-два раза большего длины отрезков AB в AC .
3. Провести указанным раствором циркуля последовательно с центрами B и C дуги окружностей так, чтобы они охватили точку A и образовали две точки пересечения друг с другом D и E .
4. Взять линейку и приложить ее к точкам D и E и соединить их отрезком. При правильном построении отрезок пройдет через точку A и будет искомым перпендикуляром.

Заметим, что приведенный список правил способен выполнить только человек, поскольку он привлечет свои интеллектуальные ресурсы для разрешения «узких мест». Для компьютера – это неразрешимая задача.

Вот перечень тех «узких мест», которые компьютер не способен разрешить.

В пункте 1 требуется от исполнителя сделать выбор отрезка произвольной длины (для построения точек B и C надо провести окружность произвольного радиуса r с центром в точке A).

В пункте 2 требуется сделать выбор отрезка в полтора-два раза большего длины отрезков AB и AC . В пункте 3 надо провести дуги, которые также однозначно не определяются их описанием.

Человек-исполнитель, применяющий повторно данное правило, к одним и тем же исходным данным (прямой MN и точке A), получит несовпадающие промежуточные результаты¹ и вынужден применить свой интеллектуальный аппарат, чего не может сделать компьютер.

Преобразовать данное правило, чтобы оно удовлетворяло свойствам алгоритма можно следующим образом.

Для этого вместо выбора произвольного радиуса будем указывать в каждом случае конкретный радиус. Однако этим неопределенность команд полностью не снимается. В *инструкциях 1 и 2*, кроме проведения окружностей, требуется находить точки пересечения и как-то их обозначать.

В *инструкции 2* требуется присвоить имена точкам пересечения прямой MN и окружности. Здесь можно договориться обозначать точки, например, так, чтобы векторы и были сонаправлены.

В *инструкции 3* требуется обозначить точки пересечения окружностей. Договоримся, например, обозначать через D ту из двух точек пересечения, которая находится левее, если смотреть из центра B в направлении центра C .

Кроме того, вместо дуг окружностей в *инструкции 3*, мы будем проводить окружности, ибо тем самым снимается неопределенность инструкции «проводсти дугу, охватывающую точку».

Таким образом, алгоритм проведения перпендикуляра к прямой MN в заданной точке ($A \in MN$) будет выглядеть следующим образом.

1. Провести окружность ω радиуса 1 с центром в точке A .
2. Обозначить точки пересечения окружности ω с прямой MN через B и C так, чтобы $\overline{BC} \uparrow\uparrow \overline{MN}$.
3. Последовательно провести окружности ω_B и ω_C радиуса 2 с центром соответственно в точках B и C .

¹ Не выполнено свойство детерминированности алгоритма.

4. Обозначить точки пересечения окружностей ω_B и ω_C через D и E так, чтобы обход многоугольника $BDCE$ (последовательно от B через D и C к E) совершился по часовой стрелке.
5. Провести прямую DE . Прямая DE – искомый перпендикуляр.

Теперь можно описать технологию разработки алгоритма при решении некоторой задачи.

Разработка алгоритма

Для того чтобы корректно разработать алгоритм решения задачи, необходимо выполнить следующие действия:

1. понять задачу, для которой разрабатывается алгоритм;
2. выбрать *метод* проектирования алгоритма;
3. выбрать *форму* записи алгоритма (блок-схемы, псевдокод и др.);
4. подготовить *тесты* и методы тестирования;
5. *спроектировать* алгоритм.

Разработка алгоритма сложный процесс, поскольку он является с одной стороны *творческим* и ориентирован на *интеллект* разработчика (уровень его *знаний* и *умений*, творческое мышление, хорошие базовые знания и многое другое), а с другой стороны – обладает всеми чертами *ремесленничества* (технологичности), поскольку требует от человека *умений* и *навыков* проектирования.

Понимание задачи

Как уже отмечалось ранее, человек понимает задачу только тогда, когда он способен однозначно распознать все предъявляемые ему результаты, как решение, или как не-решение задачи.

В какой же момент наступает *понимание задачи*?

Ответ следующий. Человек понимает стоящую перед ним задачу, а не просто *томится* ею, когда у него есть *нечто* (критерий осмыслинности), позволяющее отделять процесс решения от процесса не-решения. В информатике таким нечто является алгоритм, реализующий на компьютере, заданные *ограничения* и сформули-

рованные исключительные ситуации. Именно эта тройка гарантирует осмысленность задачи.

Кроме этого, необходимо зафиксировать исходные данные и представление результатов вычислений (содержание и форма).

Таким образом, для понимания задачи человеком необходимо:

- структурировать исходные данные;
- сформулировать «критерий осмысленности» (алгоритм, ограничения и исключительные ситуации);
- определиться с формой и содержимым результатов решения задачи.

Пожалуй, самым сложным в этом процессе является разработка алгоритма. С этого мы и начнем.

Выбор метода проектирования

Разработка алгоритма с одной стороны процесс творческий, а с другой – ремесленнический. Творчество связано с наличием соответствующего интеллектуального потенциала (знаний, умений и навыков), а ремесло – с разумным выбором метода проектирования и реализации алгоритма в виде программного кода, для исполнения его на компьютере.

Творческая составляющая разработки алгоритма решения задачи достаточно хорошо описана [Пойа].

Методы проектирования алгоритмов

Среди методов проектирования алгоритмов рассмотрим следующие:

- метод частных целей, подъема и обрабатывания назад;
- эвристики;
- программирование с отходом назад;
- метод ветвей и границ;
- рекурсия;
- моделирование;
- аксиоматический метод.

Метод частных целей, подъема и обрабатывания назад

Указанные в заголовке методы в своей основе базируются на декомпозиции исходной задачи¹.

Метод частных целей сводится к тому, что первоначальную задачу пытаются *разбить* на более простые (мелкие), которые не меняют исходные цели.

Метод подъема (восходящий) начинается с принятия начального предложения или вычисления начального решения задачи. Затем начинается *быстрое* (насколько это возможно) *движение вверх* от начального решения по направлению к лучшим решениям.

Метод обрабатывания назад начинается с цели или решения и движется по направлению к начальной постановки задачи. Затем, если эти действия обратимы, движемся *обратно* от постановки задачи к решению.

Эвристики

Эвристический алгоритм, или *эвристика*, определяется как алгоритм со следующими свойствами [Гудман и Хидентиеми]:

- он обычно находит хорошие, хотя *не обязательно оптимальные* (по занимаемому объему оперативной памяти и быстродействию) решения;
- его можно *быстрее и проще реализовать*, чем любой известный точный алгоритм (т.е. тот, который гарантирует оптимальное решение).

Программирование с отходом назад

Программирование с отходом назад – это организованный исчерпывающий поиск, который часто позволяет избежать исследования всех возможностей.

¹ См. пример. Задача о джипе, пресекающем пустыню («Гудман С., Хидентиеми С. Введение в разработку и анализ алгоритмов. – М.: Мир, 1981»).

Этот метод особенно удобен для решения задач, требующих проверки потенциально большого, но конечного числа решений.

Примером применения этого метода проектирования алгоритмов может служить задача о комбинационном замке для велосипеда [Гудман].

Промоделируем каждую возможную комбинацию вектором из N нулей и единиц с помощью двоичного дерева (рис. 32).

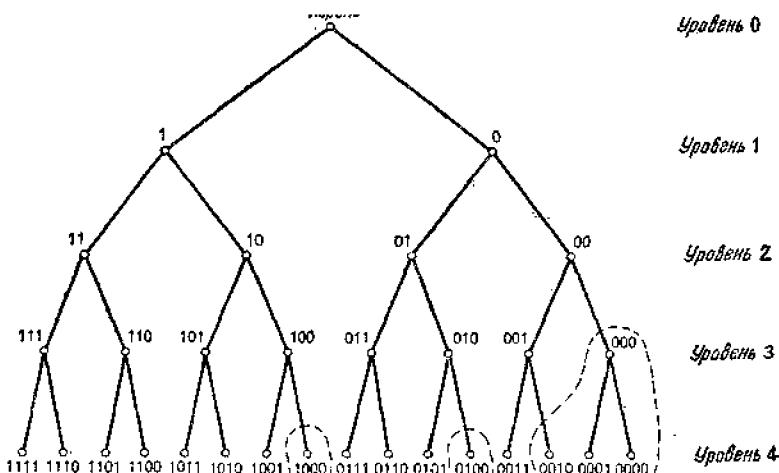


Рис. 32. Дерево решения задачи о замке

Метод ветвей и границ

Метод ветвей и границ похож на методы с отходами назад тем, что он исследует древовидную модель пространства решений и применим для широкого круга дискретных комбинаторных задач.

Пример. Задача коммивояжера – классический алгоритм Литл, Мерти, Суини и Карел [СиК].

Рекурсия

Рекурсия – это метод определения или выражения функции, процедуры, языковой конструкции или решения задачи посредством *той же функции*, процедуры и т.д.

В качестве примера рассмотрим вычисление факториала и чисел Фибоначчи

Факториальная функция определяется рекурсивно следующим образом:

$$0! = 1,$$

$$N! = N * (N - 1)!, \text{ если } N > 0,$$

или в виде фрагмента программы:

$$FAC(0) = 1, \quad (1)$$

$$FAC(N) = N * FAC(N - 1), \text{ если } N > 0. \quad (2)$$

Пример. Задача о кроликах.

Каждый месяц самка из пары кроликов приносит двух кроликов (самца и самку). Через два месяца новая самка сама приносит пару кроликов. Нужно найти число кроликов в конце года, если в начале года была одна новорожденная пара кроликов и в течение этого года кролики не умирали.

Формы записи алгоритма

Наибольшее распространение формы записи алгоритмов получили:

- аналитическая;
- графическая;
- табличная;
- псевдокод;
- в виде диаграмм.

Аналитическая форма записи алгоритма

Аналитическая форма записи алгоритмов может быть представлена: *словесным описанием, псевдокодом и на языке математики*.

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке.

На языке математики форма записи алгоритма может быть представлена набором формул.

Например, для записи алгоритма решения полного квадратного уравнения

$$ax^2 + bx + c = 0$$

достаточно указать следующую формулу

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

которая выполняется на множестве вещественных чисел.

Например, словесная запись алгоритма Евклида (вычисления наибольшего общего делителя) выглядит следующим образом.

Описание алгоритма нахождения НОД делением

1. Большее число делим на меньшее число.
2. Если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла).
3. Если есть остаток, то большее число заменяем на остаток от деления.
4. Переходим к пункту 1.

Сам алгоритм выглядит следующим образом.

1. Поместить в участок памяти с именем x число m ; перейти к выполнению пункта 2.
2. Поместить в участок памяти с именем y число n ; перейти к выполнению пункта 3.
3. Если выполняется условие « x не равно y », то перейти к выполнению пункта 5, иначе перейти к выполнению пункта 4.
4. Поместить в участок памяти с именем НОД значение из блока памяти x ; перейти к выполнению пункта 8.

5. Если выполняется условие $x > y$, то перейти к выполнению пункта 6, иначе перейти к выполнению пункта 7;
6. Поместить в участок памяти с именем x значение выражения
7. $x - y$; перейти к выполнению пункта 3.
8. Поместить в участок памяти с именем y значение выражения
9. $y - x$; перейти к выполнению пункта 3.
10. Закончить работу.

Графическая форма записи алгоритмов



Рис. 33. Блок-схема алгоритма

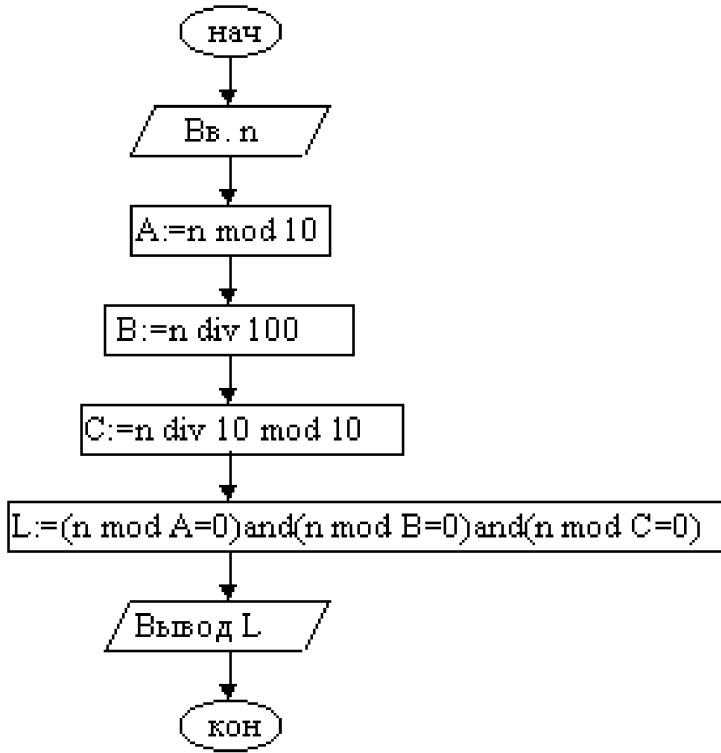
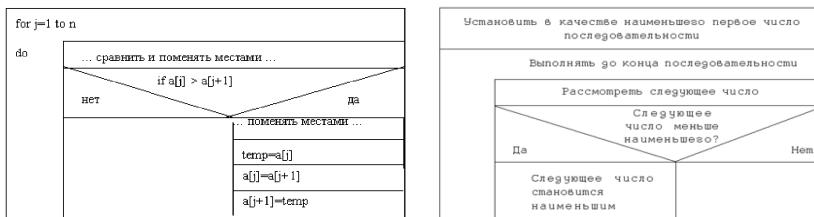


Рис. 34. Графическая форма записи алгоритма

К разновидностям графической формы записи алгоритмов относятся схемы Несси-Шнейдермана, Р-язык Вельбицкого, контекстные диаграммы (рис. 34 а-в) и некоторые другие.



Графическая схема алгоритма

Схемы Насси-Шнейдермана

Такая схема визуально представляет собой подобный псевдокод текст, помешанный в прямоугольные блоки, структурирующие алгоритм. Это более наглядный способ представления алгоритмов, чем псевдокод, но менее громоздкий, чем блок-схемы.

Алгоритм "Переход человека через автомобильную дорогу на нерегулируемом переходе"

Стоять на тротуаре в начале перехода

Смотреть налево

Слева близко едет машина?

ДА	НЕТ
----	-----

Стоять, ждать проезда машины

Идти до середины дороги

Дойдя до середины дороги остановиться

Смотреть направо

Справа близко едет машина?

ДА	НЕТ
----	-----

Стоять, ждать проезда машины

Перейти дорогу к конца

28

Рис. 34 а. Схемы Насси-Шнейдермана

P-язык записи алгоритмов

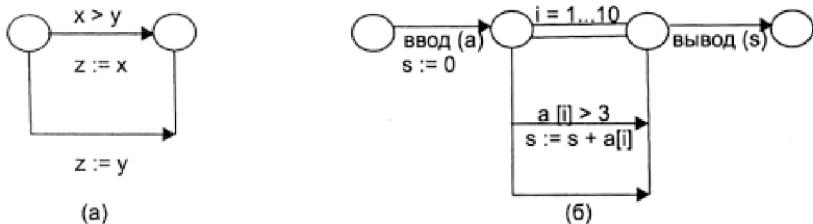


Рис. 34 б. Р-язык Вельбицкого

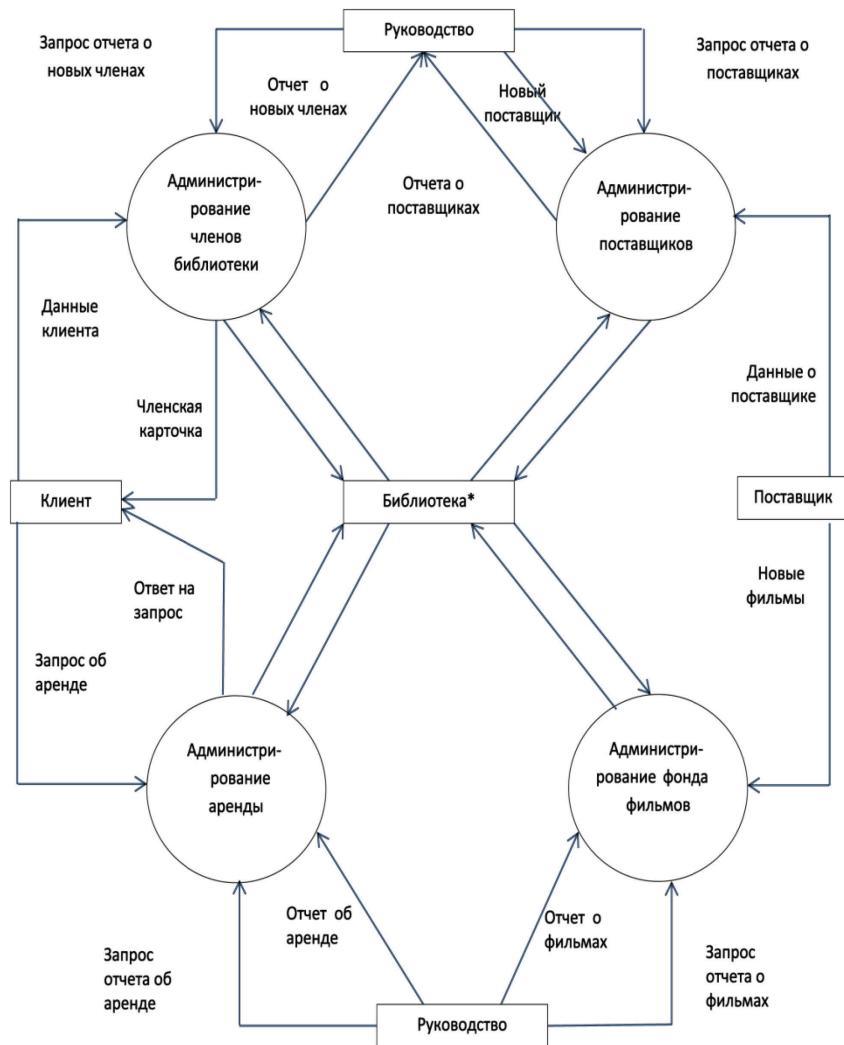


Рис. 34 в. Контекстная диаграмма

Табличная форма записи алгоритма

Для графической формы записи алгоритма применяются таблицы решений (рис. 35)

Наименование таблицы	Правило 1	Правило 2	...	Правило <i>m</i>
Условие 1	b_{11}	b_{12}	...	b_{1m}
Условие 2	b_{21}	b_{22}	...	b_{2m}
.		...		
.		...		
Условие <i>n</i>	b_{n1}	b_{n2}	...	b_{nm}
Действие 1	X			
Действие 2		X		
.				
.				
Действие <i>k</i>			X	X

Рис. 35. Схема таблицы решений

Ниже представлен пример таблицы решений для задачи выбора прибора измерения тока, напряжения, сопротивления и мощности.

Пример. Для выбора электрических приборов должны быть установлены следующие величины: W (мощность), I (ток), U (напряжение) и R (сопротивление).

При этом чаще всего задаются две величины, а две остальные вычисляются по следующим формулам:

$$F1 : R = U/I; \quad W = UI;$$

$$F2 : I = U/R; \quad W = U^2/R;$$

$$F3: R = U^2/W; \quad I = W/U;$$

$$F4 : U = IR; \quad W = I^2R;$$

$$F5 : U = \sqrt{RW}; \quad I = \sqrt{W/R};$$

$$F6: R = W/I^2; \quad U = W/I.$$

Тогда табличную форму (таблицу решений) записи алгоритма решающего соответствующую задачу можно представить в следующем виде (рис. 36).

TP5	R1	R2	R3	R4	R5	R6	E
W	Y	Y	Y	-	-	-	
U	Y	-	-	Y	Y	-	
I	-	Y	-	Y	-	Y	
R	-	-	Y		Y	Y	
ИСПОЛЬЗУЙ ОШИБКА	F3	F6	F5	F1	F2	F4	X

Рис. 36. Таблица решений

Запись алгоритма в форме диаграммы

Пример записи алгоритма в виде диаграмм в CASE-технологии *Silverrun* (рис. 37).

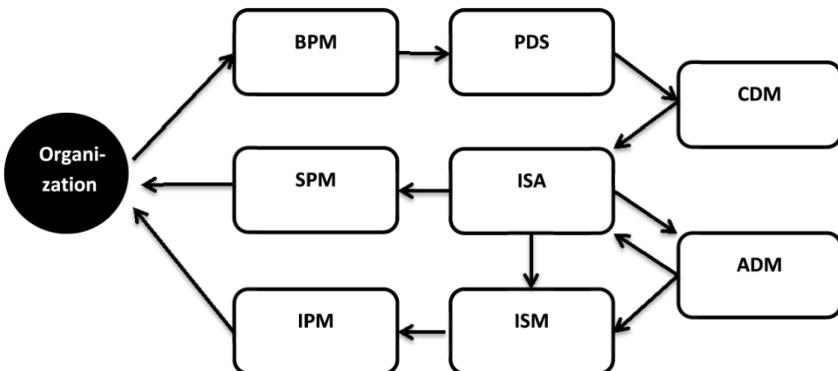


Рис. 37. Запись алгоритма в CASE-технологии

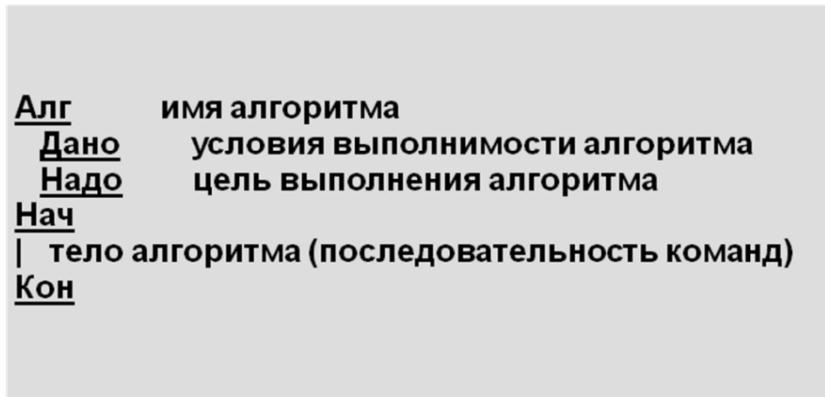
Здесь:

- **BPM** (Business Process Model) – модель бизнес-процессов.
- **PDS** (Primary Data Structure) – структура первичных данных.
- **CDM** (Conceptual Data Model) – концептуальная модель данных.
- **SPM** (System Process Model) – модель процессов системы.
- **ISA** (Information System Architecture) – архитектура информационной системы.
- **ADM** (Application Data Model) – модель данных приложения.
- **IPM** (Interface Presentation Model) – модель представления интерфейса.
- **ISM** (Interface Specification Model) – модель спецификации интерфейса.

Псевдокод

Псевдокод представляет собой систему обозначений и правил, предназначенную для единообразной записи алгоритмов. Он занимает промежуточное место между естественным и формальным языком.

Запись алгоритма на псевдокоде (рис. 38, *например*, на школьном алгоритмическом языке) выглядит



Алг имя алгоритма
Дано условия выполнимости алгоритма
Надо цель выполнения алгоритма
Нач
| тело алгоритма (последовательность команд)
Кон

Рис. 38. Запись алгоритма на псевдокоде

Школьный алгоритмический язык

Начало школьному алгоритмическому языку положил академик А.П. Ершов (рис 39).

Школьный алгоритмический язык

- **алг** название алгоритма (аргументы и результаты)
 - дано** условия применимости алгоритма
 - надо** цель выполнения алгоритма
 - нач** описание промежуточных величин
- последовательность команд (тело алгоритма)
- кон**

```
алг КВУР (арг веш a, b, c, рез веш x1, x2, рез лит t)
  дано a < > 0
  нач веш D
    D := b**2 - 4*a*c
    если D < 0
      то t := "Действительных корней нет"
      иначе
        если D=0
          то t := "Корни равны"; x1 := -b/(2*a); x2 := x1
          иначе t := "Два корня"
            x1 := (-b + sqrt(D)) / (2*a)
            x2 := (-b - sqrt(D)) / (2*a)
        все
    все
кон
```

Рис. 39. Алгоритм вычисления квадратного корня на школьном алгоритмическом языке

Рассмотрим алгоритм Герона (приближенное вычисление корня квадратного), записанного на псевдокоде.

Задача 1. Рассмотрим алгоритм Герона для вычисления квадратного корня. Пусть x_n есть приближенное значение \sqrt{x} . Тогда следующее приближение равняется

$$x_{n+1} = \left(\frac{x}{x_n} + x_n \right) / 2.$$

Если $x_n \geq \sqrt{x}$, то $x_n \geq x_{n+1} \geq \sqrt{x}$. Если ε_n обозначает относительную ошибку приближения x_n , т. е. $\varepsilon_n = (x_n - \sqrt{x}) / \sqrt{x}$, то при $x_n \geq \sqrt{x} > 0$

$$\varepsilon_{n+1} = \frac{\varepsilon_n^2}{2(1 + \varepsilon_n)}.$$

(проверить!) и, следовательно, $\varepsilon_{n+1} < \varepsilon_n/2$ всегда, а при $\varepsilon_n < 0,1$ число верных знаков на каждой итерации примерно удваивается (если $\varepsilon_n < 0,1$, то $\varepsilon_{n+1} < 0,01; \varepsilon_{n+2} < 0,0001, \dots$).

$$\begin{array}{ll}
 1) & I_n + x \Rightarrow V \\
 2) & V + 0 \Rightarrow R2 \\
 3) & x / V \Rightarrow (x/V) \\
 4) & (x/V) + V \Rightarrow (2V) \\
 5) & (2V) \times (1/2) \Rightarrow V \\
 6) & V < R2
 \end{array}
 \quad \boxed{ }$$

Исполнители алгоритмов

Существует три основных класса арифметических моделей исполнителей алгоритмов.

Первый класс моделей основан на арифметизации алгоритмов. Предполагается, что любые данные можно закодировать числами, как следствие – всяко их преобразование становится в этом случае арифметическим вычислением, алгоритмом в таких моделях ее вычисление значения некоторой числовая функции, а его элементы – тарные шаги – арифметические операции. Последовательность шагов определяется двумя способами. *Первый способ* – суперпозиция, т.е. подстановка функции в функцию, а *второй – рекурсия*, т.е. определение значения функции через «ранее» вычисленные значения этой же функции. Функции, которые можно построить из целых чисел и арифметических операций с помощью суперпозиций рекурсивных определений, называются *рекурсив-*

ными функциями. Простейшим примером рекурсивной функции является факториал.

$$(n+1)! = n!(n+1)$$

Второй класс моделей порожден следующей идеей. Для того чтобы алгоритм понимался однозначно, а его каждый шаг считался элементарным и выполнимым, он должен быть представлен так чтобы его могла выполнять машина, к которой предъявляются уже упомянутые требования простоты и универсальности. Одной из таких машин явилась *абстрактная машина Тьюринга*. Машина Тьюринга состоит из трех частей ленты, головки и управляющего устройства (УУ). Лента бесконечна в обе стороны и разбита на ячейки. Элементарный шаг машины Тьюринга состоит из следующих действий:

- головка считывает символ, записанный в ячейке, над которой она находится;
- считанный символ и текущее состояние головки однозначно определяют новое состояние, новый записываемый символ и перемещение головки (которое может иметь значение на ячейку влево, на ячейку вправо, оставаться на месте).

Устройство управления хранит и выполняет команды машины.

Третий класс моделей алгоритмов очень близок к предыдущему, но не оперирует конкретными машинными механизмами. Наиболее известной алгоритмической моделью этого типа служит *нормальные алгоритмы Маркова*.

Для нормального алгоритма задается алфавит, над которым он работает, конечное множество допустимых подстановок и порядок их применения. Если в качестве алфавита взять алфавит русского языка, а в качестве множества подстановок то, используя следующие правила 1–3:

- 1) проверить возможность подстановок в порядке возрастания их номеров, и если она возможна (левая часть подстановки обнаружена в исходном слове), произвести подстановку (заменив левую часть на правую);
- 2) если в примененной подстановке имеется символ «!», то преобразования прекращаются, а если нет, то текущее состояние становится исходным и весь процесс начинается заново;

3) если ни одна подстановка не применима, то процесс преобразования завершен, и можно обнаружить по заданному алгоритму исходное слово.

В теории алгоритмов строго доказано, что по своим возможностям преобразования нормальные алгоритмы эквивалентные машине Тьюринга и другим моделям, уточняющим понятия алгоритма.

Заметим, что при решении задач алгоритм обеспечивает возможность получения результата, но не сам результат. Чтобы получить его алгоритм должен быть исполнен

Исполнитель алгоритма – это некоторая абстрактная или реальная (техническая, биологическая или биотехническая) система, способная выполнить действия, предписываемые алгоритмом.

Исполнителя характеризует¹:

- среда;
- элементарные действия;
- система команд;
- отказы.

Среда (или обстановка) – это «место обитания» исполнителя. Например, для исполнителя «Робота» из школьного учебника, среда – это бесконечное клеточное поле. Стены и закрашенные клетки тоже часть среды. А их расположение и положение самого «Робота» задают конкретное состояние среды.

Система команд. Каждый исполнитель может выполнять команды только из некоторого строго заданного списка – системы команд исполнителя. Для каждой команды должны быть заданы условия применимости (в каких состояниях среды может быть выполнена команда) и описаны результаты выполнения команды. Например, команда «Робота» «вверх» может быть выполнена, если выше «Робота» нет стены. Ее результат – смещение «Робота» на одну клетку вверх.

После вызова команды исполнитель совершает соответствующее элементарное действие.

Отказы исполнителя возникают, если команда вызывается при недопустимом для нее состоянии среды.

¹ См., например, http://book.kbsu.ru/theory/chapter7/1_7.html.

Некоторые именованные алгоритмы

Здесь уместно описать некоторые знаменитые именованные алгоритмы, получившие имена ученых их создавших.

Алгоритм Ньютона

Вычисление квадратного корня обеспечивает быстросходящийся алгоритм Ньютона.

Исаак Ньютон разработал метод извлечения квадратного корня, который восходил еще к Герону Александрийскому (около 100 г. н.э.). Метод этот (известный как метод Ньютона) заключается в следующем.

Пусть A_1 – первое приближение числа $\text{sqrt}(X)$ (в качестве A_1 можно брать значения квадратного корня из натурального числа – точного квадрата, не превосходящего X). На практике можно брать и 1, все равно будет сходимость. Рекомендуется взять $X/2$, так итераций будет меньше.

Следующее, более точное приближение числа найдется по формуле:

$$A_2 = 0.5 * (A_1 + X/A_1)$$

Третье, еще более точное приближение:

$$A_3 = 0.5 * (A_2 + X/A_2)$$

Таким образом, $(i+1)$ приближение найдется по следующей формуле:

$$A_{i+1} = 0.5 * (A_i + x/A_i)$$

Можно задавать точность расчета, то есть считать до тех пор, пока предыдущее значение и текущее значение не будут различаться на конкретную величину ϵ , например 0.000001.

$$|A_{i+1} - A_i| \leq \epsilon, \text{ где } \epsilon = 0.000001$$

Следовательно, алгоритм Ньютона для вычисления корня квадратного ($A=\text{Sqrt}(x)$) представляет быстро сходящуюся (при хорошем начальном приближении) серию итераций.

Алгоритм Евклида – нахождение наибольшего общего делителя

Наибольший общий делитель (НОД) – это число, которое делит без остатка два числа и делится само без остатка на любой другой делитель данных двух чисел. Проще говоря, это самое большое число, на которое можно без остатка разделить два числа, для которых ищется НОД.

Нахождение НОД методом деления.

1. Большее число делим на меньшее.
2. Если делится без остатка, то меньшее число и есть НОД (следует выйти из цикла).
3. Если есть остаток, то большее число заменяем на остаток от деления.
4. Переходим к пункту 1.

Реализация алгоритма Евклида делением

Найти НОД для 30 и 18.

$30 / 18 = 1$ (остаток 12)

$18 / 12 = 1$ (остаток 6)

$12 / 6 = 2$ (остаток 0)

Конец: НОД – это делитель 6.

НОД (30, 18) = 6

```
a = 50  
b = 130
```

```
while a != 0 and b != 0:  
    if a > b:  
        a = a % b  
    else:  
        b = b % a  
  
print(a + b)
```

В цикле в переменную a или b записывается остаток от деления. Цикл завершается, когда хотя бы одна из переменных равна

нулю. Это значит, что другая содержит НОД. Однако какая именно, мы не знаем. Поэтому для НОД находим сумму этих переменных. Поскольку в одной из переменных ноль, он не оказывает влияние на результат.

Реализация алгоритма Евклида вычитанием

1. Из большего числа вычитаем меньшее.
2. Если получается 0, то значит, что числа равны друг другу и являются НОД (следует выйти из цикла).
3. Если результат вычитания не равен 0, то большее число заменяем на результат вычитания.
4. Переходим к пункту 1.

Пример:

Найти НОД для 30 и 18.

$$30 - 18 = 12$$

$$18 - 12 = 6$$

$$12 - 6 = 6$$

$$6 - 6 = 0$$

Конец: НОД – это уменьшаемое или вычитаемое.

$$\text{НОД}(30, 18) = 6$$

```
a = 50  
b = 130
```

```
while a != b:  
    if a > b:  
        a = a - b  
    else:  
        b = b - a  
  
print(a)
```

Алгоритм Дейкстры

Рассмотрим, на примере, нахождение кратчайшего пути¹ в графе. Пусть дана сеть автомобильных дорог, соединяющих области города. Некоторые дороги односторонние. Найти кратчайшие пути от центра города до каждого города области.

Для решения указанной задачи можно использовать **алгоритм Дейкстры** – алгоритм на графах, изобретённый нидерландским ученым Э. Дейкстрой в 1959 году. Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Работает только для графов без рёбер отрицательного веса.

Пусть требуется найти кратчайшие расстояния от 1-й вершины до всех остальных.

Кружками обозначены вершины, линиями – пути между ними (ребра графа). В кружках обозначены номера вершин, над ребрами обозначен их вес – длина пути. Рядом с каждой вершиной красным обозначена метка – длина кратчайшего пути в эту вершину из вершины 1.

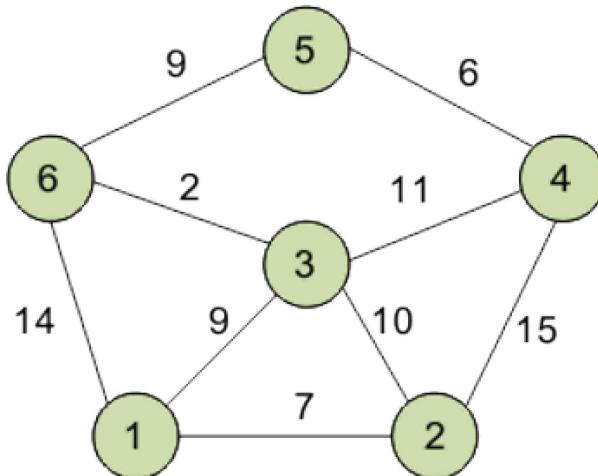
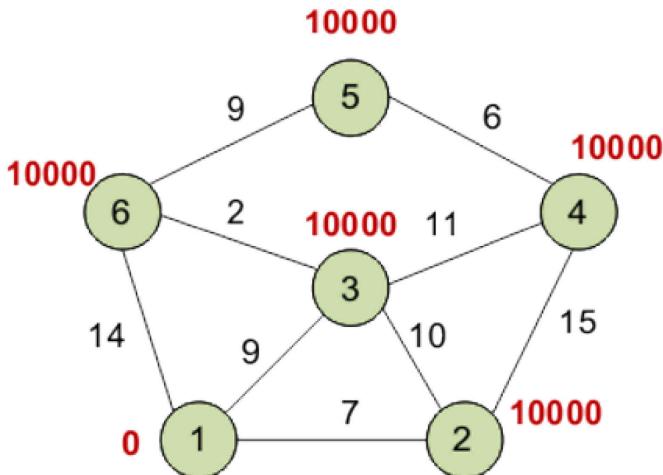


Рис. 40

¹ См. <https://prog-cpp.ru/deikstra/>

Инициализация

Метка самой вершины 1 полагается равной 0, метки остальных вершин – недостижимо большое число (в идеале – бесконечность). Это отражает то, что расстояния от вершины 1 до других вершин пока неизвестны. Все вершины графа помечаются как не-посещенные.



Первый шаг

Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6. Обходим соседей вершины по очереди.

Первый сосед вершины 1 – вершина 2, потому что длина пути до неё минимальна. Длина пути в неё через вершину 1 равна сумме кратчайшего расстояния до вершины 1 (значению её метки) и длины ребра, идущего из 1-й во 2-ю, то есть $0 + 7 = 7$. Это меньше текущей метки вершины 2 (10000), поэтому новая метка 2-й вершины равна 7.

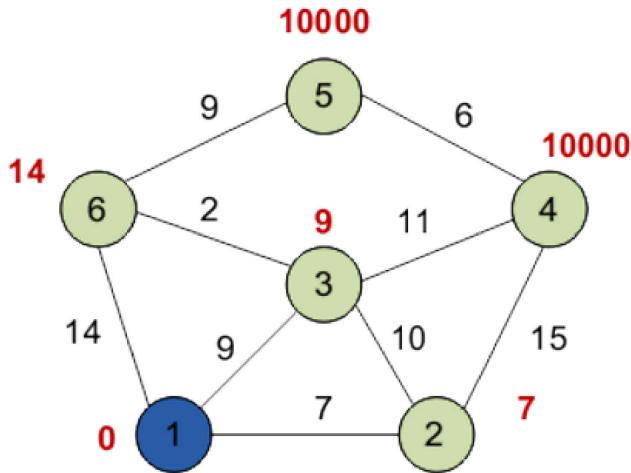


Рис. 42.

Аналогично находим длины пути для всех других соседей (вершины 3 и 6).

Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотрю не подлежит. Вершина 1 отмечается как посещенная.

Второй шаг

Шаг 1 алгоритма повторяется. Снова находим «ближайшую» из непосещенных вершин. Это вершина 2 с меткой 7.

Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.

Вершина 1 уже посещена. Следующий сосед вершины 2 – вершина 3, так как имеет минимальную метку из вершин, отмеченных как не посещённые. Если идти в неё через 2, то длина такого пути будет равна 17 ($7 + 10 = 17$). Но текущая метка третьей вершины равна 9, а $9 < 17$, поэтому метка не меняется.

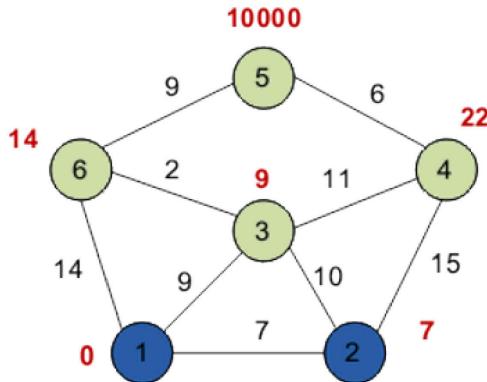


Рис. 43

Ещё один сосед вершины 2 – вершина 4. Если идти в неё через 2-ю, то длина такого пути будет равна 22 ($7 + 15 = 22$). Поскольку $22 < 10000$, устанавливаем метку вершины 4 равной 22.

Все соседи вершины 2 просмотрены, помечаем её как посещенную.

Третий шаг

Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим следующие результаты.

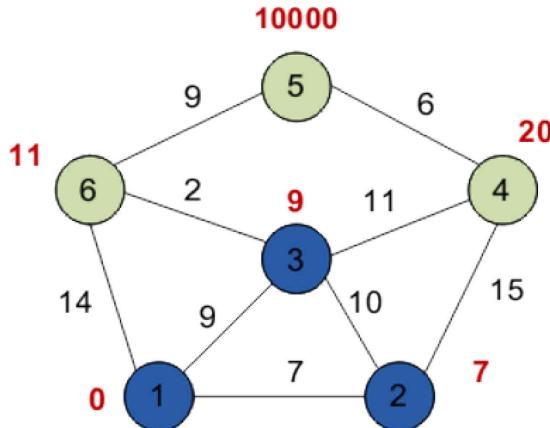


Рис. 44

Четвертый шаг

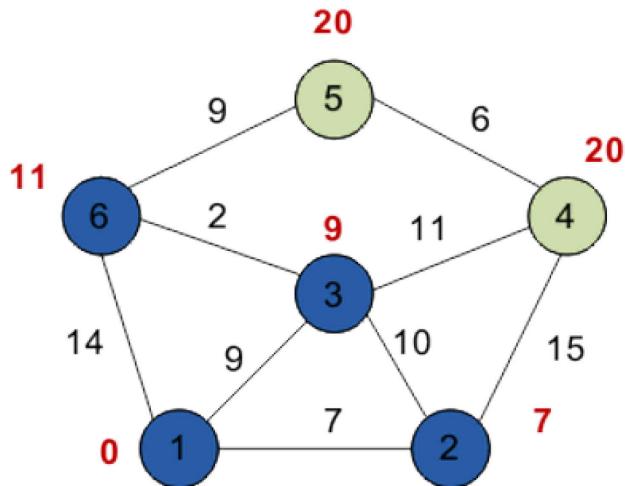


Рис. 45

Пятый шаг

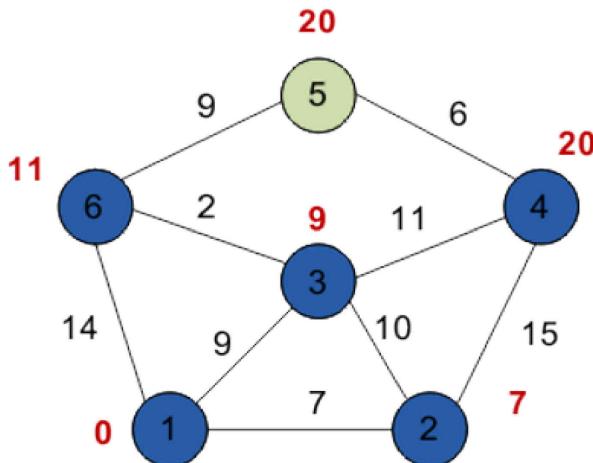


Рис. 46

Шестой шаг

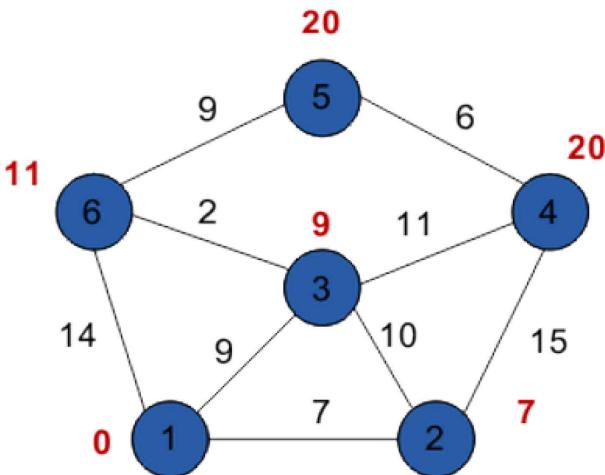


Рис. 47

Таким образом, кратчайшим путем из вершины 1 в вершину 5 будет путь через вершины 1 – 3 – 6 – 5, поскольку таким путем мы набираем минимальный вес, равный 20.

Займемся выводом кратчайшего пути. Мы знаем длину пути для каждой вершины, и теперь будем рассматривать вершины с конца. Рассматриваем конечную вершину (в данном случае – вершина 5), и для всех вершин, с которой она связана, находим длину пути, вычитая вес соответствующего ребра из длины пути конечной вершины.

Так, вершина 5 имеет длину пути 20. Она связана с вершинами 6 и 4.

Для вершины 6 получим вес $20 - 9 = 11$ (совпал).

Для вершины 4 получим вес $20 - 6 = 14$ (не совпал).

Если в результате мы получим значение, которое совпадает с длиной пути рассматриваемой вершины (в данном случае – вершина 6), то именно из нее был осуществлен переход в конечную вершину. Отмечаем эту вершину на искомом пути.

Далее определяем ребро, через которое мы попали в вершину 6. И так пока не дойдем до начала.

Если в результате такого обхода у нас на каком-то шаге совпадут значения для нескольких вершин, то можно взять любую из них – несколько путей будут иметь одинаковую длину.

Минимальное оставное дерево

Оставным деревом называют такой граф, у которого из каждой вершины в любую другую имеется ровно один путь и нельзя пройти все вершины, не пройдя по одному и тому же ребру дважды.

Оставным деревом графа называется дерево, которое можно получить из него путём удаления некоторых рёбер. У графа может существовать несколько оставных деревьев, и чаще всех их достаточно много.

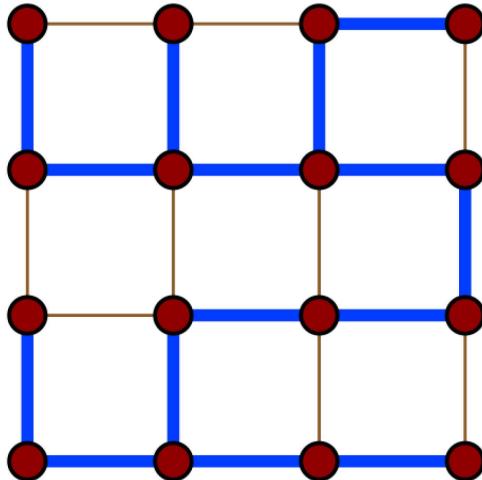


Рис. 48

На иллюстрации приведено одно из оставных деревьев (рёбра выделены синим цветом) решёткообразного графа.

Для взвешенных графов существует понятие веса оставного дерева, которое определено как сумма весов всех рёбер, входящих в оставное дерево. Из этого вытекает понятие *минимального оставного дерева*, т.е. дерева с минимальным возможным весом.

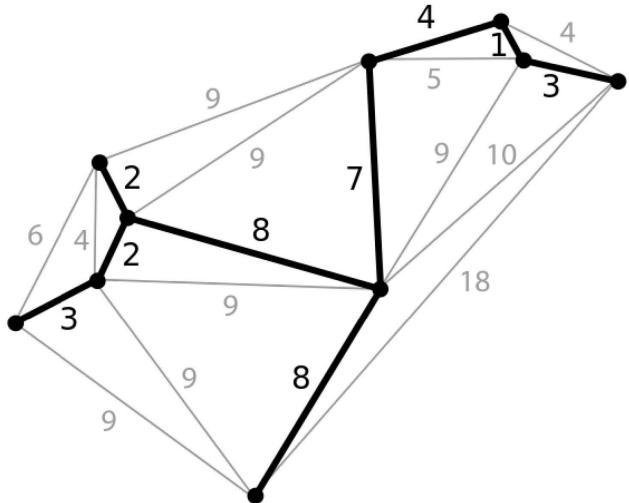


Рис. 49

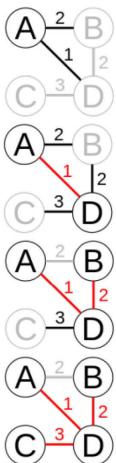
Для нахождения минимального оствовного дерева графа существуют два основных алгоритма: алгоритм Прима и алгоритм Крускала. Они оба имеют сложность $O(M \log N)$, поэтому выбор одного из них зависит от ваших личных предпочтений. Мы рассмотрим оба.

Алгоритм Прима

Алгоритм Прима¹ в идее и реализации очень похож на алгоритм Дейкстры. Как и в алгоритме Дейкстры, мы поддерживаем уже обработанную часть графа (минимального оствовного дерева), и постепенно её расширяем за счёт ближайших вершин.

Утверждается, что если разделить вершины графа на два множества (обработанные и необработанные), первое из которых составляет связную часть минимального оствовного дерева, то ребро минимальной длины, связывающее эти два множества гарантированно будет входить в минимальное оствовное дерево.

¹ См. <https://brestprog.by/topics/mst/>



Таким образом, для нахождения минимального оствовного дерева начнём с произвольной вершины и будем постепенно добавлять ближайшие к уже имеющимся.

На иллюстрации красным цветом выделены рёбра, уже вошедшие в минимальный оствов, а чёрным – текущие кандидаты, из которых выбирается ребро с минимальным весом.

Рис. 50. Формирование оствовного дерева

Реализация алгоритма Прима

Ниже представлена реализация алгоритма Прима. Вначале ищется вес минимального оствовного дерева. Для нахождения ближайшей вершины воспользуемся очередью с приоритетом (аналогично алгоритму Дейкстры), в которой будем хранить пары (расстояние от оствова до вершины, номер вершины).

```

1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 const int INF = 1e9 + 7;
6
7 vector<pair<int, int>> graph[100000];
8 bool used[100000]; //включили ли мы
9 //соответствующую вершину в оствов
10
11 int main() {
12     //Ввод графа...
13
14     int mst_weight = 0; //Текущий вес оствова.

```

```

15 priority_queue<pair<int, int>, vector<pair<int, int>>,
16 greater<pair<int, int>>> q;
17
18 q.push({0, 0}); //Начнём с вершины 0.
19
20 while (!q.empty()) {
21     pair<int, int> c = q.top();
22     q.pop();
23
24     int dst = c.first, v = c.second;
25
26     if (used[v]) { //вершина уже добавлена в остов
27         continue;
28     }
29
30     used[v] = true;
31     mst_weight += dst;
32
33     for (pair<int, int> e: graph[v]) {
34         int u = e.first, len_vu = e.second;
35
36         if (!used[u]) {
37             q.push({len_vu, u}); //Заметьте: мы
38             //учитываем только длину ребра.
39         }
40     }
41 }
42
43 cout << "Minimum spanning tree weight: " <<
44 mst_weight << endl;
45 }
```

Алгоритм Крускала

Алгоритм Крускала достаточно прост в своей идеи и реализации. Он заключается в сортировке всех рёбер в порядке возрастания длины, и поочерёдному добавлению их в минимальный остов, если они соединяют различные компоненты связности.

Более формально: пусть мы уже нашли некоторые рёбра, входящие в минимальный остов. Утверждается, что среди всех рёбер, соединяющих различные компоненты связности, в минимальный остов будет входить ребро с минимальной длиной.

Для реализации алгоритма Крускала необходимо уметь сортировать рёбра по возрастанию длины (для этого воспользуемся собственным типом данных) и проверять, соединяет ли ребро две различных компоненты связности. Для этого будем просто поддерживать текущие компоненты связности с помощью структуры данных DSU.

Визуализация работы алгоритма Крускала представлена на рис.51:

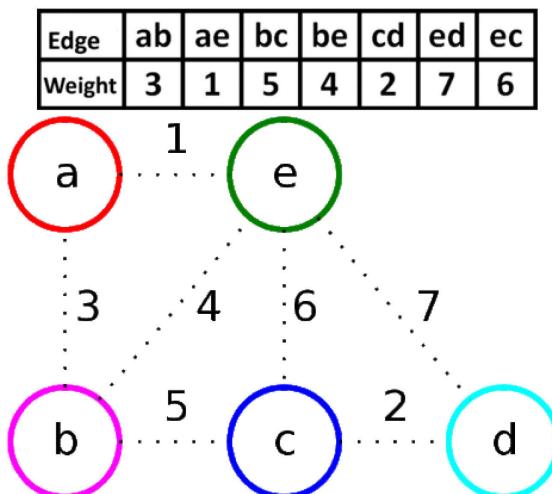


Рис. 51

Реализация алгоритма Крускала

Ниже представлена программа, реализующая алгоритм Крускала:

```
1 #include <bits/stdc++.h>
2
3 using namespace std;
4
5 int p[100000];
6 int rk[100000];
7
8 void init_dsu() {
9     for (int i = 0; i < 100000; i++) {
10         p[i] = i;
11         rk[i] = 1;
12     }
13 }
14
15 int get_root(int v) {
16     if (p[v] == v) {
17         return v;
18     } else {
19         return p[v] = get_root(p[v]); //На выходе из
20         рекурсии непроподвешиваем v
21     }
22 }
23
24 bool merge(int a, int b) {
25     int ra = get_root(a), rb = get_root(b);
26
27     if (ra == rb) {
28         return false;
29     } else {
30         if (rk[ra] < rk[rb]) {
```

```

31     p[ra] = rb;
32 } else if (rk[rb] < rk[ra]) {
33     p[rb] = ra;
34 } else {
35     p[ra] = rb;
36     rk[rb]++;
37 }
38
39     return true;
40 }
41 }
42
43 //Тип для представления рёбер.
44 struct edge {
45     int a, b, len;
46
47     bool operator<(const edge& other) {
48         return len < other.len;
49     }
50 };
51
52 int main() {
53     vector<edge> edges;
54     //Ввод edges...
55
56     sort(edges.begin(), edges.end());
57
58     int mst_weight = 0;
59
60     init_dsu();
61
62     for (edge e: edges) {
63         if (merge(e.a, e.b)) {    //Если a и b находятся в
64             разных компонентах,

```

```

65         mst_weight += e.len; //Добавить ребро в
66         минимальный остов.
67     }
68 }

cout << "Minimum spanning tree weight: " <<
mst_weight << endl;
}

```

Алгоритм Беллмана – Форда

Пусть дан ориентированный взвешенный граф G с n вершинами и m рёбрами, и указана некоторая вершина v . Требуется найти длины кратчайших путей от вершины v до всех остальных вершин.

В отличие от алгоритма Дейкстры, этот алгоритм применим также и к графикам, содержащим рёбра отрицательного веса. Впрочем, если график содержит отрицательный цикл, то, понятно, кратчайшего пути до некоторых вершин может не существовать (по причине того, что вес кратчайшего пути должен быть равен минус бесконечности); впрочем, этот алгоритм можно модифицировать, чтобы он сигнализировал о наличии цикла отрицательного веса, или даже выводил сам этот цикл.

Алгоритм носит имя двух американских учёных¹: Ричарда Беллмана (Richard Bellman) и Лестера Форда (Lester Ford). Форд фактически изобрёл этот алгоритм в 1956 г. при изучении другой математической задачи, подзадача которой свелась к поиску кратчайшего пути в графике, и Форд дал набросок решающего эту задачу алгоритма. Беллман в 1958 г. опубликовал статью, посвящённую конкретно задаче нахождения кратчайшего пути, и в этой статье он чётко сформулировал алгоритм в том виде, в котором он известен нам сейчас.

Мы считаем, что график не содержит цикла отрицательного веса. Случай наличия отрицательного цикла рассматривается отдельно.

¹ https://e-maxx.ru/algo/ford_bellman

Заведём массив расстояний $d[0 \dots n - 1]$, который после отработки алгоритма будет содержать ответ на задачу. В начале работы мы заполняем его следующим образом: $d[v] = 0$, а все остальные элементы $d[]$ = бесконечности ∞ .

Сам алгоритм Форда-Беллмана представляет из себя несколько фаз. На каждой фазе просматриваются все рёбра графа, и алгоритм пытается произвести *релаксацию* (relax, ослабление) вдоль каждого ребра (a, b) стоимости c . Релаксация вдоль ребра – это попытка улучшить значение $d[b]$ значением $d[a] + c$. Фактически это значит, что мы пытаемся улучшить ответ для вершины b , пользуясь ребром (a, b) и текущим ответом для вершины a .

Утверждается, что достаточно $n - 1$ фазы алгоритма, чтобы корректно посчитать длины всех кратчайших путей в графе (повторимся, мы считаем, что циклы отрицательного веса отсутствуют). Для недостижимых вершин расстояние $d[]$ останется равным бесконечности ∞ .

Давайте разберемся в алгоритме на следующем примере графа. Пусть начальная вершина равна 0. Примите все расстояния за бесконечные, кроме расстояния до самой `src`. Общее число вершин в графе равно 5, поэтому все ребра нужно пройти 4 раза (см. рис. 52-54).

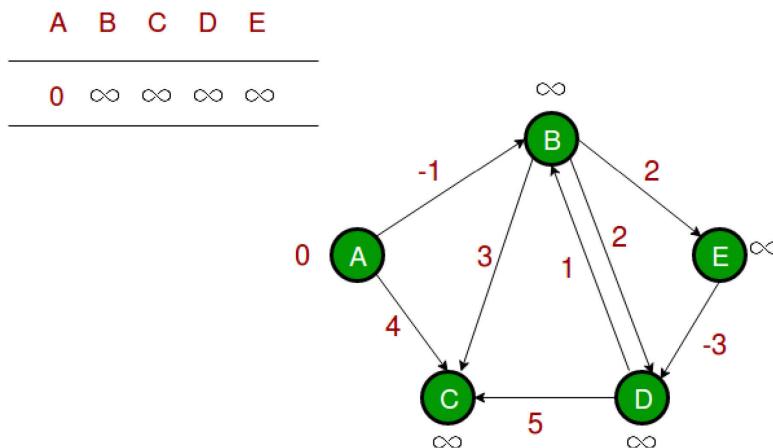


Рис. 52

Пусть ребра отрабатываются в следующем порядке: (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). Мы получаем следующие расстояния, когда проход по ребрам был совершен первый раз. Первая строка показывает начальные расстояния, вторая строка показывает расстояния, когда ребра (B, E), (D, B), (B, D) и (A, B) обрабатываются. Третья строка показывает расстояние при обработке (A, C). Четвертая строка показывает, что происходит, когда обрабатываются (D, C), (B, C) и (E, D).

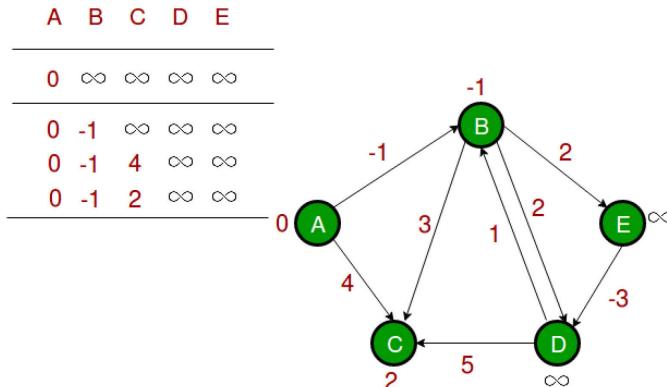


Рис. 53

Первая итерация гарантирует, что все самые короткие пути будут не длиннее путей в 1 ребро. Мы получаем следующие расстояния, когда будет совершен второй проход по всем ребрам (в последней строке показаны конечные значения).

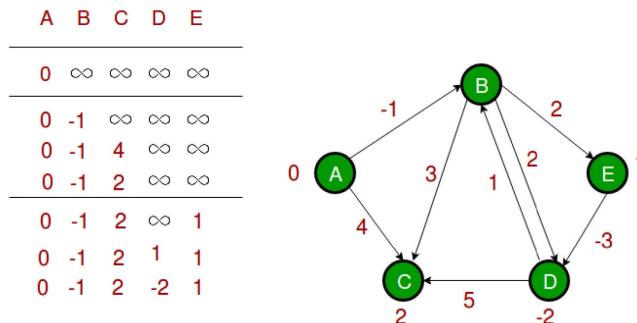


Рис. 54

Вторая итерация гарантирует, что все кратчайшие пути будут иметь длину не более 2 ребер. Алгоритм проходит по всем ребрам еще 2 раза. Расстояния минимизируются после второй итерации, поэтому третья и четвертая итерации не обновляют значения расстояний.

Реализация алгоритма Форда-Беллмана

```
void solve() {
    vector<int> d (n, INF);
    d[v] = 0;
    for (;;) {
        bool any = false;
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    any = true;
                }
        if (!any) break;
    }
    // вывод d, например, на экран
}
```

Алгоритм Хаффмана

Избавиться от нерационального кодирования нам поможет *алгоритм Хаффмана (Huffman)*, формирующий код с наименьшей средней длиной¹. Суть его в следующем: все содержащиеся в файле символы записываются в список по возрастанию числа повторов. Два последних объединяются в новый составной узел, число повторов которого равно сумме повторов исходных символов. Формируется новый список, и вновь два последних значения повторов объединяются в новый узел. Данные манипуляции повторяются до тех пор, пока не останется единственное значение. И оно будет равно числу повторов всех символов, из которых состоит файл. В результате мы построим дерево, каждый узел кото-

¹ См. <http://www.univer.omsk.su/omsk-old/Edu/infpro/1/mkomp/xaffm.html>

рого имеет суммарное значение числа повторов всех узлов, находящихся ниже.

Процесс эффективного кодирования Huffman'a отображен в Таблице 4. Чтобы сформировать код для любого символа, проследите его «путь» по колонкам и строкам от начала до конца.

Таблица 4
Вспомогательная таблица количества повторов символов

c	22	22	22	22	32	42	58	100
e	20	20	20	22	26	32	42	
h	16	16	16	20	22	26		
l	16	16	16	16	20			
a	10	10	16	16				
k	10	10	10					
m	4	6						
b	2							

В созданном дереве от ячейки, в которой указана сумма всех повторов (в данном случае она равна 100), мы ведем две ветки. Ветке с большим значением повторов мы назначаем код длиной 1, а ветке с меньшим – 2. Двигаясь по дереву сверху вниз, присваиваем каждому символу собственный уникальный код.

Именно алгоритм Хаффмана применяется для создания таких типов архивов, как PKZIP, LHA, ZOO, ARJ.

Теперь понятно, как осуществляется сжатие файла. Если кратко, то это достигается так: назначайте самые короткие коды наиболее повторяющимся символам. А более длинные коды присваиваются символам, реже встречающимся в файле. В итоге мы получаем следующее: разные символы имеют различную длину кода, что может значительно затруднить распаковку архива. Неплохо было бы ввести разделительный символ, он будет сигнализировать о начале следующего кода. Но это невозможно по двум причинам. Во-первых, в нашем случае комбинации кодов объединяются в байты, а выбранный

c	01
e	00
h	111
l	110
a	100
k	1011
m	10101
b	10100

Таблица 5

разделительный символ может нарушить целостность частей кода – то есть все закончится большой путаницей. Во-вторых, такой метод увеличит размер архива еще на один байт, и пользы от работы не будет.

Получается, наиболее эффективно – обеспечить синонимичное кодирование без каких-либо дополнительных символов. Итак, от нас требуется найти такой код, чтобы ни одна из его комбинаций не могла породить более длинную комбинацию, – он называется *префиксным*.

Например, у нас есть следующие символы и такие коды:

a	0
b	10
c	110
d	111

Таблица 6

Тогда строка 0011011111101010 может быть интерпретирована как

0	0	110	111	111	0	10	10
a	a	c	d	d	a	b	b

Таблица 7

Но если бы у нас был такой код:

a	00
b	01
c	101
d	010

Таблица 8

Где код символа «b» может быть воспринят как начало кода символа «d», то есть три возможных варианта раскодирования строки 000101010101:

00	01	01	01	010	101
a	b	B	b	d	c

Таблица 9

00	010	101	010	101
a	d	c	d	c

Таблица 10

00	01	010	101	01	01
a	b	d	c	b	b

Таблица 11

Упражнения.

1. В чем заключается алгоритм Хаффмана? Где он используется?
2. С помощью таблицы 4 закодировать: а-2, в-5, с-7, п-16, е-21 (количество повторов).
3. Придумайте пример префиксного кода.
4. Раскодируйте сами строку: 0011101001111010100, если известно, что есть 4 буквы. Напишите количество их повторов.

Алгоритм Форда – Фалкерсона

Алгоритм Форда-Фалкерсона – алгоритм, решающий задачу нахождения максимального потока в транспортной сети.

Идея алгоритма заключается в следующем. Изначально величине потока присваивается значение 0: $f(u,v) = 0$ для всех u,v из V . Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t , вдоль которого можно послать ненулевой поток). Рассматривается алгоритм, осуществляющий этот поиск с помощью обхода в глубину (dfs). Процесс повторяется, пока можно найти увеличивающий путь.

Пример не сходящегося алгоритма

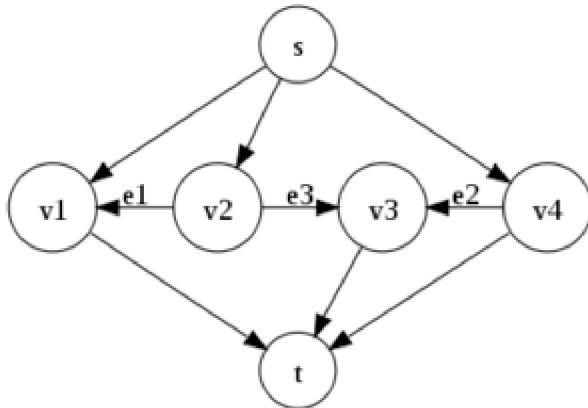


Рис. 55

Рассмотрим приведённую справа сеть (рис. 55) с источником s , стоком t , пропускными способностями рёбер e_1, e_2 и e_3 соответственно 1, $r=(\sqrt{5}-1)/2$ и 1 и пропускной способностью всех остальных рёбер, равной целому числу $M \geq 2$. Константа r выбрана так, что $r^2 = 1 - r$. Мы используем пути из остаточного графа, приведённые в таблице 12, причём $p_1=\{s,v_4,v_3,v_2,v_1,t\}$, $p_2=\{s,v_2,v_3,v_4,t\}$ и $p_3=\{s,v_1,v_2,v_3,t\}$.

Таблица 12

Шаг	Найденный путь	Добавленный поток	Остаточные пропускные способности		
			e_1e_1	e_2e_2	e_3e_3
00	--	--	$r_0=1$	$r_0=1$	11
11	$\{s,v_2,v_3,t\}$ $\{s,v_2,v_3,t\}$	11	r_0r_0	r_1r_1	00
22	p_1p_1	r_1r_1	r_2r_2	00	r_1r_1
33	p_2p_2	r_1r_1	r_2r_2	r_1r_1	00
44	p_1p_1	r_2r_2	00	r_3r_3	r_2r_2
55	p_3p_3	r_2r_2	r_2r_2	r_3r_3	00

Заметим, что после шага 1, как и после шага 5, остаточные способности рёбер e_1, e_2 и e_3 имеют форму r^n, r^{n+1} и 0, соответственно, для какого-то натурального n . Это значит, что мы можем использовать увеличивающие пути p_1, p_2, p_1 и p_3 бесконечно

много раз, и остаточные пропускные способности этих рёбер всегда будут в той же форме. Полный поток после шага 5 равен $1 + 2(r^1 + r^2)$. За бесконечное время полный поток сойдётся к $1 + 2 \sum_{i=1}^{\infty} r = \infty$; $i = 3 + 2r_1 + 2\sum_{i=1}^{\infty} r_i = 3 + 2r$, тогда как максимальный поток равен $2M+1$. Таким образом, алгоритм не только работает бесконечно долго, но даже и не сходится к оптимальному решению.

При использовании поиска в ширину алгоритму потребуется всего лишь два шага. Данна сеть (рис. 56).

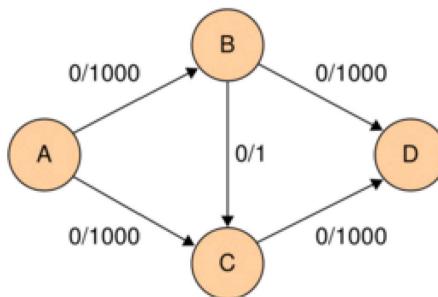


Рис. 56

Благодаря двум итерациям (рис. 57 и рис. 58)

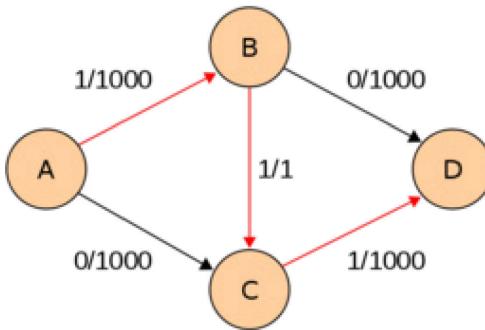


Рис. 57

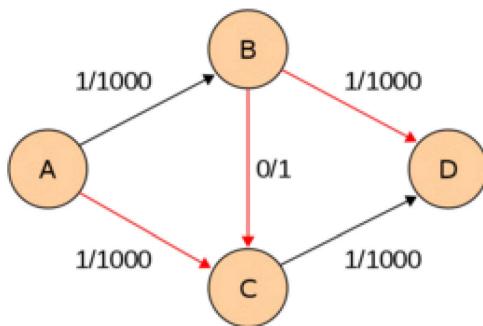


Рис. 58

рёбра AB, AC, BD, CDAB, AC, BD, CD насытились лишь на 11. Конечная сеть будет получена ещё через 1998 итераций (Рис. 59).

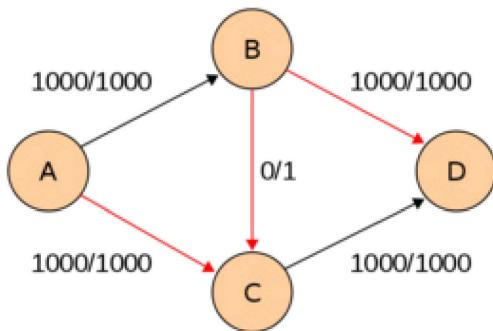


Рис. 59

Сортировка Шелла

Сортировка Шелла¹ является довольно интересной модификацией алгоритма сортировки простыми вставками.

Рассмотрим следующий алгоритм сортировки массива $a[0]..a[15]$.

12	8	14	6	4	9	1	8	13	5	11	3	18	3	10	9
----	---	----	---	---	---	---	---	----	---	----	---	----	---	----	---

1. Вначале сортируем простыми вставками каждые 8 групп из 2-х элементов ($a[0], a[8]$, $(a[1], a[9])$, ..., $(a[7], a[15])$).

12	5	11	3	4	3	1	8	13	8	14	6	18	9	10	9
----	---	----	---	---	---	---	---	----	---	----	---	----	---	----	---

2. Потом сортируем каждую из четырех групп по 4 элемента ($a[0], a[4], a[8], a[12]$), ..., ($a[3], a[7], a[11], a[15]$).

номер группы:

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
4	3	1	3	12	5	10	6	13	8	11	8	18	9	14	9

В нулевой группе будут элементы 4, 12, 13, 18, в первой – 3, 5, 8, 9 и т.п.

3. Далее сортируем 2 группы по 8 элементов, начиная с ($a[0], a[2], a[4], a[6], a[8], a[10], a[12], a[14]$).

1	3	4	3	10	5	11	6	12	8	13	8	14	9	18	9

¹См. http://algolist.ru/sort/shell_sort.php

4. В конце сортируем вставками все 16 элементов.

1	3	3	4	5	6	8	8	9	9	10	11	12	13	14	18
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----

Очевидно, лишь последняя сортировка необходима, чтобы расположить все элементы по своим местам. Так зачем нужны остальные?

На самом деле они продвигают элементы максимально близко к соответствующим позициям, так что в последней стадии число перемещений будет весьма невелико. Последовательность и так почти отсортирована. Ускорение подтверждено многочисленными исследованиями и на практике оказывается довольно существенным.

Единственной характеристикой сортировки Шелла является *приращение* – расстояние между сортируемыми элементами, в зависимости от прохода. В конце приращение всегда равно единице – метод завершается обычной сортировкой вставками, но именно последовательность приращений определяет рост эффективности.

Использованный в примере набор ..., 8, 4, 2, 1 – неплохой выбор, особенно, когда количество элементов – степень двойки. Однако гораздо лучший вариант предложил Р. Седжвик. Его последовательность имеет вид

$$inc[s] = \begin{cases} 9 \times 2^s - 9 \times 2^{\frac{s}{2}} + 1, & \text{если } s \text{ четно} \\ 8 \times 2^s - 6 \times 2^{\frac{s+1}{2}} + 1, & \text{если } s \text{ нечетно} \end{cases}$$

При использовании таких приращений среднее количество операций: $O(n^{7/6})$, в худшем случае – порядка $O(n^{4/3})$.

Обратим внимание на то, что последовательность вычисляется в порядке, противоположном используемому: $inc[0] = 1$, $inc[1] = 5$, ... Формула дает сначала меньшие числа, затем все большие и большие, в то время как расстояние между сортируемыми элементами, наоборот, должно уменьшаться. Поэтому массив приращений inc вычисляется перед запуском собственно сортировки до максимального расстояния между элементами, которое будет пер-

вым шагом в сортировке Шелла. Потом его значения используются в обратном порядке.

При использовании формулы Седжвика следует остановиться на значении $\text{inc}[s-1]$, если $3 * \text{inc}[s] > \text{size}$.

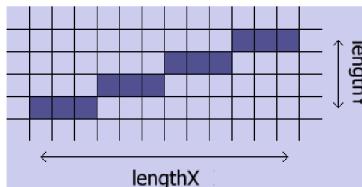
Алгоритм Брезенхэма

Алгоритм Брезенхема (англ. Bresenham's line algorithm) – это алгоритм, определяющий, какие точки двумерного растра нужно закрасить, чтобы получить близкое приближение прямой линии между двумя заданными точками.

Алгоритм широко используется, в частности, для рисования линий на экране компьютера.

Этот алгоритм, разработанный Джеком Е. Брезенхэмом (Jack E. Bresenham) в 1962 году в компании IBM, является одним из самых старых алгоритмов в компьютерной графике. Он позволяет получить приближение идеальной прямой точками растровой сетки.

Рассмотрим произвольный отрезок $p_1(x_1, y_1) - p_2(x_2, y_2)$:



Чтобы растеризовать его можно поступить следующим образом (небольшая модификация алгоритма DDA-линий):

- посчитаем длины отрезка по осям координат $\text{lengthX} = |x_2 - x_1|$ и $\text{lengthY} = |y_2 - y_1|^1$;
- выберем большую из них $\text{length} = \max(\text{lengthX}, \text{lengthY})$;
- случай $\text{length} == 0$ особый: закрашиваем пикセル $(x_1, y_1) = (x_2, y_2)$ и завершаем работу алгоритма;
- допустим большая длина оказалась lengthX . Установим начальную точку $x = x_1$, $y = y_1$;

¹ Обратите внимание, что $|x_2 - x_1|$ и $|y_2 - y_1|$ это расстояния между **центрами** первого и последнего пикселей.

- сделаем $\text{length} + 1$ итераций:
 - закрашиваем пиксель с координатами $(x, \text{roundf}(y))$ ¹;
 - координата x увеличивается на единицу, координата y увеличивается на $\text{lengthY} / \text{lengthX}$.

Алгоритм Эль-Гамаля

Алгоритм Эль-Гамаля может использоваться для формирования электронной подписи или для шифрования данных². Он базируется на трудности вычисления дискретного логарифма.

Схема была предложена Тахером Эль-Гамалем в 1985 году. Алгоритм Эль-Гамаля базируется на трудности вычисления дискретного логарифма.

*Первый этап алгоритма Эль-Гамаля*³ заключается в генерации ключей. Этот этап включает следующую последовательность действий:

1. Генерируется случайное простое число p длины n бит.
2. Выбирается произвольное целое число a , являющееся *первообразным (примитивным) корнем* по модулю p .
3. Выбирается случайное число x из интервала $(1, p)$, взаимно простое с $p-1$.
4. Вычисляется $y = a^x \pmod{p}$.

Открытым ключом является тройка (a, p, y) , закрытым ключом – число x .

Второй этап алгоритма включает шифрование.

Сообщение M шифруется следующим образом:

1. Выбирается случайное секретное число k , взаимно простое с $p - 1$.
2. Вычисляется

$$\gamma = a^k \pmod{p}, \\ \delta = M * y^k \pmod{p},$$

где M – сходное сообщение.

Пара чисел (γ, δ) является *шифртекстом*.

¹ roundf – округление

² http://cryptowiki.net/index.php?title=Схема_Эль-Гамаля

³ Башкегев М.А. 2015

Нетрудно заметить, что длина шифртекста в схеме Эль-Гамаля длиннее исходного сообщения M вдвое.

Заключительный этап схемы Эль-Гамаля – это расшифрование.

Зная закрытый ключ x и учитывая тот факт, что $M = (\delta * \gamma^{p-1-x}) \pmod{p}$, исходное сообщение M можно вычислить из шифртекста (γ, δ) по формуле:

$$M = \gamma^{-x} * \delta \pmod{p}$$

На рисунке 48 приведена схема работы алгоритма шифрования Эль-Гамаля.

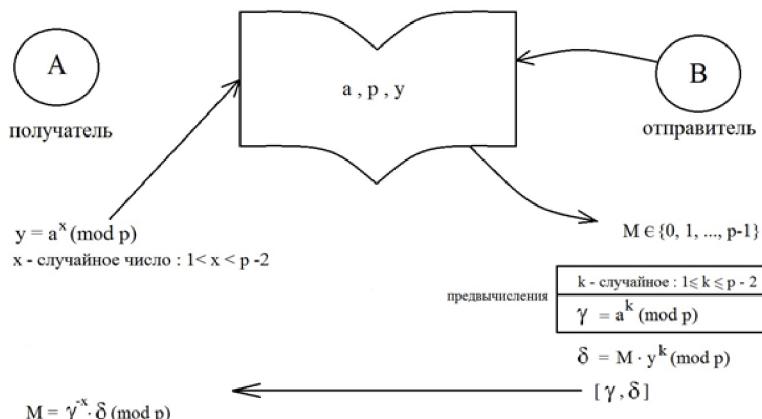


Рис. 60. Схема работы алгоритма Эль-Гамаля

Ввиду того, что число k является произвольным, то такую схему еще называют схемой вероятностного шифрования. Вероятностный характер шифрования является преимуществом для схемы Эль-Гамаля, т.к. у схем вероятностного шифрования наблюдается большая стойкость по сравнению со схемами с определенным процессом шифрования. Недостатком схемы шифрования Эль-Гамаля является удвоение длины зашифрованного текста по сравнению с начальным текстом. Для схемы вероятностного шифрования само сообщение M и ключ не определяют шифртекст однозначно. В схеме Эль-Гамаля необходимо использовать раз-

личные значения случайной величины k для шифровки различных сообщений M и M' .

Если использовать одинаковые k , то для соответствующих шифртекстов (γ, δ) и (γ', δ') выполняется соотношение $\delta * (\delta')^{-1} = M * (M')^{-1} \pmod{p}$. Из этого выражения можно легко вычислить M , если известно M' .

Алгоритм Гаусса – Лежандра

Во втором издании в 4-й части «Теории чисел» Лежандр приводит свою знаменитую эмпирическую формулу вычисления числа π .

$$\pi(x) \approx \frac{x}{\ln x - 1,08366},$$

найденную им 1798 году.

С недавних пор существует элегантная формула для вычисления числа Пи, которую в 1995 году впервые опубликовали Дэвид Бэйли, Питер Борвайн и Саймон Плафф:

$$\sum_{k=0}^{\infty} 16^{-k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

Казалось бы: что в ней особенного – формул для вычисления Пи великое множество: от школьного метода Монте-Карло до труднопостижимого интеграла Пуассона и формулы Франсуа Виета из позднего Средневековья. Но именно на эту формулу стоит обратить особое внимание – она позволяет вычислить n -й знак числа пи без нахождения предыдущих.

Как же работает алгоритм вычисления N -го знака Пи?

К примеру, если нам нужен 1000-й шестнадцатеричный знак числа Пи, мы домножаем всю формулу на 16^{1000} , тем самым обращая множитель, стоящий перед скобками, в $16^{(1000-k)}$. При возведении в степень мы используем двоичный алгоритм возведения в степень или, как будет показано в примере ниже, возведение в степень по модулю. После этого вычисляем сумму нескольких членов ряда. Причём необязательно вычислять много: по мере возрастания k $16^{(N-k)}$ быстро убывает, так что, последующие

члены не будут оказывать влияния на значение искомых цифр). Вот и вся магия – гениальная и простая.

Формула Бэйли-Борвайна-Плаффа была найдена Саймоном Плаффом при помощи алгоритма PSLQ, который был в 2000 году включён в список Top 10 Algorithms of the Century. Сам же алгоритм PSLQ был в свою очередь разработан Бэйли. Вот такой мексиканский сериал про математиков. Кстати, время работы алгоритма – $O(N)$, использование памяти – $O(\log N)$, где N – порядковый номер искомого знака.

Генетические алгоритмы

Генетический алгоритм – это алгоритм, использующий принципы естественного отбора (биология) для нахождения решения задачи. Он является разновидностью эволюционных вычислений. Алгоритм представлен следующей схемой (рис. 61).

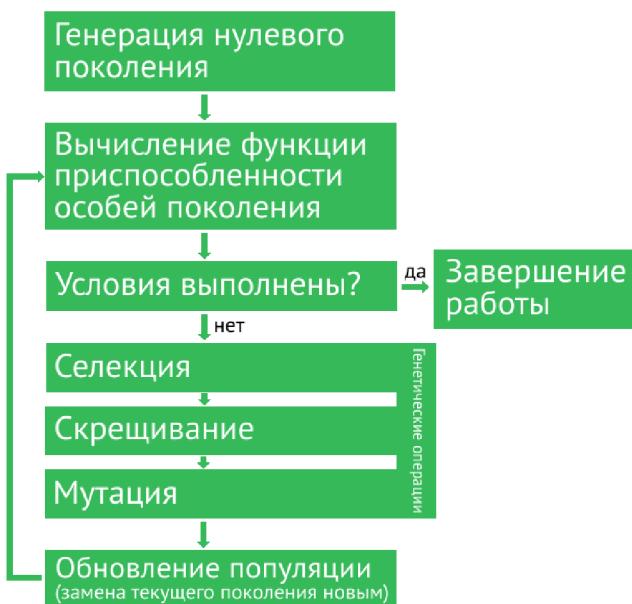


Рис. 61. Генетический алгоритм

Отличительной особенностью генетического алгоритма является использование специального оператора – «*кроссовера*», который производит операцию, роль которой аналогична роли скрещивания в живой природе.

Идея алгоритма следующая.

Задача кодируется таким образом, чтобы её решение могло быть представлено в виде вектора (*«хромосома»*).

Случайным образом создаётся некоторое количество начальных векторов (*«начальная популяция»*).

Они оцениваются с использованием *«функции приспособленности»*, в результате чего каждому вектору присваивается определённое значение (*«приспособленность»*), которое определяет вероятность выживания организма, представленного данным вектором.

После этого с использованием полученных значений приспособленности выбираются вектора (*«селекция»*), допущенные к *«скрещиванию»*.

К этим векторам применяются *«генетические операторы»* (в большинстве случаев *«кроссовер»* и *«мутация»*), создающие следующее *«поколение»*.

Особи следующего поколения также оцениваются, затем производится селекция, применяются генетические операторы и т. д.

Так моделируется *«эволюционный процесс»*, продолжающийся несколько жизненных циклов (поколений), пока не будет выполнен *«критерий останова алгоритма»*.

Таким критерием может быть:

- нахождение глобального, либо субоптимального решения;
- исчерпание числа поколений, отпущеных на эволюцию;
- исчерпание времени, отпущеного на эволюцию.

Применение генетических алгоритмов

Генетические алгоритмы служат, главным образом, для *«поиска решений»* в очень больших, сложных пространствах поиска.

Генетические алгоритмы применяются для решения следующих задач:

- оптимизация функций;
- разнообразные задачи на графах (задача коммивояжера, раскраска, нахождение паросочетаний);
- настройка и обучение искусственной нейронной сети;
- задачи компоновки;
- составление расписаний;
- игровые стратегии;
- аппроксимация функций;
- искусственная жизнь (например, «жизнь муравейника»);
- биоинформатика (свертывание белков).

Методы тестирования алгоритмов

Тестирование программного обеспечения – процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом (ISO/IEC TR 19759:2005).

*Тестирование алгоритма*¹ – это проверка правильности или неправильности работы *алгоритма* на специально заданных тестах или тестовых примерах – задачах с известными входными данными и результатами (иногда достаточны их приближения). *Тестовый набор* должен быть минимальным и полным, то есть обеспечивающим проверку каждого отдельного типа наборов входных данных, особенно исключительных случаев.

Пример. для задачи решения квадратного уравнения $ax^2 + bx + c = 0$ такими исключительными случаями, например, будут: 1) $a = b = c = 0$; 2) $a = 0$, b , c – отличны от нуля; 3) $D = b^2 - 4ac < 0$ и др.

Тестирование алгоритма не может дать полной (100%-ой) гарантии правильности алгоритма для всех возможных наборов входных данных, особенно для достаточно сложных алгоритмов, поскольку этот процесс бесконечен.

¹ См. <https://www.intuit.ru/studies/courses/108/108/lecture/3153>.

Полную гарантию правильности алгоритма может дать описание работы и результатов алгоритма с помощью системы аксиом и правил вывода или *верификация* алгоритма.

Пример. Составим алгоритм нахождения числа всех различных двоичных сообщений (двоичных последовательностей) длины n битов, используя при этом не более одной операции умножения, и докажем правильность этого алгоритма. Вначале найдем число всех таких сообщений. Число двоичных сообщений длины 1 равно $2 = 2^1$ (это «0» и «1»), длины 2 равно $4 = 2^2$ («00», «01», «10», «11»). Отправляясь от этих частных фактов, методом математической индукции докажем, что число различных сообщений длины n равно 2^n . Этот индуктивный вывод докажет правильность алгоритма.

Выделяют структурное и функциональное тестирование алгоритма.

Структурное тестирование основывается на детальном изучении логики алгоритма и подборе тестов, позволяющих обеспечить максимально возможное количество проверяемых операторов, логических ветвлений и условий.

При *функциональном тестировании* логика алгоритма не учитывается, а обращается внимание лишь на входные и выходные спецификации. Одним из наиболее известных методов этой группы является метод *эквивалентного разбиения*. В этом методе сначала выделяются классы эквивалентности, а затем на основе этих классов строятся сами тесты. Для разбиения на классы эквивалентности необходимо проанализировать входные условия и разбить их на два или более класса. Для любого условия существуют правильный класс эквивалентности, который содержит корректные входные данные и неправильный, содержащий ошибочные значения.

Анализ сложности алгоритма

При измерении сложности алгоритмов и структур данных мы обычно говорим о двух вещах: количество операций, требуемых для завершения работы (вычислительная сложность), и объем ресурсов, в частности, памяти, который необходим алгоритму (пространственная сложность). Связано это с тем, что алгоритмы реа-

лизуют в виде программ для вычислений на компьютерах, а в компьютерах существует два дефицитных параметра: тактовая частота центрального процессора (время) и объем оперативной памяти (пространство).

Заметим, что данная оценка сложности алгоритма достаточно субъективна. Так, например, алгоритм, который выполняется в десять раз быстрее, но использует в десять раз больше места, может вполне подходить для серверной машины с большим объемом памяти. Но на встроенных машинных системах, где количество памяти ограничено, такой алгоритм использовать нельзя.

При анализе сложности алгоритма используют такое понятие, как *порядок роста* сложности алгоритма.

Порядок роста

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего он представлен в виде О-нотации (*от нем. «Ordnung» – порядок*): $O(f(x))$, где $f(x)$ – формула, выражающая сложность алгоритма. В формуле может присутствовать переменная n , представляющая размер входных данных. Ниже приводится список наиболее часто встречающихся порядков роста, но он ни в коем случае не полный.

Константный – $O(1)$

Порядок роста $O(1)$ означает, что вычислительная сложность алгоритма не зависит от размера входных данных. Следует помнить, однако, что единица в формуле не значит, что алгоритм выполняется за одну операцию или требует очень мало времени. Он может потребовать и микросекунду, и год. Важно то, что это время не зависит от входных данных.

```
public int GetCount(int[] items)
{
    return items.Length;
}
```

Линейный – $O(n)$

Порядок роста $O(n)$ означает, что сложность алгоритма линейно растет с увеличением входного массива. Если линейный алгоритм обрабатывает один элемент пять миллисекунд, то мы можем ожидать, что тысячу элементов он обработает за пять секунд.

Такие алгоритмы легко узнать по наличию цикла по каждому элементу входного массива.

```
public long GetSum(int[] items)
{
    long sum = 0;
    foreach (int i in items)
    {
        sum += i;
    }

    return sum;
}
```

Логарифмический – $O(\log n)$

Порядок роста $O(\log n)$ означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива. (Прим. пер.: в анализе алгоритмов по умолчанию используется логарифм по основанию 2). Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность. Метод Contains бинарного дерева поиска (*binary search tree*) также имеет порядок роста $O(\log n)$.

Линеарифметический – $O(n \cdot \log n)$

Линеарифметический (или линейно-логарифмический) алгоритм имеет порядок роста $O(n \log n)$. Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию. В следующих частях мы увидим два таких примера – сортировка слиянием и быстрая сортировка.

Квадратичный – $O(n^2)$

Время работы алгоритма с порядком роста $O(n^2)$ зависит от квадрата размера входного массива. Несмотря на то, что такой ситуации иногда не избежать, квадратичная сложность – повод пересмотреть используемые алгоритмы или структуры данных. Проблема в том, что они плохо масштабируются. Например, если массив из тысячи элементов потребует 1 000 000 операций, массив из миллиона элементов потребует 1 000 000 000 000 операций. Если одна операция требует миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов 32 года. Даже если он будет в сто раз быстрее, работа займет 84 дня.

Так, например, квадратичной сложностью обладает алгоритм пузырьковой сортировки.

Обычно, при оценке сложности алгоритма указывают наилучший, средний и наихудший случай

Что мы имеем в виду, когда говорим, что порядок роста сложности алгоритма – $O(n)$? Это усредненный случай? Или наихудший? А может быть, наилучший?

Обычно имеется в виду наихудший случай, за исключением тех случаев, когда наихудший и средний сильно отличаются. К примеру, мы увидим примеры алгоритмов, которые в среднем имеют порядок роста $O(1)$, но периодически могут становиться $O(n)$ (например, `ArrayList.add`). В этом случае мы будем указывать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает.

Самое важное здесь то, что $O(n)$ означает, что алгоритм потребует **не более n шагов**.

Говоря об оценке сложности алгоритма, нельзя не упомянуть алгоритмически неразрешимые проблемы.

Алгоритмически неразрешимые проблемы

Основным стимулом, приведшим к выработке понятия алгоритма и созданию теории алгоритмов, явилась *потребность доказательства неразрешимости* многих проблем, возникших в различных областях математики.

Специалист, решая задачу, всегда должен считаться с возможностью того, что она может оказаться неразрешимой.

При доказательстве неразрешимости той или иной проблемы часто используется так называемый *метод сводимости*, заключающийся в следующем.

Пусть, например, в результате некоторых рассуждений удалось показать, что решение проблемы *Pr1* приводит к решению другой проблемы *Pr2*. В этом случае говорят, что проблема *Pr2* *сводится* к проблеме *Pr1*. Таким образом, если проблема *Pr2* сводится к проблеме *Pr1*, то из разрешимости *Pr1* следует разрешимость *Pr2* и, наоборот, из неразрешимости *Pr2* следует неразрешимость *Pr1*.

Неразрешима задача установления истинности произвольной формулы исчисления предикатов (т.е. *исчисление предикатов неразрешимо*) – эта теорема была доказана в 1936 г. Черчом.

В 1946-47 гг. А.А. Марковым и Э. Постом независимо друг от друга доказали отсутствие алгоритма для распознавания эквивалентности слов в любом ассоциативном исчислении.

В теории алгоритмов к алгоритмически неразрешимой относится *«проблема остановки»*: можно ли по описанию алгоритма (*Q*) и входным данным (*x*) установить, завершится ли выполнение алгоритма результ ativной остановкой? Не существует алгоритма (машины Тьюринга), позволяющего по описанию произвольного алгоритма и его исходных данных (и алгоритм и данные заданы символами на ленте машины Тьюринга) определить, останавливается ли этот алгоритм на этих данных или работает бесконечно.

Одной из наиболее знаменитых алгоритмических проблем математики являлась 10-я проблема Гильберта, поставленная им в числе других в 1901 г. на Международном математическом конгрессе в Париже. Требовалось найти алгоритм, определяющий для любого *диафантона уравнения*, имеет ли оно целочисленное решение. Диафантово уравнение есть уравнение вида $F(x, y, \dots, z) = 0$, где $F(x, y, \dots, z)$ – многочлен с целыми показателями степеней и с целыми коэффициентами. В общем случае эта проблема долго оставалась нерешенной, и только в 1970 г. советский математик Ю.В. Матиясевич доказал ее неразрешимость.

Важность доказательства алгоритмической неразрешимости в том, что если такое доказательство получено, оно имеет *смысл*.

закона-запрета, позволяющего не тратить усилия на поиск решения, подобно тому, как законы сохранения в физике делают бессмысленными попытки построения вечного двигателя.

Вместе с этим необходимо сознавать, что алгоритмическая неразрешимость какой-либо задачи в общей постановке *не исключает возможности* того, что разрешимы какие-то ее частные случаи.

Приведем некоторые из них.

Проблема 1: Распределение девяток в записи числа π ;

Определим функцию $f(n) = i$, где n – количество девяток подряд в десятичной записи числа π , а i – номер самой левой девятки из n девяток подряд:

$$\pi=3,141592\dots \quad f(1)=5.$$

Задача состоит в вычислении функции $f(n)$ для произвольно заданного n .

Поскольку число π является иррациональным и трансцендентным, то мы не знаем никакой информации о распределении девяток (равно как и любых других цифр) в десятичной записи числа π . Вычисление $f(n)$ связано с вычислением последующих цифр в разложении π , до тех пор, пока мы не обнаружим n девяток подряд, однако у нас нет общего метода вычисления $f(n)$, поэтому для некоторых n вычисления могут продолжаться бесконечно – мы даже не знаем в принципе (по природе числа π) существует ли решение для всех n .

Проблема 2: Вычисление совершенных чисел;

Совершенные числа – это числа, которые равны сумме своих делителей, например: $28 = 1+2+4+7+14$.

Определим функцию $S(n) = n\text{-ое}$ по счёту совершенное число и поставим задачу вычисления $S(n)$ по произвольно заданному n . Нет общего метода вычисления совершенных чисел, мы даже не знаем, множество совершенных чисел конечно или счетно, поэтому наш алгоритм должен перебирать все числа подряд, проверяя их на совершенность. Отсутствие общего метода решения не позволяет ответить на вопрос о останове алгоритма. Если мы проверили M чисел при поиске n -ого совершенного числа – означает ли это, что его вообще не существует?

Проблема 3: Десятая проблема Гильберта;

Пусть задан многочлен n -ой степени с целыми коэффициентами – P , существует ли алгоритм, который определяет, имеет ли уравнение $P = 0$ решение в целых числах?

Ю.В. Матиясевич показал, что такого алгоритма не существует, т.е. отсутствует общий метод определения целых корней уравнения $P=0$ по его целочисленным коэффициентам.

б) Информационная неопределенность задачи

Проблема 4: Позиционирование машины Поста на последний помеченный ящик;

Пусть на ленте машины Поста заданы наборы помеченных ящиков (кортежи) произвольной длины с произвольными расстояниями между кортежами и головка находится у самого левого помеченного ящика. Задача состоит установке головки на самый правый помеченный ящик последнего кортежа.

Попытка построения алгоритма, решающего эту задачу приводит к необходимости ответа на вопрос – когда после обнаружения конца кортежа мы сдвинулись вправо по пустым ящикам на M позиций и не обнаружили начало следующего кортежа – больше на ленте кортежей нет или они есть где-то правее? Информационная неопределенность задачи состоит в отсутствии информации либо о количестве кортежей на ленте, либо о максимальном расстоянии между кортежами – при наличии такой информации (при разрешении информационной неопределенности) задача становится алгоритмически разрешимой.

в) Логическая неразрешимость (в смысле теоремы Гёделя о неполноте)

Проблема 5: Проблема «останова» (см. выше);

Проблема 6: Проблема эквивалентности алгоритмов;

По двум произвольным заданным алгоритмам (например, по двум машинам Тьюринга) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.

Проблема 7: Проблема тотальности;

По произвольному заданному алгоритму определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи – является ли частичный алгоритм P всюду определённым?

Примеры анализа алгоритмов¹

Пример 1. Рассмотрим задачу поиска наибольшего элемента в списке из n чисел. Для простоты предположим, что этот список реализован в виде массива. Ниже приведен псевдокод алгоритма решения этой задачи.

Алгоритм MaxElement (A [0.. $n - 1$])

```
// Входные данные: массив вещественных чисел A[0..n - 1]
// Выходные данные: возвращается значение наибольшего
// элемента массива A
begin
    maxval ← A[0]
    for i ← 1 to n - 1 do
        if A[i] > maxval
            then maxval ← A[i]
    return maxval
end
```

Очевидно, что в этом алгоритме размер входных данных нужно оценивать по количеству элементов в массиве, т.е. числом n . Операции, выполняемые чаще всего, находятся во внутреннем цикле `for` алгоритма. Таких операций две: сравнение ($A[i] > maxval$) и присваивание ($maxval ← A[i]$). Какую из них считать базовой? Поскольку сравнение выполняется на каждом шаге цикла, а присваивание – нет, логично считать, что основной операцией алгоритма является операция присваивания. (Обратите внимание, что для любого массива размером n , количество операций сравнения, в рассматриваемом алгоритме, постоянно. Поэтому не имеет смысла отдельно рассматривать эффективность алгоритма для худшего, типичного и лучшего случаев.)

Обозначим через $f(n)$ количество выполняемых в алгоритме операций сравнения и попытаемся вывести формулу, выражющую их зависимость от размера входных данных n . Известно, что за один цикл в алгоритме выполняется одна операция сравнения. Этот процесс повторяется для каждого значения счетчика цикла i , которое изменяется от 1 до $n - 1$ включительно. Поэтому для $f(n)$ получаем следующую сумму:

¹ См. <https://studopedia.org/4-12785.html>

$$f(n) = \sum_{i=1}^{n-1} 1$$

Значение этой суммы очень легко вычислить, поскольку в ней единица суммируется сама с собой $n - 1$ раз. Поэтому

$$f(n) = \sum_{i=1}^{n-1} 1 = n - 1, \quad \Theta(f(n)) = \Theta(n - 1) = n$$

Пример 2. Рассмотрим задачу проверки единственности элементов (т.е. есть необходимо попарно сравнить элементы и убедиться, что все элементы массива различны).

АЛГОРИТМ UniqueElements (A [0.. n – 1])

```
// Входные данные: массив вещественных чисел A[0.. n–1]
// Выходные данные: возвращается значение «true», если все
// элементы массива A различны, и «false»— в противном случае
begin
    for i <- 0 to n – 2 do
        for j <- i + 1 to n – 1 do if A[i] = A[j]
            then return false // и досрочное завершение цикла
    return true
end
```

В этом алгоритме, как и в примере 1, размер входных данных вполне естественно оценивать по количеству элементов в массиве, т.е. числом n . Поскольку в наиболее глубоко вложенном внутреннем цикле алгоритма выполняются только одна операция сравнения двух элементов, ее и будем считать основной операцией этого алгоритма. Обратите внимание, что количество операций сравнения будет зависеть не только от общего числа n элементов в массиве, но и от того, есть ли в массиве одинаковые элементы, и если есть, то на каких позициях они расположены. Ограничимся рассмотрением наихудшего случая.

По определению наихудший случай входных данных соответствует такому массиву элементов, при обработке которого количество операций сравнений $f(n)$ будет максимальным среди всей совокупности входных массивов размером n . После анализа внутреннего цикла алгоритма приходим к выводу, что наихудший случай входных данных (т.е. когда цикл выполняется от начала до

конца, а не завершается досрочно) может возникнуть при обработке массивов двух типов:

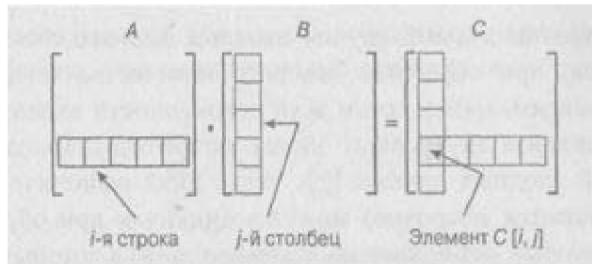
- а) в которых нет одинаковых элементов;
- б) в которых два одинаковых элемента находятся рядом и расположены в самом конце массива.

В подобных случаях при каждом повторе внутреннего цикла в нашем алгоритме выполняется одна операция сравнения. При этом переменная цикла j последовательно принимает значения от $i + 1$ до $n - 1$. Внутренний цикл каждый раз повторяется заново при каждом выполнении внешнего цикла. При этом переменная внешнего цикла i последовательно принимает значения от 0 до $n - 2$. Таким образом, получаем:

$$\begin{aligned}
 f(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
 &= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] \\
 &= \sum_{i=0}^{n-2} (n-1-i) = \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\
 &= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
 &= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} = \frac{1}{2}(n^2 - n) \\
 O(f(n)) &= O\left(\frac{1}{2}(n^2 - n)\right) = O(n^2 - n) = \max\{n^2, n\} = n^2
 \end{aligned}$$

Обратите внимание, что этот результат можно было легко предсказать: в рассматриваемом алгоритме в самом худшем случае для массива, состоящего из n элементов, нужно сравнить между собой все $(n-1)n/2$ различных пар элементов.

Пример 3. Для двух заданных матриц А и В размером $n \times n$ определите временную эффективность алгоритма их умножения $C = AB$, основанного на определении этой операции. По определению C – это матрица размером $n \times n$, элементы которой являются скалярными произведениями соответствующих строки матрицы А и столбца матрицы В, как показано ниже.



АЛГОРИТМ MatrixMultiplication (A [0..n - 1, 0..n - 1],
 B [0..n - 1, 0..n - 1])

```
// Выполняется умножение двух квадратных матриц
// размером  $n \times n$ . Используется алгоритм,
// основанный на определении этой операции
// Входные данные: две квадратные  $n \times n$  матрицы A к B
// Выходные данные: матрица C = AB
```

BEGIN

```
FOR i ← 0 TO n-1 DO
    FOR j ← 0 TO n-1 DO
        C[i,j] ← 0.0
        FOR k ← 0 TO n-1 DO
            C[i,j] ← C[i,j]+A[i,k]*B[k,j]
    RETURN C
END
```

В этом алгоритме размер входных данных соответствует размеру матрицы n . Во внутреннем цикле алгоритма выполняются две арифметические операции: умножение и сложение. В принципе, в качестве основной операции алгоритма можно выбрать как одну, так и другую операцию. Мы рассмотрим случай, когда в качестве основной выбрана операция умножения. Заметьте, что для рассматриваемого алгоритма не обязательно отдавать предпочтение одной из этих операций, поскольку на каждом шаге внутреннего цикла каждая из них выполняется только один раз. Поэтому, подсчитав, сколько раз выполняется одна из операций, мы автоматически подсчитываем и количество выполнения другой операции. А теперь давайте составим сумму для общего числа операций умножения $f(n)$, выполняемых в алгоритме. Поскольку

это число зависит только от размера исходных матриц, не требуется отдельно рассматривать наихудший, типичный и наилучший случаи.

Вполне очевидно, что на каждом шаге внутреннего цикла алгоритма выполняется только одна операция умножения. При этом значение переменной цикла k последовательно изменяется от нижней границы 0 до верхней границы $n - 1$. Поэтому количество операций умножения, выполняемых для каждой пары значений переменных i и j , можно записать следующей тройной суммой:

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Для вычисления значения этой суммы воспользуемся формулами из следующего раздела:

$$f(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3$$

Рассматриваемый алгоритм достаточно прост.

Время выполнения алгоритма на конкретном компьютере можно оценить с помощью следующего произведения:

$$T(n) \approx c_m f(n) = c_m n^3$$

c_m – время выполнения одной команды умножения на рассматриваемом компьютере. Чтобы получить более точную оценку, необходимо также учесть время выполнения команд сложения:

$$T(n) \approx c_m f(n) + c_a f_a(n) = c_m n^3 + c_a n^3 = (c_m + c_a)n^3$$

где $c_a \sim$ время выполнения одной команды сложения. Обратите внимание, что полученная оценка отличается от прежней только постоянным множителем, а не порядком роста.

Заключение

В заключение заметим, что процесс разработки алгоритма и выбор оптимальных, с точки зрения решаемой задачи, структур данных – процесс творческий и весьма трудоемкий, но успешное выполнение – гарантия качественного решения вашей задачи на компьютере.

Вопросы для самопроверки

1. Введите понятие алгоритма.
2. Какими свойствами обладает алгоритм.
3. Сформулируйте основные свойства (характеристики) алгоритмов.
4. Какие способы представления алгоритмов вы знаете.
5. Что такое псевдокод.
6. Объясните свойство «массовость» в теории алгоритмов.
7. Понятие сложностных классов задач, класс P;
8. Сложностной класс NP, понятие сертификата;
9. Проблема $P=NP$, и ее современное состояние
10. Укажите направление развития теории алгоритмов.
11. Понятие индукции и рекурсии;
12. Примеры рекурсивного задания функций;
13. Рекурсивная реализация алгоритмов
14. Трудоемкость механизма вызова функции в языке высокого уровня;
15. Рекурсивное дерево, рекурсивные вызовы и возвраты;
16. Анализ трудоемкости рекурсивного алгоритма вычисления факториала;
17. Формальное описание машины Тьюринга;
18. Функция переходов в машине Тьюринга;
19. Как работает универсальная машина Тьюринга?
20. Понятие об алгоритмически неразрешимых проблемах
21. Проблема позиционирования в машине Поста;
22. Проблема соответствий Поста над алфавитом S;
23. Проблема останова в машине Тьюринга;
24. Проблема эквивалентности и тотальности;

25. Какие алгоритмически неразрешимые проблемы вы знаете?

Литература

1. Ахо А., Хопкрофт Дж., Ульман Дж. Структуры данных и алгоритмы. – М.: «Вильямс», 2001. – 384 с.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. – М.: Мир, 1979.
3. Бентли Д. Жемчужины творчества программистов. – М.: Радио и связь, 1990.
4. Бхаргава А. Грекаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб.: Питер, 2017. – 288 с.
5. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985.
6. Вирт Н. Алгоритмы и структуры данных. – М: Мир, 1989. – 360 с.
7. Грин Д., Кнут Д. Математические методы анализа алгоритмов. – М: Мир, 1987.
8. Гудман С., Хидентиemi С. Введение в разработку и анализ алгоритмов. – М.: Мир, 1981. 368 с.
9. Дейкстра Э. Дисциплина программирования. – М: Мир, 1978.
10. Кнут Д.Е. Искусство программирования для ЭВМ. В 3-х томах. – М.: Мир, 1976.
11. Кнут Д.Е. Искусство программирования. В 3-х томах. – М.: «Вильямс», 2000.
12. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: Построение и анализ. – М.: МЦНМО, 2005.
13. Литл Дж., Мурти К., Суни Д., Кэрел К. Алгоритм для решения задачи о коммивояжере// Экономика и математические методы. – 1965. Т. 1. Вып. 1. С. 94–107.
14. Лэгсам Й., Огенстайн М. Структуры данных для персональных ЭВМ. – М.: Мир, 1989. – 586 с.
15. Оре О. Графы и их применение. – М.: Мир, 1965.
16. Пойа. Как решать задачу. –
17. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. – М.: Мир, 1980.

18. Сибуя М., Ямамото Т. Алгоритмы обработки данных. – М: Мир, 1986. – 218 с.
19. Успенский В.А., Семенов А.Л. Теория алгоритмов: основные открытия и приложения. – М.: Наука, 1987.
20. Харари Ф. Теория графов. – М: Мир, 1973.
21. Левитин А.В. Алгоритмы: введение в разработку и анализ: Пер. с англ. – М.: Издательский дом «Вильямс», 2006. – 576.
22. В.Д. Далека, А.С. Деревянко, О.Г. Кравец, Л.Е. Тимановская Модели и структуры данных. Учебное пособие. Харьков: ХТПУ, 2000. – 241 с.

Русскоязычные ресурсы Интернет

- 1) <http://algo.4u.ru/>
- 2) <http://algolist.manual.ru/>
- 3) <http://alglib.chat.ru/>
- 4) <http://algo.do.ru/>
- 5) <http://hcinsu.chat.ru/>
- 6) <http://algolist.da.ru/>
- 7) <http://progstone.narod.ru/links/wantalgo.html>
- 8) <http://www.sevmashvtuz.edu/links/algorithms.html>

Приложения

Приложение 1

Пример проектирования алгоритмов

В качестве примера разработки алгоритма рассмотрим создание программы для решения класса задач на полное квадратное уравнение.

Пример спецификации осмысленной задачи

Рассмотрим следующую задачу.

Описание задачи.

Капля дождя зарядом Q падает на некоторую горизонтальную поверхность и разбивается на две части, которые откатываются друг от друга на расстояние l и взаимодействуют с силой F . Требуется определить заряды вновь образовавшихся капель (рис. 62).

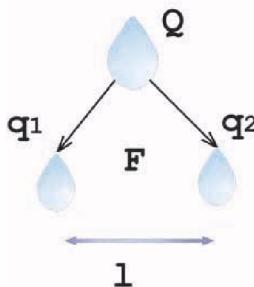


Рис. 62. Задача о капле дождя

Приступая к точной математической постановке описанной выше задачи необходимо, вначале, ее понять, а, следовательно, проанализировать и выяснить те предметные области, знания которых помогут нам при ее постановке.

Иначе говоря, на данном этапе происходит *осмысление* задачи.

Анализируя описание нашей задачи, мы можем отметить следующее:

1) Первый этап – анализ и исследование задачи (прагматика).

Поскольку речь идет о зарядах, то необходимо внимательно приглядеться к разделу физики, в котором они рассматриваются. Для нашей задачи интерес представляет закон Кулона, который связывает все рассматриваемые нами понятия: заряды капель, расстояние между каплями и силу взаимодействия зарядов и записывается следующим образом

$$F = \gamma \frac{q_1^* q_2}{l^2} \quad (1)$$

где q_1 и q_2 – заряды вновь образовавшихся капель;

F – сила, с которой взаимодействуют вновь образовавшиеся капли;

l – расстояние между вновь образовавшимися каплями;

γ – константа.

2) Второй этап – фиксация и/или определение вводимых понятий (методология).

Анализируя задачу, мы использовали такие понятия, как: *заряд, сила, расстояние, константа*. Все эти понятия мы будем трактовать так, как их трактуют в физике и математике. Фиксация этих понятий (*методология*) позволяет нам – говорить точно о неточном.

3) Третий этап – точная математическая постановка задачи (теория)

На данном этапе мы абстрагируемся от реальности и строим математическую модель той задачи (или класса задач), которую собираемся решать.

Исходная капля дождя имела заряд Q и согласно законов физики и математики мы можем записать еще одно равенство:

$$Q = q_1 + q_2 \quad (2)$$

Теперь строим математическую модель задачи (или класса задач), которую собираемся решать. Для точной постановки задачи пользуемся алгеброй. Здесь A – множество вещественных чисел; с сигнатурой $\{+, -, x, :, **, \sqrt{\cdot}\}$

В нашем примере это выглядит следующим образом.

Все рассматриваемые понятия (заряды, сила, расстояние и константа) будут рассматриваться во множестве действительных чисел.

Из закона Кулона следует, что R не может равняться нулю ($l \neq 0$).

Для того чтобы найти заряды капель, мы должны рассмотреть систему уравнений (3)

$$\begin{cases} F = \gamma \frac{q_1 * q_2}{l^2} \\ Q = q_1 + q_2 \end{cases} \quad (3)$$

Применяя теорему Виета, мы можем упростить нашу модель, сведя её к приведенному квадратному уравнению

$$x^2 + px + q = 0 \quad (4)$$

где $p = -Q$

$$q = \frac{F * l^2}{\gamma}$$

или полагая $a = \gamma$, $b = -Qx$, $c = F * l^2$ получаем полное квадратное уравнение

$$ax^2 + bx + c + 0 \quad (5)$$

Точная постановка осмысленной задачи

Все расчеты будут вестись во множестве вещественных чисел. Расстояние между вновь образовавшимися каплями дождя не может быть равно нулю

$$(R \neq 0).$$

Из курса математики мы знаем, что полное квадратное уравнение имеет решение во множестве вещественных чисел (а именно с ними мы только и имеем право работать в нашем примере) только тогда, когда его дискриминант неотрицателен, т. е.

$$D = b^2 - 4ac \geq 0 \quad (6).$$

Сами же корни уравнения (5) находятся по следующему алгоритму

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (7)$$

Пункты (6) и (7) обеспечивают нам *критерий осмыслинности* нашей задачи.

Окончательно: *точная постановка осмыслинной задачи* для нашего примера будет следующей.

Дано полное квадратное уравнение
 $ax^2 + bx + c = 0$.

Найти его решение во множестве вещественных чисел.

Четвертый этап – спецификация задачи для ее решения на компьютере.

Перед тем, как мы приступим к разработке программного обеспечения, для решения задачи на компьютере, мы должны выполнить следующие действия:

- 1) выбрать метод реализации алгоритма решения задачи;
- 2) описать структуру входных и выходных данных задачи;
- 3) зафиксировать все особые условия решения данной задачи.

Для нашего примера это будет выглядеть следующим образом:

- 1) Поскольку самым критичным местом в данном алгоритме является вычисление корня квадратного из дискриминанта, то с него и начинается проектирование алгоритма. Метод проектирования – водопадный (описан во второй части пособия).
- 2) Как было выяснено на третьем этапе – все расчеты ведутся во множестве вещественных чисел. Значит тип входных и выходных данных – вещественные числа. Способ их пред-

ставления в компьютере (фиксированная или плавающая запятая) определяется заказчиком.

- 3) К особым условиям относится проверка на ноль введенного расстояния между вновь образовавшимися каплями ($l > 0$).

Таким образом, спецификация задачи включает: выбор или разработку алгоритма, определение типов входных и выходных данных, фиксацию ограничений и т. п.

Дано: заряд исходной капли Q , рассматривается на множестве вещественных чисел;
сила взаимодействия вновь образовавшихся капель равна F , также определена на множестве вещественных чисел;
расстояние между вновь образовавшимися каплями равно l ($l > 0$) и тоже определено на множестве вещественных чисел.

Найти: заряды q_1 и q_2 вновь образовавшихся капель.

Задача будет иметь решение на множестве вещественных чисел и расчет будет вестись по формуле (7), если дискриминант (6) будет неотрицателен.

Это и есть **критерий осмыслинности** задачи.

Приложение 2

Системы счисления

Все фантастические возможности вычислительной техники (ВТ) реализуются путем создания разнообразных комбинаций сигналов высокого и низкого уровней, которые условились называть «единицами» и «нулями».

Система счисления (СС) – это система записи чисел с помощью определенного набора цифр. СС называется *позиционной*, если одна и та же цифра имеет различное значение, которое определяется ее местом в числе. Десятичная СС является позиционной: 999. Римская СС является *непозиционной*. Значение цифры X в числе XXI остается неизменным при вариации ее положения в числе. Количество различных цифр, употребляемых в позиционной СС, называется *основанием СС*.

Развернутая форма числа – это запись, которая представляют собой сумму произведений цифр числа на значение позиций.

Например: $8527 = 8 \cdot 10^3 + 5 \cdot 10^2 + 2 \cdot 10^1 + 7 \cdot 10^0$

Развернутая форма записи чисел произвольной системы счисления имеет вид

$$X = \sum_{i=-n}^{-m} a_i q^i$$

где

X – число;

a – основа системы счисления;

i – индекс;

m – количество разрядов числа дробной части;

n – количество разрядов числа целой части.

Например: 327.46 n=3, m=2, q=10

$$\begin{aligned} X = \sum_{i=-2}^{-2} a_i q^i &= a_2 \cdot 10^{-2} + a_1 \cdot 10^{-1} + a_{-2} \cdot 10^0 + a_{-1} \cdot 10^{-1} + a_{-2} \cdot 10^{-2} = \\ &= 3 \cdot 10^{-2} + 2 \cdot 10^{-1} + 7 \cdot 10^0 + 4 \cdot 10^{-1} + 6 \cdot 10^{-2} \end{aligned}$$

Если основание используемой СС больше десяти, то для цифр вводят условное обозначение со скобкой вверху или буквенное обозначение.

Например: если $10=A$, а $11=B$, то число $7A.5B_{12}$ можно расписать так:

$$7A.5B_{12} = B \cdot 12^{-2} + 5 \cdot 2^{-1} + A \cdot 12^0 + 7 \cdot 12^1.$$

В шестнадцатеричной СС основа – это цифры 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 с соответствующими обозначениями 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Примеры чисел: 17D.ECH, F12AH.

Двоичная СС – это система, в которой для записи чисел используются две цифры 0 и 1. Основанием двоичной системы счисления является число 2.

Двоичный код числа – запись этого числа в двоичной системе счисления. Например,

$$\begin{aligned} 0 &= 0_2 \\ 1 &= 1_2 \\ 2 &= 10_2 \\ 3 &= 11_2 \dots \\ 7 &= 111_2 \\ 120 &= 1111000_2. \end{aligned}$$

В ВТ применяют позиционные СС с недесятичным основанием: двоичную, восьмеричную, шестнадцатеричную. Для обозначения используемой СС число снабжают верхним или нижним индексом, в котором записывают основание СС. Другой способ – использование латинских букв после записи числа:

- D – десятичная СС
- B – двоичная СС
- O – восьмеричная СС
- H – 16-ричная СС.

Несмотря на то, что 10-тичная СС имеет широкое распространение, цифровые ЭВМ строятся на двоичных элементах, т.к. реализовать элементы с 10 четко различимыми состояниями сложно. Историческое развитие ВТ сложилось таким образом, что ЭВМ

строются на базе двоичных цифровых устройств: триггеров, регистров, счетчиков, логических элементов и т.д.

16-ричная и 8-ричная СС используются при составлении программ на языке машинных кодов для более короткой и удобной записи двоичных кодов – команд, данных, адресов и операндов.

Задача перевода из одной СС в другую часто встречается при программировании, особенно, на языке Ассемблера. Например, при определении адреса ячейки памяти. Отдельные стандартные процедуры языков программирования Паскаль, Бейсик, Си, HTML требуют задания параметров в 16-ричной СС. Для непосредственного редактирования данных, записанных на жесткий диск, также необходимо умение работать с 16-ричными числами. Отыскать неисправность в ЭВМ невозможно без представлений о двоичной СС.

В таблице 13 приведены некоторые числа, представленные в различных СС.

Таблица 13.

Двоичные числа	Восьмеричные числа	Десятичные числа	Шестнадцатеричные числа
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Перевод чисел из произвольной СС в десятичную и обратно

Перевод чисел из произвольной системы в десятичную. Для перевода числа из любой позиционной СС в десятичную необходимо использовать развернутую форму числа, заменяя, если это необходимо, буквенные обозначения соответствующими цифрами. Например:

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$$
$$17D.ECH = 12 \cdot 16^2 + 14 \cdot 16^1 + 13 \cdot 16^0 + 7 \cdot 16^{-1} + 1 \cdot 16^{-2} = 381.921875$$

Перевод чисел из десятичной СС в заданную.

1) Для преобразования целых чисел десятичной системы счисления в число любой системы счисления последовательно выполняют деление нацело на основание СС, пока не получат нуль. Числа, которые возникают как остаток от деления на основание СС, представляют собой последовательную запись разрядов числа в выбранной СС от младшего разряда к старшему. Поэтому для записи самого числа остатки от деления записывают в обратном порядке.

Например:

$$\begin{array}{r} 475|2 \\ 1 \quad 237|2 \\ 1 \quad 118|2 \\ 0 \quad 59|2 \\ 1 \quad 29|2 \\ 1 \quad 14|2 \\ 0 \quad 7|2 \\ 1 \quad 3|2 \\ 1 \quad 1|2 \\ 1 \quad 0 \end{array}$$

Читая остатки от деления снизу вверх, получим 111011011.

Проверка:

$$1 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$
$$1 + 2 + 8 + 16 + 64 + 128 + 256 = 475_{10}.$$

2) Для преобразования десятичных дробей десятичной СС в число любой СС последовательно выполняют умножение на основание системы счисления, пока дробная часть произведения не станет равной нулю. Полученные целые части являются разряда-

ми числа в новой системе, и их необходимо представлять цифрами этой новой системы счисления. Целые части в дальнейшем отбрасываются.

Например: перевести число 0.375_{10} в двоичную СС.

$$\begin{array}{r} 0.375 \quad \rightarrow 0 \\ \underline{\quad 2} \\ 0.750 \quad \rightarrow 0 \\ \underline{\quad 2} \\ 1.50 \quad \rightarrow 1 \\ \underline{\quad 2} \\ 0.50 \\ \underline{\quad 2} \\ 1.00 \quad \rightarrow 1 \end{array}$$

Полученный результат – 0.011_2 .

Необходимо отметить, что не каждое число может быть точно выражено в новой системе счисления, поэтому иногда вычисляют только требуемое количество разрядов дробной части, округляя последний разряд.

Перевод между основаниями, составляющими степень 2

Для того, чтобы из восьмеричной системы счисления перевести число в двоичный код, необходимо каждую цифру этого числа представить триадой двоичных символов. Лишние нули в старших разрядах отбрасываются.

Например:

$$1234.777_8 = 001\ 010\ 011\ 100.111\ 111\ 111_2 = 1\ 010\ 011\ 100.111\ 111\ 111_2$$

$$1234567_8 = 001\ 010\ 011\ 100\ 101\ 110\ 111_2 = 1\ 010\ 011\ 100\ 101\ 110\ 111_2$$

Обратный перевод: каждая триада двоичных цифр заменяется восьмеричной цифрой, при этом, если необходимо, число выравнивается путем дописывания нулей перед целой частью или после дробной.

Например:

$$1100111_2 = 001\ 100\ 111_2 = 147_8$$

$$11.1001_2 = 011.100\ 100_2 = 3.448$$

$$110.0111_2 = 110.011\ 100_2 = 6.348$$

При переводах между двоичной и шестнадцатеричной СС используются четверки цифр. При необходимости выравнивание выполняется до длины двоичного числа, кратной четырем.

Например:

$$1234.AB77_{16} = 0001\ 0010\ 0011\ 0100.1010\ 1011\ 0111\ 0111_2 = 1\ 0010\ 0011\ 0100.1010\ 1011\ 0111\ 0111_2$$

$$CE4567_{16} = 1100\ 1110\ 0100\ 0101\ 0110\ 0111_2$$

$$0.1234AA_{16} = 0.0001\ 0010\ 0011\ 0100\ 1010\ 1010_2$$

$$1100111_2 = 0110\ 0111_2 = 67_{16}$$

$$11.1001_2 = 0011.1001_2 = 3.9_{16}$$

$$110.0111001_2 = 0110.0111\ 0010_2 = 65.72_{16}$$

При переходе из *восьмеричного счисления в шестнадцатеричное счисление* и обратно используется вспомогательный двоичный код числа.

Например:

$$1234567_8 = 001\ 010\ 011\ 100\ 101\ 110\ 111_2 = 0101\ 0011\ 1001\ 0111\ 0111_2 \\ = 53977_{16}$$

$$0.12034_8 = 0.001\ 010\ 000\ 011\ 100_2 = 0.0010\ 1000\ 0011\ 1000_2 = 0.2838_{16}$$

$$120.34_8 = 001\ 010\ 000.\ 011\ 100_2 = 0101\ 0000.0111\ 0000_2 = 50.7_{16}$$

$$1234.AB77_{16} = 0001\ 0010\ 0011\ 0100.1010\ 1011\ 0111\ 0111_2 =$$

$$= 001\ 001\ 000\ 110\ 100.101\ 010\ 110\ 111\ 011\ 100_2 = 11064.526734_8$$

$$CE4567_{16} = 1100\ 1110\ 0100\ 0101\ 0110\ 0111_2 = 110\ 011\ 100\ 100\ 010\ 101\ 100\ 111_2 = 63442547_8$$

$$0.1234AA_{16} = 0.0001\ 0010\ 0011\ 0100\ 1010\ 1010_2 = 0.000\ 100\ 100\ 011\ 010\ 010\ 010_2 = 0.04432252_8$$

Приложение 3

Свойства логарифмов

В приведенных ниже формулах основание алгоритмов считается большей 1; его конкретная величина значения не имеет. Запись

$\lg a$ означает логарифм по основанию 10;

$\ln x$ – логарифм по основанию $e = 2.718281828\dots$;

x и y – произвольные положительные числа.

1. $\log x^y = y \log x$
2. $\log xy = \log x + \log y$
3. $\log_{\frac{x}{y}} = \log x - \log y$
4. $\log_a x = \log_a b \log_b x$
5. $a^{\log_b x} = x^{\log_b a}$

Правила работы с суммами

1. $\sum_{i=l}^u ca_i = c \sum_{i=l}^u a_i$
2. $\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i$
3. $\sum_{i=l}^u (ca_i + b_i) = c \sum_{i=l}^u a_i + \sum_{i=l}^u b_i$
4. $\sum_{i=l}^u a_i = \sum_{i=l}^m a_i + \sum_{i=m+1}^u a_i$, где $l \leq m < u$
5. $\sum_{i=l}^u (a_i - a_{i-1}) = a_u - a_{l-1}$

Приложение 4

Языки программирования

Языки программирования – это средство записи алгоритмов для вычислителей алгоритмов (чаще всего компьютеров). С их помощью можно выполнять разнообразные по видам алгоритмы: делать математические вычисления, обрабатывать текстовые данные, изменять графику и др. В каком-то смысле компьютер может делать многое из того, что делает человек, а некоторые вещи намного быстрее. Однако, человек и компьютер «разговаривают» на совершенно разных языках. Человек – на естественном (русском, английском и др.), а компьютер – на формальном (машинном) языке.

Разработав алгоритм, человек должен как-то «объяснить» его компьютеру. Для этих целей служат языки программирования и трансляторы, а результатом записи алгоритма на них является программа. В настоящее время язык программирования – это скорее некий посредник между человеком и вычислительной машиной.

Программа – это алгоритм, закодированный на языке программирования. В дальнейшем переводится на машинный язык специальной программой-транслятором, но это уже другой материал, который в данном издании не рассматривается.

Москвитин А.А.

СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

Подписано в печать 15.07.2021 г.

Формат 60x84/16. Усл. печ. л. 8,37

Тираж 60 экз. Заказ № 328

Отпечатано в типографии

ООО «Рекламно-информационное агентство на КМВ»

Пятигорск, ул. Козлова, 19

Тел 8(879 3) 39-09-03, 33-36-56 (факс)