Информатика

Специальность
«Математическое обеспечение и администрирование информационных систем»
1 курс

68 часов лекций
40 часов лабораторных работ
экзамен

Литература:

- Глушков В.Л. Основы безбумажной информатики. М.: Наука, 1987.
- Бауэр Ф. Л., Гооз Г. Информатика (в 2 книгах). М.: Мир, 1990.
- Брой М. (Манфред) Информатика (в 4 книгах). М.: Диалог - МИФИ, 1996.
- Симонович С.В. Информатика. Базовый курс. Питер, 2000
- Острейковский В.А. Информатика. М, Высшая школа, 2000.
- Королев Л.Н., Миков А.И. Информатика. Введение в компьютерные науки. М.: Высшая школа, 2003.
- Стариченко Б.Е. Теоретические основы информатики. М.: Горячая линия - Телеком, 2003.
- Степанов А.Н. Информатика. 6 издание, СПб.: Питер, 2010
- Степанов А.Н. Курс информатики для студентов информационно-математических специальностей, СПб.: Питер, 2018

Тема 1. Роль информации в живой природе, жизни человека и общественном развитии.

Обмен (прием или передача), обработка и хранение информации — одна из самых важных сторон жизнедеятельности любых живых организмов

- ***Обмен*** информацией — процесс передачи информации одним объектом (субъектом) и ее прием другим объектом (субъектом). Под **обменом информацией** понимается ее *прием* или *передача* в тех случаях, когда безразлично, о чем именно идет речь.
- ***Обработка*** информации — упорядоченный процесс ее целесообразного преобразования.
- ***Хранение*** информации — поддержание информации в таком виде, который обеспечивает ее выдачу в нужном виде и в нужное время.

Носитель информации — любая материальная среда, служащая для ее хранения или передачи

Человек на протяжении всей своей жизни постоянно, ежечасно, ежеминутно сталкивается с необходимостью принимать, передавать, обрабатывать и хранить информацию.

Тема 2. Основные этапы развития информационных технологий

Технологией называется совокупность знаний о способах и средствах проведения производственных процессов, при которых происходит необходимое качественное изменение обрабатываемых объектов (techno – мастерство, log – учение, то есть учение о мастерстве, мастерство – искусство делать вещи) .

- ***Начальное состояние*** — информация хранится и обрабатывается в мозгу человека, обмен с помощью органов чувств, нечленораздельные звуки, телодвижения.

- ***Появление речи*** — самого совершенного в живой природе способа ***обмена*** информацией (1 000 000 лет назад).

- ***Появление письменности*** — способа ***долговременного хранения*** информации (30 000 лет назад).

- ***Изобретение книгопечатания*** — способа ***тиражирования*** информации (середина XV века).

Печатный станок Гуттенберга

- ***Развитие средств механизации и автоматизации обработки информации*** (с начала XVI века)***.***

❖1500 г., Леонардо да Винчи, эскиз суммирующего устройства

❖1623 г., Вильгельм Шиккард, действующее суммирующее устройство

❖1641–1645 г.г., Блез Паскаль, суммирующая машина

❖1671–1674 г.г., Готфрид Лейбниц, арифмометр

Арифмометр Лейбница

- ***Развитие средств механизации и автоматизации обработки информации*** (с начала XVI века)***.***

Арифмометр В.Т. Однера XIX века и его потомок середины XX века арифмометр «Феликс М»

- ***Развитие средств механизации и автоматизации обработки информации*** (с начала XVI века)***.***

❖1801–1808г.г., Жозеф Жаккард, автоматический ткацкий станок

❖1822 г. Чарльз Бэббидж, описание «разностной» машины

❖1834 г., Чарльз Бэббидж, эскиз «аналитической» машины

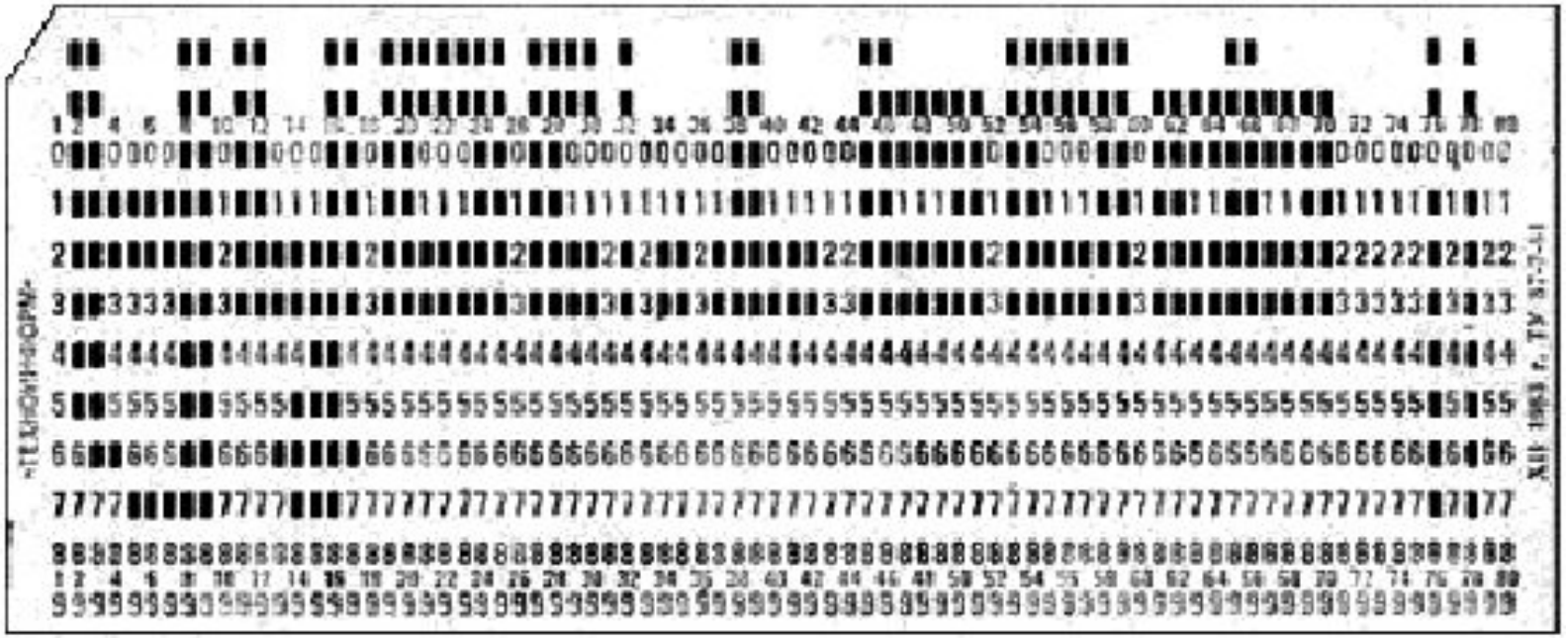
❖1843 г., Ада Лавлейс, основы программирования, первая в мире программа для аналитической машины Беббиджа (расчет чисел Фиббоначи)

❖1887 г., Герман Холлерит, первый табулятор

❖1897 г., Герман Холлерит, основание фирмы Tabulating Machine Company, впоследствии **IBM** (International Business Machines)

Программа представляет собой план выполнения действий, записанный в специальной, понятной исполнителю действий форме.

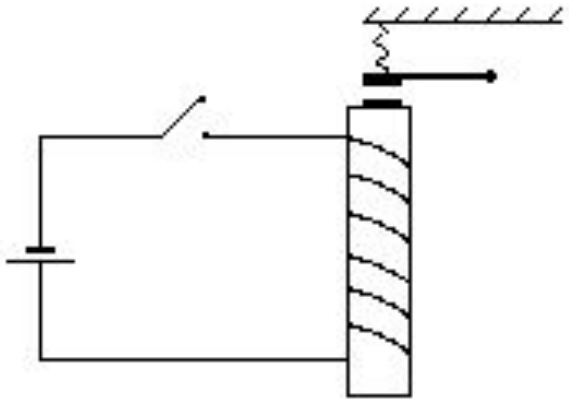
- ***Развитие средств механизации и автоматизации обработки информации*** (с начала XVI века)***.***



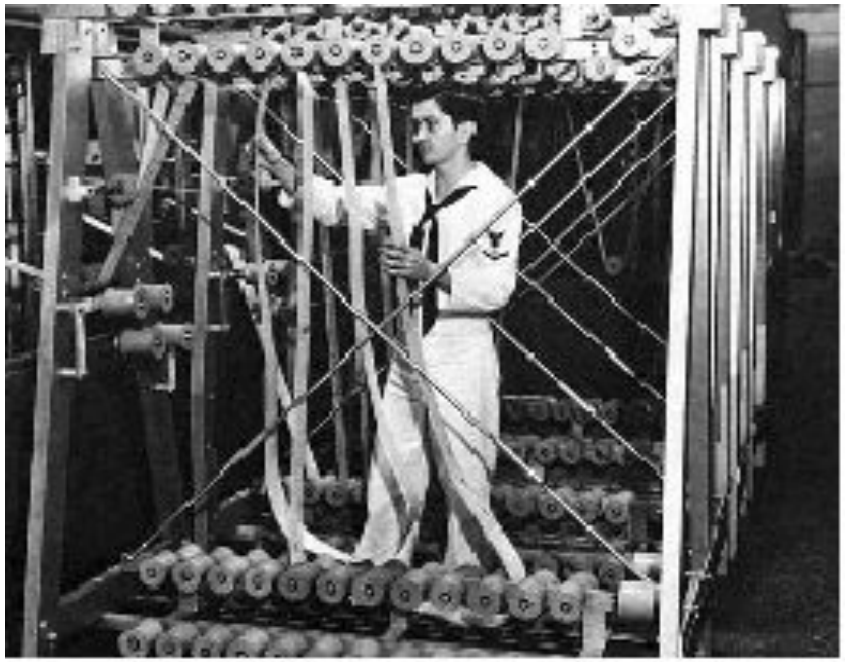
Перфокарта и перфолента середины XX века, прототипы использовались с начала XIX века

 Электромеханические машины (конец XIX века).

- 1939–1941 г.г., Конрад Цузе, Германия, машина «Z-3», память — 64 числа, сложение 0,3 секунды, умножение 5 секунд.
- 1937–1944 г.г., Говард Айкен, фирма IBM, механическая машина «Марк-1»,
- 1947 г., Говард Айкен, фирма IBM, электромеханическая машина «Марк-2», умножение 0,7 секунд
- 1957 г., Н. И. Бессонов, СССР, электромеханическая машина «РВМ-1», умножение за 0,05 с., лучшая в мире релейная машина



Электромагнитное реле (принципиальная схема)



Фрагмент машины «Марк-1»

Электронные вычислительные машины (ЭВМ) или компьютеры, середина XX века

- 1937–1942 г.г., Дж. Атанасов и К. Берри, США, первая полностью электронная машина «ABC» (Atanasoff-Berry Computer), 600 электронных ламп накаливания. Только операции сложения и вычитания.

Электронная лампа накаливания

Электронные вычислительные машины (ЭВМ) или компьютеры, середина XX века

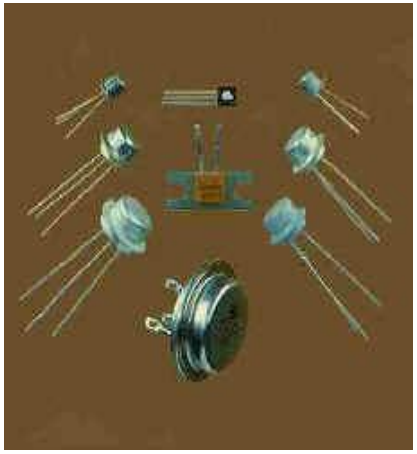
- 1943–1945 г.г. Пенсильванский университет, США, Д. Мочли и П. Эккерт, ENIAC — Electronic Numerical Integrator And Computer, вес 30 тонн, высота 6 метров, площадь120 м^2 , 18 тысяч электронных ламп накаливания, 5 тысяч операций в секунду

Машина ЭНИАК

Электронные вычислительные машины (ЭВМ) или компьютеры, середина XX века

- 1944–1945 г.г. Джон фон Нейман, принципы разработки и функционирования ЭВМ
- 1949 г. М. Уилкс, Великобритания, первая электронная машина с хранимой программой «EDSAC» (Electronic Delay Storage Automatic Calculator). С этой машины принято вести отсчет ***первого поколения компьютеров***.
- 1947–1951 г.г., С.А. Лебедев, СССР, машина МЭСМ
- Середина .60 годов — появление науки ***информатика*.**

Лампа накаливания



Транзисторы

Интегральные схемы

Переход человеческой цивилизации в информационный этап развития (конец XX — начало XXI века)

- 1969 г. , первые элементы будущей глобальной сети Internet.
- 1974 г., первый персональный компьютер Altair 8800.
- 1981 г. , первый персональный компьютер модели IBM PC
- 2006 г., суперЭВМ Blue Gen/L – 280 триллионов операций в секунду
- 2018 г. суперЭВМ Summit – 200 000 триллионов операций в секунду
- 2021 г. суперЭВМ Fugaki – 442 000 триллионов операций в секунду (442 петафлопса), 7 630 848 ядер. Пиковая скорость простых расчётов 2 000 000 триллионов (2 экзафлопса).

Важнейшей характеристикой компьютера является его **быстродействие** (**скорость вычислений**, **производительность**, **мощность**), которое в простейшем случае определяется как количество арифметических операций, выполняемых компьютером за одну секунду. Такая единица измерения быстродействия называется флопсом и обозначается FLOPS (от Floating-point Operations Per Second – операций с плавающей точкой в секунду). Операциями с плавающей точкой считаются операции, аналогичные умножению многозначных чисел.

Информатика представляет собой **фундаментальную** естественную науку о структуре и общих свойствах информации, а также об осуществляемой преимущественно с помощью автоматизированных средств целесообразной обработке информации, рассматриваемой как отображение знаний и фактов, сведений, данных в различных областях человеческой деятельности. Это наука о средствах, методах и способах сбора, обмена, хранения и обработки информации.

Фундаментальной считается наука, изучающая наиболее общие объективные законы природы и общества, осуществляющая теоретическую систематизацию знаний о действительности. Это наука понятия, методы, законы и выводы которой имеют значение для большого круга других научных дисциплин.

Базовые понятия информатики:

- **Информация.**
- **Информационная (математическая) модель.**
- **Алгоритм.**
- **Электронная вычислительная машина (компьютер).**

Основными научными дисциплинами, относящимися к теоретической информатике, являются:

- дискретная математика, дающая математическую основу различным дисциплинам теоретической и прикладной информатики;
- теория информации, изучающая общие свойства информации, а также закономерности, управляющие её рождением, развитием и уничтожением. К теории информации также относятся вопросы передачи информации по различным каналам связи;
- теория кодирования, в задачу которой входит изучение способов представления (кодирования) в компьютерах различных типов информации (текстовой, графической, звуковой и т.д.) с целью обеспечения её надёжного хранения и передачи;
- теория алгоритмов, в задачи которой входит исследование общих свойств алгоритмов, а также способов их представления. Кроме того, к задачам теории алгоритмов относятся доказательство *отсутствия* алгоритмов решения для конкретных задач (доказательство их алгоритмической неразрешимости);
- теория сложности алгоритмов, которая включает в себя асимптотический ана лиз сложности алгоритмов, классификацию алгоритмов в соответствии с классами сложности, разработка критериев сравнительной оценки качества алгоритмов и т. п.;

- теория систем, занимающаяся изучением общих свойств, видов и типов систем; а также основных принципов и закономерностей поведения, функционирования и развития систем;
- теория автоматов, дающая основу для создания программных систем, управляющих работой различных автоматических устройств, таких как, например, банкоматы, торговые автоматы и т. д.;
- теория формальных языков и трансляторов, занимающаяся изучением вопросов создания языков программирования и их трансляции;
- реляционная алгебра, являющаяся основой построения баз данных;
- криптография, служащая теоретическим фундаментом для построения систем защиты программ и данных.

К области прикладной информатики относятся следующие разделы и дисциплины:

- архитектура компьютеров;
- операционные системы;
- языки программирования и методы трансляции;
- технологии программирования;
- информационные системы и технологии;

- системное программное обеспечение;
- прикладное программное обеспечение;
- базы данных и системы управления базами данных;
- базы знаний;
- искусственный интеллект;
- компьютерная графика;
- распределенные системы;
- параллельные системы;
- компьютерные сети и интернет;
- безопасность информационных систем;
- вирусология.

Тема 3. Информация и сообщения

Общепринятое понятие информации:

Информация (от лат. informatio разъяснение, изложение, осведомленность) — первоначальное значение: сведения передаваемые между людьми устным, письменным или иным способом. С середины XX века —понятие, включающее обмен сведениями между любыми двум объектами или субъектами, обмен сигналами в животном и растительном мире и т.д.
(Большой энциклопедический словарь, 1990 г.).

Философское понятие информации:

Информация есть объективное отражение реального мира, атрибут материи, отображающий ее структуру. Эволюционный ряд познания материи, природы, форм проявления материи: **вещество — энергия — информация**

Основоположник кибернетики Норберт Винер говорил об информации так: «Информация есть информация, а не материя и не энергия». То есть, по его мнению, информация относится к *фундаментальным понятиям*, которые не сводятся к каким-либо другим.

Существует большое количество других трактовок информации. Однако большинство учёных и специалистов в настоящее время придерживаются следующего представления об определении этого понятия:

Математически строго определить понятие «информация» невозможно, поскольку по своей сути оно относится к так называемым *первичным*, *неопределяемым* понятиям.

В математике и в других науках существуют такие понятия, дать строгое определение которым *принципиально невозможно*. К ним относятся, например, понятия «множество», «точка» и некоторые другие. Любая попытка каким-либо образом определить их сведется к использованию синонимов, которые в свою очередь окажутся неопределёнными.

В рамках предмета информатика предлагается опираться на следующее *объяснение*, *трактовку* понятия информация:

Под **информацией** понимается отображение в *сознании* человека полученных им сведений (фактов, данных).

Самым характерным свойством информации, существенно отличающим это понятие от таких базовых понятий как вещество и энергии, является свойство «нетеряемости при передаче»: если один человек передаёт другому человеку некоторую информацию, то после передачи этой информацией обладают в равной мере и передающая и принимающая стороны. То есть, сторона, передающая информацию, *её не теряет*. Если же передается вещество или энергия, то по законам сохранения количество вещества или энергии у передавшей стороны уменьшается ровно на ту величину, которая была передана.

В соответствии с приведённой выше трактовкой информации как «отображения в сознании человека» для её появления необходим *субъект*, получающий информацию — **получатель информации**.

Очевидно, должен существовать также *объект* или *субъект*, передающий её — **источник информации**. Следует заметить, что источник и получатель информации могут совпадать.

Получателем информации может быть только субъект — одушевлённая сущность — человек. А вот источником информации может быть как субъект, так и любой объект — неодушевленная сущность — техническое устройство, природный или искусственный объект, явление.

Источник и получатель информации могут быть разделены как в пространстве, так и во времени.

Рассматривать случай *одновременного* нахождения источника и получателя информации *в одной и той же* точке пространства бессмысленно.

Если имеется разделение во времени, то необходимо организовать *хранение* информации, а если имеется разделение в пространстве, то её *передачу*.

В любом случае, как для хранения, так и для передачи необходим некоторый *материальный* **носитель информации**, с помощью которого она попадает от источника к получателю.

В качестве носителя может быть использована бумага, на которой что-либо написано, колебания воздуха, передающие устную речь или музыку, электромагнитные колебания, передающие изображения и т.д.

Кроме того, при *передаче* информации её носитель должен переместиться в пространстве из точки передачи в точку приёма *через некоторую материальную среду*, которую принято называть **каналом связи**.

Информация попадает к её получателю в результате контакта *материального* носителя информации с *материальными* органами чувств человека.

Результатом такого контакта является осознание, выявление человеком смысла принятой информации. Например, один человек слышит произнесённые другим человеком слова и понимает, где и когда должна произойти встреча.

Это **понимание**, которое *формируется*, *фиксируется в сознании* человека и есть принятая им информация. Это понимание, собственно говоря, имеется в виду в приведенной выше трактовке понятия «информация», которая представляет собой некоторое «*отображение в сознании человека*».

Из сказанного выше вытекает, что информация существует только в *сознании её получателя* и только в процессе её получения, её осознания, выявления её смысла.

Спустя некоторое время принятая информация либо *безвозвратно теряется* (человек забывает полученные сведения), либо переходит в его *знания*

Отметим, что принятая информация в общем случае может оказаться неточной, неполной, искаженной или вообще ложной. Поэтому её необходимо ещё сопоставить с реальным миром, проверить на соответствие фактической ситуации

Понимание подразумевает умение выявить смысл сообщения, которым представлена информация, и установить степень её соответствия реальному миру

Итак, выявлены следующие основные *аспекты* информации:

- *материальное* представление информации в виде носителя, который используется для её хранения и передачи, а также в процессе обработки;
- происходящее в сознании человека *субъективное*, *нематериальное* понимание, которое включает в себя:

  - ❖ осознание смысла, значения полученного;
  - ❖ проверку на соответствие выявленного содержания, смысла реальному миру, фактической ситуации.

Таким образом, можно заметить, что вся сложность в использовании понятия «информация» является следствием того, что информация возникает только при *взаимодействии субъективного сознания* человека *с материальным носителем информации* и реальным миром.

Чтобы все-таки иметь возможность объективно исследовать свойства информации, связанные с её хранением и передачей, а также использовать неодушевленные устройства для её обработки, целесообразно *отделить друг от друга* субъективные и объективные аспекты информации.

В теории информации такое отделение основано на использовании дополнительного *неопределяемого* понятия «сообщение». Это создаёт возможность все материальные аспекты связывать с понятием «сообщение», а все субъективные — с понятием «информация».

Сообщение — это *конкретная* материальная форма представления информации

Информация — это абстрактный, нематериальный смысл, извлекаемый человеком из сообщения.

Сообщение — материальный носитель информации, ***информация*** — нематериальное содержание, смысл сообщения. ***Сообщение*** всегда конкретно и материально, ***информация*** абстрактна и нематериальна.

Слово «абстрактный» здесь означает, что при выявлении смысла сообщения человек полностью отвлекается от его конкретных особенностей — от способа получения (устно, письменно, в виде условного знака), от таких деталей как громкий или слабый звук, какими чернилами, на каком листе бумаги написано и т.д.

Знания, а также отдельные систематизированные и не систематизированные факты, сведения, данные, всегда передаются с помощью *материального сообщения*, а смысл сообщения, то есть собственно передаваемая *информация* выделяется, формируется *в сознании человека* при получении этого сообщения.

Итак, информация, извлеченная человеком из сообщения, и сообщение, несущее эту информацию, всегда связаны друг с другом. Информация не может быть получена человеком без приёма им некоторого материального сообщения.

С этой точки зрения оказывается, что все рассмотренные примеры, которые трактовались как информация (наскальные рисунки, запахи, звуки, бумажные тексты, фотографии и т. д.), на самом деле представляют собой примеры *сообщений*, несущих человеку некоторую важную или же неважную для него *информацию*.

В связи с введением понятия сообщения следует ещё раз отметить принципиальный момент. Одушевлённый субъект — человек может *передать информацию*, создав и передав соответствующее сообщение. Он может также *принять информацию*, выявив смысл принятого сообщения, а *обработав* принятую *информацию* (обдумав её), может предпринять некоторое действие.

Неодушевленные объекты могут работать *только с сообщениями*. Объекты — автоматические устройства, радио и телеприёмники, компьютеры и т.д. могут принимать, передавать и обрабатывать только *сообщения*, получить, передать, обработать и тем более *осознать информацию* они *в принципе не могут*.

Использование понятий «сообщение» и «информация» позволяет строгими научными методами изучать *объективные* свойства сообщений, совершенно не затрагивая при этом принципиально *субъективные* аспекты информации

При этом подходе изучение информации, как смысла принятого сообщения, а также её соответствия реальности выходит за рамки предмета информатика.

В связи с уточнением терминологии в рамках предмета информатики и *разграничением* понятий «информация» и «сообщение», следует определиться также и с терминами «сведения» и «данные». Термин «сведения» в дальнейшем используется как синоним понятия «информация», но в тех случаях, когда следует использовать его множественное число.

Термин «данные» в естественных и технических науках имеет используемый точный смысл:

Данными называются сообщения, представленные в формализованном виде, пригодном для использования в некотором техническом устройстве (измерительном приборе, компьютере и т.д.).

Формализованным называется представление, подчиняющееся точно сформулированной, исчерпывающе полной системе правил

Содержание сообщения, его смысл и значение существенно зависит от получателя, от того насколько он сумел понять сообщение, насколько его смысл важен для этого конкретного человека.

Одно и то же сообщение, полученное разными людьми, может ими восприниматься по-разному, его важность и ценность для разных людей могут быть различными.

Таким образом, соответствие между информацией и сообщением, с помощью которого она передается, не является однозначным.

Это значит, что, во-первых, из одного и того же сообщения разные люди могут извлечь различную информацию.

И, во вторых, что для более важно для практики, *одну и ту же информацию можно передать с помощью разных сообщений*.

Этот важный момент можно трактовать по-другому, а именно, можно считать, что существует только *одно* сообщение, но применяются *разные* способы его записи, кодировки.

Это значит, что сообщение может быть без потери его смысла, то есть без потери заключенной в нём информации, представлено, зашифровано, закодировано разными способами.

Особенно важна эта возможность для организации обработки произвольных сообщений с помощью компьютера: все сообщения, вне зависимости от исходной формы представления и без потери содержащейся в них информации, преобразуются к используемой в компьютере форме.

Другими словами, преобразование сообщений в *компьютерные данные* может быть произведено *без потери исходной информации*.

В связи с выявленной неоднозначность возникает вопрос о способе, качестве и полноте *извлечения* информации из сообщения.

Анализ показывает, что способ выявления смысла сообщения (извлечения информации), может быть *общепринятым*, как например, в случае передачи сообщений на естественных языках, либо должны использоваться некоторые *специальные правила*, которые известны и отправляющей и принимающей сторонам.

Такие правила могут быть доступными любому человеку (например, правила дорожного движения), либо же они являются секретными и известными только узкой группе лиц (скажем, при применении специальных методов шифрования).

Решающим фактором для обеспечения возможности извлечения информации из сообщения является *знание языка*, на котором сформулировано сообщение, и/или *способа его шифрования* (*кодирования*).

Если сообщение передается на неизвестном слушателю языке или же если оно неизвестным образом зашифровано, его получатель не сможет извлечь из такого сообщения вообще никакой информации. При этом другие его получатели могут понять сообщение полностью или частично

Пример сообщения:

П р и х о д и н е м е д л е н н о

В любом случае смысл сообщения выявляется с помощью некоторого правила, которое принято называть **правилом интерпретации сообщений**.

Математическая запись правил интерпретации сообщения: $s \xrightarrow{\alpha} i$

где: s — конкретное сообщение, i — полученная информация,

α — использованное правило интерпретации сообщения

Язык интерпретации сообщений

Множество правил $A, \alpha \in A$ Множество сообщений $S, s \in S$

$$S \xrightarrow{A} I \qquad A: S \to I \qquad I = A(S)$$

Тройка множеств (I, S, A) называется ***информационной системой***

Свойства информации:

- Объективность
- Полнота
- Достоверность
- Адекватность
- Доступность
- Актуальность

Объективность — свойство информации, определяющее степень её зависимости от человека, объективность информации тем больше, чем меньше в ней субъективности.

Полнота — свойство информации исчерпывающим образом характеризовать отображаемый объект или процесс. Полнота информации определяет её достаточность для принятия того или иного решения, выполнения тех или иных действий.

Достоверность — это свойство информации не иметь скрытых ошибок, то есть искажений, о которых получателю ничего не известно.

Под **адекватность** информации понимается уровень ее соответствия реальному, процессу, явлению, объективному миру. Информация может быть *достоверной*, но недостаточно хорошо описывающей ситуацию, явление, объект, поскольку некоторые важные аспекты в ней могут отсутствовать.

Актуальность — это способность информации соответствовать нуждам её получателя в некоторый момент времени. В частности, это степень соответствия текущему моменту времени.

Доступность — это свойство информации, характеризующее возможность её извлечения данным человеком из данного сообщения. Это мера возможности по лучения информации, которая определяется, во-первых, доступностью сообщения, а, во-вторых, доступностью способа извлечения информации.

Действия над сообщениями

Основные операции:

- передача
- прием
- обработка
- хранение

Дополнительные операции :

- сбор
- формализация
- фильтрация
- сортировка
- архивация
- защита

Сбором сообщений (более привычная, но не совсем точная форма речи — сбор информации) называется деятельность человека или технического устройства, в ходе которой они получают требуемые сведения.

Формализация сообщений представляет собой их приведение к некоторой *единой форме* для обеспечения возможности сопоставления различных сообщений.

Фильтрация сообщений — это отсеивание помех, шумов, ненужных или пустых сообщений, что повышает их адекватность и достоверность,

Сортировкой сообщений называется процесс изменения порядка их следования, с целью размещения сообщений в порядке возрастания или убывания определенного признака или набора признаков.

Архивация сообщений это создание их резервных копий, обеспечивающих повышение надежности хранения и возможность восстановления сообщений, утраченных или искаженных по каким-либо причинам.

Защита сообщений это комплекс мероприятий, направленных на предотвращение их утраты, воспроизведения или изменения.

Хранение сообщений связано с фиксацией некоторого состояния носителя, а **передача** связана с изменением состояния носителя.

Однако организация хранения сообщений на любом носителе, обязательно связана с первичной передачей этого сообщения на носитель, во время которой сообщение некоторым образом фиксируется на нём. Таким образом сначала необходимо выяснить как происходит передача сообщений.

П*ередача* сообщения всегда связана с необходимостью *изменить* носитель во времени или в пространстве, Изменение носителя можно понимать как любое изменение его **состояния**, то есть любое изменение его свойств, характеристик.

Отсюда следует, что передача сообщений с помощью носителя с *неизменяющимися* свойствами *невозможна.*

В теории связи и теории информации *зафиксированные изменения* во времени или в пространстве состояние, то есть свойств или характеристик наблюдаемого объекта или явления принято называть **сигналом**. При этом изменяющаяся характеристика объекта называется ***параметром сигнала.***

Отметим, что на бытовом уровне термин «сигнал» достаточно часто означает некоторый *кратковременный* процесс (звук сирены, сигнальная ракета). В теории связи и теории информации сигнал может иметь *любую длительность*.

Итак, в общем случае сообщение представляет собой некоторую последовательность физических сигналов, зафиксированных тем или иным образом.

Важно понимать, что передача сообщения всегда занимает некоторое время, *происходит во времени*, то есть представляет собой некоторый *процесс*. А понятие процесса базируется на понятии состояния, поэтому необходимо более точно выявить его смысл.

Под состоянием объекта, явления понимается некоторый набор его свойств, характеристик, *значений параметров*, которые, собственно говоря, и определяют состояние, жёстко связаны с ним.

Например, можно рассматривать следующие состояния человека: «стоит», «идёт», «бежит». Они отличаются друг от друга разными значениями *одного параметра* скорость тела.

Постоянное, неизменное значение параметра скорость тела или же *его изменение в определённых границах* означает, что текущее состояние человека не меняется. А изменение скорости с выходом за установленные границы приводит к переходу из одного состояния в другое.

Под **состоянием** объекта или явления понимается набор *стабильных* значений группы параметров, которые адекватно описывают рассматриваемый объект или явление с точки зрения решаемой задачи. Значение считается **стабильным**, если оно не изменяется во времени или изменяется в заданных пределах.

Процессом (лат. processus — продвижение) называется *последовательная* смена состояний наблюдаемого объекта или явления во времени.

В качестве примера *процесса*, как смены состояний при изменении параметров, можно указать превращение жидкости в твёрдое тело при замерзании, или в газ при испарении. Такое изменение может быть вызвано внешним воздействием, например, нагреванием вещества или же внутренними причинами, например, некоторой химической реакцией, происходящей в веществе.

Информационным процессом называется изменение со временем содержания информации или представляющего информацию сообщения. К информационным процессам относятся, например, приём, передача и обработка сообщений.

Все процессы делятся на две группы — **стационарные** и **нестационарные**

Стационарными называются объекты, явления, процессы с *неизменяющимися* во времени характеристиками. Объект, явление, процесс считается нестационарным при наличии какого-либо изменения любой его характеристики.

Примеры стационарных процессов — постоянный, равномерный гул; синусоидальное колебание с постоянной амплитудой, фазой и частотой.

Передача сообщений с помощью стационарных процессов *невозможна*, для передачи могут быть использованы только нестационарные процессы.

Можно также выделить группы **дискретных** и **непрерывных** процессов.

У *дискретных* процессов все *временные состояния* чётко отделены друг от друга ненулевым отрезком времени, и для любого состояния можно указать одно или два *соседних*.

Соседними называются два последовательных состояния процесса, между которыми *нет никаких других его состояний*. Начальное и конечное состояние имеют ровно один соседний (последующий и предыдущий соответственно), а все остальные — ровно два соседних.

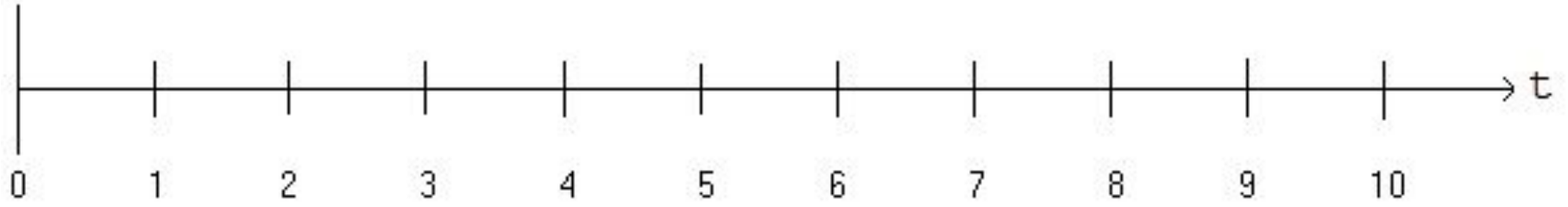
Все возможные состояния дискретного процесса всегда могут быть перенумерованы целыми числами, то есть их общее количество *конечно* или *счётно*.

Часто говорят, что дискретный процесс представляет собой последовательную смену состояний объекта, системы, явления, которая происходит в так называемом дискретном времени.

Дискретное время представляет собой конечную или счётную последовательность временных отсчётов $t \in \{0, 1, 2, \ldots\}$, в которые происходит переход дискретного процесса из одного состояния в другое, с ненулевыми промежутками времени между ними.

Моменты времени , в которые заданы или измерены значения параметра сигнала, принято называть **отсчётами**.

а)дискретное время



б)непрерывное время

Между любыми двумя соседними состояниями дискретного процесса *всегда* имеется *ненулевой* отрезок времени. Если все такие отрезки времени *равны друг другу*, то во многих случаях их называют **тактами**.

Происходящий за ненулевое время возврат процесса в некоторое исходное состояние является обязательным условием существования любого физически осуществимого *дискретного* процесса. Наличие этого ненулевого отрезка времени в любом дискретном процессе *принципиально* и *неустранимо*, так как оно является следствием фундаментальных законов природы

У *непрерывных* процессов между двумя *любыми* двумя состояниями всегда можно выделить *любое количество промежуточных*. К непрерывным процессам понятия соседних состояний *неприменимо*.

Для отдельных состояний непрерывных процессов можно говорить только о порядке их следования во времени — предшествующее или последующее состояние. Всегда можно найти два последовательных состояния процесса, которое отделены друга от друга сколь угодно маленьким временным отрезком.

Множество различных состояний непрерывного процесса *несчётно* (имеет мощность континуума).

Непрерывные процессы происходят в непрерывном времени, которое совпадает с физическим временем.

Параметр сигнала, служащий для передачи сообщения, по определению представляет собой зависящую от времени величину, является функцией времени $A = f(t)$

Изменение во времени значений параметра сигнала *A* естественно рассматривать как *процесс*, при этом значения функции *f(t)* в выбранные моменты времени являются его временными состояниями. Этот процесс может быть непрерывным или дискретным.

Это значит, что значения сигнала *A=f(t)* могут быть заданы в непрерывном времени, то есть в *любой* момент времени *t* из области его определения $T_0 \leq t \leq T_N$ (непрерывный процесс) либо в дискретном времени, то есть только в некоторые выбранные моменты времени $t_0, t_1, \ldots, t_N$ (дискретный процесс).

В первом случае сигнал называется непрерывным, а в втором – дискретным.

Понятия непрерывности и дискретности процесса связаны с множеством, к которому принадлежит *аргумент* функции . Естественно распространить эти понятия и на множество *значений параметра* сигнала.

Так, если параметр сигнала *A* принимает любые значения из некоторого интервала *Amin≤A≤Amax*, то есть множество *E* его значений *несчётно*, то этот сигнал можно рассматривать как непрерывный *по значению*.

Если все значения параметра *A* принадлежат *конечному* множеству $E=\{A_0, A_1, \ldots A_M\}$ — такой сигнал естественно считать дискретным *по значению.*

Итак, понятия непрерывности и дискретности относятся как к области определения — отрезку времени, в течение которого сигнал наблюдается, так и к области значений его параметра. Следовательно, возможны четыре различных типа сигналов:

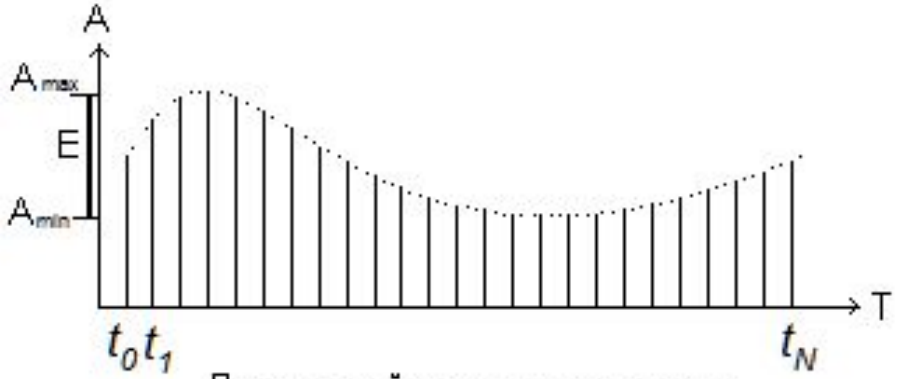
- непрерывный и по времени и по значению;
- дискретный по времени и непрерывный по значениям;
- непрерывный по времени, но дискретный по значениям;
- дискретный и по времени и по значениям.

Сообщения, основанные на непрерывных сигналах называются непрерывными, сообщения, основанные на дискретных сигналах, называются дискретными.

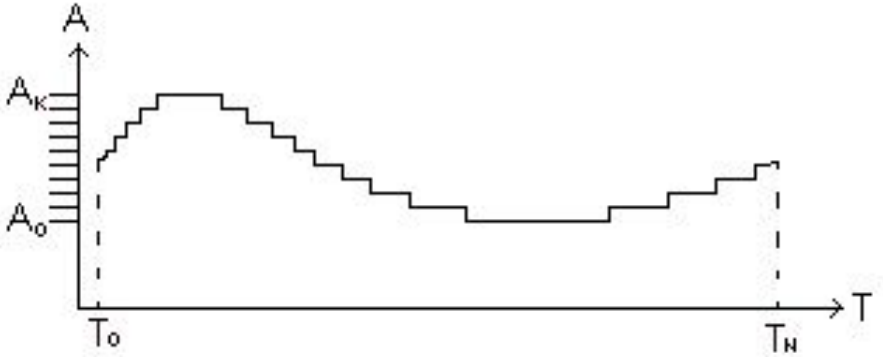
Информация не обладает ни свойством непрерывности, ни свойством дискретности

Непрерывные и дискретные сигналы

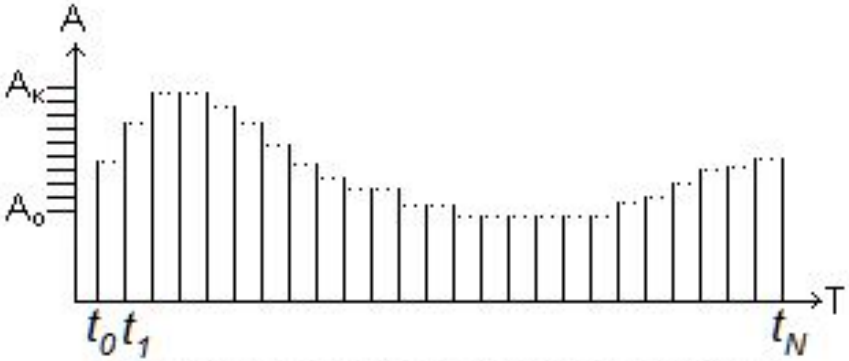
Непрерывный сигнал



Дискретный по времени сигнал



Дискретный по уровню - квантованный сигнал



Дискретныи по времени и уровню сигнал

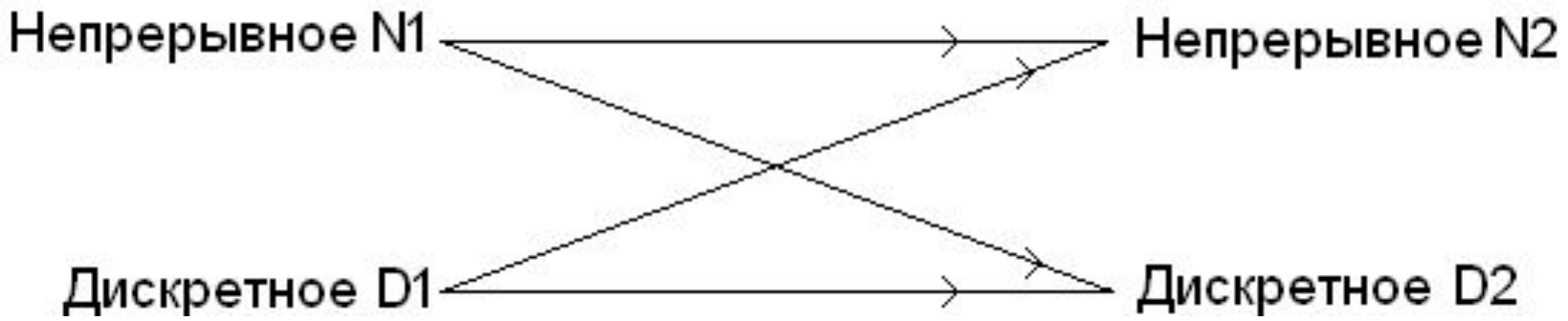
Следует отметить, что в связи с особенностями органов чувств человека некоторые непрерывные сообщения воспринимаются человеком как дискретные. Так, например, звуковые, акустические колебания физически являются непрерывным процессом.

Но слуховая система человека способна различать только 512 различных уровней громкости и 2048 различных уровней высоты звука.
Это означает, что любой звуковой сигнал после его приёма и обработки органами слуха человека фактически становится *дискретным* по значению параметра сообщением. Аналогичными свойствами «естественной дискретности» обладают и другие органы чувств человека.

Однако у органов чувств человека имеются и противоположные свойства, которые приводят к тому, что фактически дискретные сообщения воспринимаются им как непрерывные.

Например, на *неспособности* зрительной системы человека различать более 24-х последовательно сменяемых в течение одной секунды изображений одного и того же объекта основано кино. Если изображения предъявлять со скоростью, например, 20 кадров в секунду, то они будут восприниматься сознанием человека как отдельные быстросменяемые кадры, а если со скоростью 24 или более — как одно непрерывное, которое создает в сознании впечатление естественного движения объектов съемки.

Преобразование сообщений



Преобразования вида N1→N2 используются в самых разных технических устройствах, таких как, например, микрофон, телефон, магнитофон, радио и телевизионные приёмники и т.д.

Преобразование N1→N2 всегда сопровождается потерей информации. Это *принципиальное* и *неустранимое* обстоятельство, вытекающее из самой природы непрерывных сообщений.

Остальные преобразования могут быть выполнены без потери информации

Преобразования вида D1→D2 фактически являются разновидностью обработки дискретных сообщений. Эти преобразования в зависимости от используемого способа могут быть проведены как с потерей, так и без потери информации.

Преобразования вида N→D и D→N имеют большое значение для организации обработки *непрерывных* сообщений с помощью компьютера.

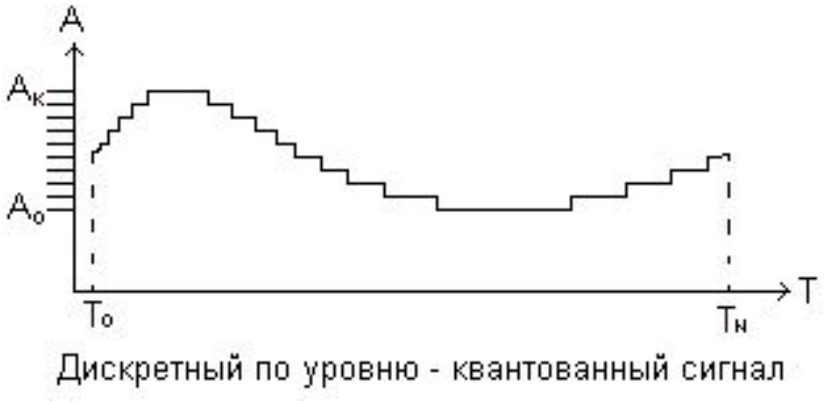
Получение по непрерывному сообщению соответствующего ему дискретного называется **дискретизацией**, а получение по дискретному сообщению соответствующего ему непрерывного называется **восстановлением**. Дискретизация сигнала осуществляется с помощью ***развертки по времени*** и ***квантования*** сигнала по уровню.

Развертка сигнала по времени

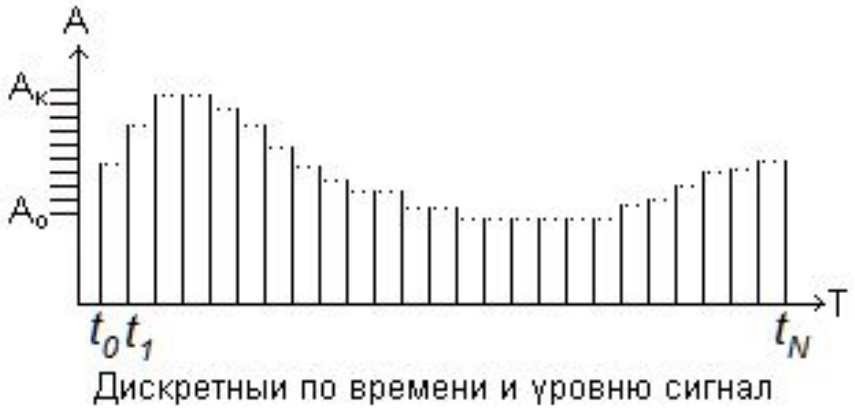
$$A = f(t), t \in [t_0, t_N] \rightarrow A_i = f(t_i), t_i = t_0 + i \times \Delta t; \Delta t = (t_N - t_0)/N; i = 0,1,..., N$$

Квантование сигнала по уровню

$$E = [A_{\min}, A_{\max}] \rightarrow \{A_0, A_1, ..., A_K\}; A_i = A_{\min} + i \times \Delta A; \Delta A = (A_{\max} - A_{\min})/K$$



Квантованный по уровню и развернутый по времени — дискретный сигнал



Вообще говоря, дискретизация приводит к потере информации, которая имелась в исходном непрерывном сообщении. Если при выполнении развёртки по времени и квантовании выбирать небольшие значения *N* и *K* , которые определяют количество узлов развёртки и уровней сигнала соответственно, то потеря информации неизбежна и велика.

Увеличивая значения *N* и *K* можно уменьшить потерю информации. Но возможно ли осуществить дискретизацию вообще без её потери?

В математическом анализе в теории рядов Фурье доказывается, что удовлетворяющая некоторым требованиям *периодическая непрерывная* функция может быть представлена в виде суммы *бесконечного* количества слагаемых

$$f(t) = \sum_{k=0}^{\infty} a_k \sin(\omega_k t + \varphi_k) = a_0 + a_1 \sin(\omega_1 t + \varphi_1) + a_2 \sin(\omega_2 t + \varphi_2) + \ldots$$

где a_k, ω_k и φ_k — амплитуда, *круговая* частота и фаза *k*-ой синусоиды.

Переход от исходной функции к её представлению в виде такой суммы принято назвать **преобразованием Фурье**, а отдельные слагаемые — **гармониками**.

Теория рядов Фурье является важным инструментом для решения задач, связанных с преобразованиями сообщений вида D→N. Дело в том, что параметры гармоник (синусоид) — коэффициенты a_k и φ_k находятся по вычисленным или измеренным *дискретным отсчётам сигнала* с помощью довольно простых соотношений, а частоты ω_k однозначно связаны с периодом или длительностью сигнала *T* – $\omega_k = k \times \omega_1$, $\omega_1 = 2\pi/T$.

Рассчитанные параметры *любой гармоники* можно использовать для воспроизведения соответствующего непрерывного синусоидального сигнала на звуковых колонках компьютера. *Одновременное* воспроизведение колонками всех гармоник представляет собой их суммирование, то есть это непрерывный сигнал (речь, музыка и т.д.). Так происходит *восстановление* сигнала по его отсчётам, то есть преобразование дискретного сигнала в непрерывный.

С другой стороны, для преобразований вида N→D, анализ частот гармонических составляющих суммы позволяет *выбрать количество отсчётов*, необходимое для того, чтобы выполнить *дискретизацию без потери информации*.

В общем случае суммы в разложении Фурье содержат счётное количество слагаемых, но во многих практически важных *частных* случаях их количество оказывается *конечным* и, следовательно, *ограниченным*. Например, речь человека не содержит частот больших, чем f_{max} =4000 герц. $\omega_{\max=} 2\pi f_{\max}$

Сигналы, создаваемые любыми реальными техническими устройствами, также имеют ограниченный набор гармоник, то есть существует конечная частота $\mathbf{f}_{max}$ такая, что все $\omega_k \leq \omega_{max}$.

Теорема отсчетов В.А. Котельникова (1933 год).

Непрерывный сигнал можно полностью отобразить и точно воссоздать (восстановить) по последовательности измерений его значений или отсчётов величины этого сигнала через одинаковые интервалы времени Δt, меньшие или равные половине периода максимальной частоты ω_{max}, имеющейся в сигнале.

$$\Delta t \leq \frac{1}{2 f_{\max}} \qquad N \geq 2 \times f_{\max}$$

При дискретизации аналогового сигнала выбранное количество отсчётов на одну секунду длительности сигнала называется **частотой дискретизации.**

Например, для точной передачи *речевого* сигнала с максимальной частотой 4 000 Гц при его развертке следует выбрать частоту дискретизации не менее 8 000 отсчётов в секунду (8 килогерц).

Для качественного воспроизведения звукозаписей с помощью компьютера в настоящее время используются частоты дискретизации 44 килогерца и более. Это связано с тем, что слуховая система человека *не воспринимает* частоты выше, чем 20 000–22 000 герц. Следовательно, частота дискретизации в 44 000 герц сохраняет для человека все слышимые им звуки в неискажённом виде.

Теорема В.А. Котельникова дает рекомендации по выбору необходимого количества отсчётов для выполнения развёртки сигнала во времени. Но для полной дискретизации необходимо выполнить ещё и квантование сигнала по уровню.

Так, для квантования звука можно ограничиться 512-ю уровнями громкости, поскольку слуховая система человека большее количество уровней просто не воспринимает.

А при квантовании изображений для *естественного* восприятия цвета следует выделять не менее чем 2^{24} = 16 777 216 цветовых градации, которые способны распознать органы зрения человека.

НАБОРЫ ЗНАКОВ И АЛФАВИТЫ

С точки зрения информатики принципиальная разница между непрерывными и дискретными сигналами состоит в том, что *дискретные* сигналы можно *обозначить*, то есть закрепить *за каждым* из возможных значений параметра сигнала, за каждым его уровнем (а их количество *конечно*) некоторый *условный знак*.

Так при квантовании сигнала, за его уровнями закреплены знаки $\{A_0, A_1,\ldots,A_K\}$

С учётом того, что параметр изображённого на этих рисунках сигнала может принимать всего десять различных значений можно было бы выбрать какие-либо другие *наборы знаков*, например, {0, 1, 2, …,9} или {а, б, в, …, и}

Набором знаков называется конечное множество объектов, выбранных для закрепления за значениями параметра (уровнями) дискретного сигнала. **Знаком** называется любой элемент этого множества.

Природа знаков, выбираемых для обозначений уровней сигналов, может быть любой: буквы, цифры, отдельные звуки, жесты, рисунки, сигналы светофора, условные знаки на чертежах и топографических картах, знаки дорожного движения, нотные знаки, знаки зодиака, знаки игральных карт и т.д.

Если сообщение имеет форму письма, то знаками являются ***графемы***, в случае устной речи в качестве знаков выступают ***фонемы*** (элементарные составляющие речи).

Удобно выбирать знаки так, чтобы каждому уровню сигнала соответствовал ровно один знак, и при этом разным уровням соответствовали разные знаки. Другими словами, чтобы между множеством уровней сигнала и множеством знаков имело место взаимно однозначное соответствие (биекция).

Тогда по любому заданному знаку можно однозначно определить соответствующий ему уровень сигнала, и, наоборот, по заданному уровню однозначно определить соответствующий ему знак. В чём недостаток такого подхода?

Поэтому для практического использования выбираются наборы, состоящие из относительно небольшого количества знаков, а уровни сигнала обозначаются некоторыми *последовательностями знаков, какими-либо их комбинациями*.

При этом сохраняется сформулированное выше требование: каждому уровню должна соответствовать только одна *комбинация знаков*, при этом разным уровням должны соответствовать различные их комбинации.

Важнейшим моментом в использовании описанного подхода является то, что *один и тот же* специально выбранный набор знаков может быть использован для обозначения множеств уровней самых разных по своей физической природе сигналов.

Это создает практическую возможность для представления дискретных сообщений *произвольной природы* в технических устройствах, предназначенных для их хранения или обработки.

Множество знаков, в котором определен линейный порядок называется ***алфавитом***.

Алфавит, как правило, задается прямым перечислением всех входящих в него знаков. При этом знаки перечисляются в порядке их следования. Примеры алфавитов:

- {а, б, в, г, д, е, ё, ж, з, и, й, к, л, м, н, о, п, р, с, т, у, ф, х, ц, ч, ш, щ, ъ, ы, ь, э, ю, я} — 33 буквы русского языка;
- {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z} — 26 букв английского языка;
- {0, 1, 2, …, 9} — 10 цифр арабской десятичной системы счисления;
- {·, –} — два знака азбуки Морзе, которая ранее широко использовалась для радиосвязи и т.д.

Поскольку при передаче дискретных сообщений параметр сигнала *должен меняться*, очевидно, что *минимальное* количество знаков в алфавите равно *двум*. Такие алфавиты называются **двоичными**.

Примеры двоичных алфавитов: {0,1} ({O, L}, {.,-}, {пробивка, нет пробивки}, {true, false}, {истина, ложь},{да, нет}, пара напряжений {2в, 12в} и т.д. Знак двоичного алфавита принято называть ***двоичной цифрой или битом*** (bit — binary digit)

Так как этот код используется для хранения и обработки данных в компьютерах — вычислительных *машинах* — его называют ещё и **машинным кодом.**

Пусть $A=\{\alpha_1, \alpha_2,\ldots\alpha_N\}$— используемый алфавит, $\alpha_i \in A$, $i=1,2,\ldots,N$ — его знаки, N — количество знаков в алфавите.

По определению любое дискретное сообщение S представляет собой некоторую *последовательность* квантованных отсчётов, а каждый отсчёт можно обозначить некоторым знаком $\alpha_j \in A$.

Следовательно, дискретное сообщение можно представить, *записать* в виде *последовательности знаков*, принадлежащих алфавиту: $S=a_1,a_2,\ldots,a_m$, где a_j— очередной знак сообщения, являющийся одним из знаков алфавита $a_j \in A$, $j=1,2,\ldots,m$; m — количество знаков в сообщении.

В такой записи запятые, которые отделяют знаки последовательности друг от друга, чаще всего пропускаются и тогда сообщение принимает вид $S=a_1a_2\ldots am$

Например, последовательность знаков русского алфавита S= и, н, ф, о, р, м, а, ц, и, я или в более кратком виде S= информация — является сообщением, состоящим из m=10 знаков.

А последовательность из m= 8 знаков: 0, 0, 1, 0, 1, 0, 0, 1, 1, или в более наглядном виде «00101001» записана в двоичном алфавите A={0,1}

Любое дискретное сообщение можно представить в виде последовательности знаков из некоторого фиксированного алфавита. Произвольная последовательность знаков $a_1a_2…a_m$ принадлежащих фиксированному алфавиту *A* называется также **цепочкой над алфавитом** *A.*

Количество знаков *m*, из которых состоит цепочка α называется **длиной цепочки**. Длину цепочки обозначают |α|. Например, |информация|=10, а |00101001 |=8.

Допускаются цепочки нулевой длины, такие цепочки называются принято называть **пустыми**. Пустые цепочки обычно обозначаются буквой λ, |λ|=0.

Кодирование сообщений

Дискретное сообщение исходя из технических соображений или из особенностей органов чувств человека, участвующих в приеме или передаче сообщений, разбивают на конечные подпоследовательности знаков, которые принято называть ***словами***. Количество знаков в слове называется ***длиной слова.*** Если все слова языка имеют одно и то же количество знаков, то такие слова называются ***n-разрядными*** (n ─ длина слова)

Слова записанные в двоичном, шестнадцатеричном и некоторых других алфавитах, принято называть ***кодом*** (двоичное слово ─ двоичный код и т.д.).

Исходный алфавит *A1*, в котором записано сообщение, называется ***первичным*** алфавитом. Целевой алфавит *A2*, в который преобразуется сообщение называется ***вторичным*** алфавитом.

Правило, описывающее соответствие между знаками первичного алфавита и знаками или сочетаниями вторичного называется ***кодом***.

Совокупность всех используемых в преобразовании соответствий называется **кодовой таблицей**. Для обеспечения необходимых свойств операции кодирования между знаками первичного и знаками или группами знаков вторичного алфавитов должно быть установлено *взаимно однозначное соответствие* (биекция), при этом кодовая таблица представляет собой способ задания этого соответствия.

Пример таблицы кодирования:

Знак	Код	Длина кода
А	1110	4
К	10001	5
М	10101	5
Н	1010	4
У	001001	6

Последовательность знаков вторичного алфавита *A2*, которыми представлен *один* знак первичного алфавита *A1* называется **кодовым словом**, **кодовой цепочкой** или просто **кодом знака**.

Кодированием называется последовательность действий по переводу сообщения из первичного во вторичный алфавита.

Декодирование представляет собой операцию, обратную кодированию, то есть это последовательность действий по восстановлению сообщения в исходном алфавите по его виду во вторичном.

Операции кодирования и декодирования считаются ***обратимыми***, если их последовательное применение обеспечивает восстановление исходной информации без потерь.

Кодером называется устройство, обеспечивающее выполнение операции кодирования. ***Декодером*** называется устройство, обеспечивающее выполнение операции декодирования.

Существует огромное количество способов построения кодов при выбранных или заданных первичном и вторичном алфавитах. Поэтому возникает проблема выбора *оптимального*, то есть в некотором смысле наилучшего кода.

Оптимальным называется кодирование сообщения, результат которого обеспечивает *минимально* возможное для используемых алфавитов и передаваемого сообщения время передачи по каналам связи и минимальные требования к памяти при хранении.

Эффективным считается кодирование с результатом *достаточно близким* к оптимальному. Эффективное кодирование применяется в тех случаях, когда для получения оптимального кода требуются значительные временные ресурсы или не хватает каких-либо данных для его построения.

Задачи эффективного и оптимального кодирования часто решаются с помощью **сжатия** сообщений.

Различают два вида сжатия — без потери информации и с потерей информации.

Сжатие без потери информации применяется при кодировании текстовых и числовых видов данных, то есть там, где потеря информации недопустима, поскольку может привести к неправильному пониманию текста или к вычислениям с непредсказуемым результатом.

Сжатие с потерей информации широко применяется при кодировании графики, звука и видео, так как в этих случаях возможно удаление некоторой части данных без существенных отличий для восприятия человеком оригинала от полученного результата сжатия. В основе применения сжатия с потерями лежит естественное квантование органами слуха и зрения человека сообщений, получаемых из внешней среды.

Выбор оптимального кода это *технический и экономический фактор*, поскольку касается затрат (и не только временных) на выполнение кодирования и декодирования, на хранение и на передачу кода из одного места в другое и т.д. Естественно желание сделать такие затраты минимально возможными.

Однако свести их к нулю в принципе невозможно. Существующие границы эффективности кодирования, которые могут быть достигнуты, а также условия, при выполнении которых эти границы достижимы, определены в работах К. Шеннона, сформулировавшего и доказавшего ряд базовых теорем кодирования.

Кодирование текстовых данных

Текстовые данные являются одной из наиболее простых, но при этом чаще всего встречающейся разновидностью данных.

При хранении в компьютере любой текст (компьютерная программа, документ, статья) рассматривается как обычная последовательность знаков.

Причем промежуток между отдельными словами — пробел, переход на следующую строчку, переход на следующую страницу — также могут рассматриваться как *специальные* знаки.

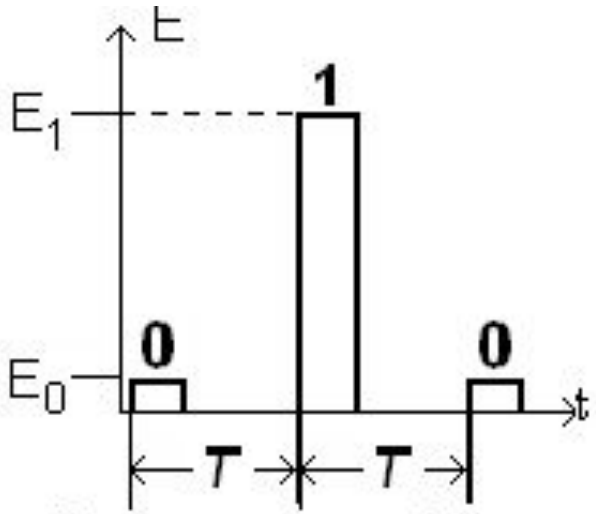
Таким образом, любой текст можно считать *дискретным сообщением*, то есть последовательностью знаков, принадлежащих некоторому алфавиту, который возможно состоит из достаточно большого количества знаков.

Существует множество различных двоичных кодов, которые используются в вычислительной технике для представления текстов. В целом эти способы можно разбить на две группы.

Способы кодирования, опирающиеся на знание частот знаков первичного алфавита, называются **статистическими**. Способы кодирования, которые используют списки неоднократно встречающихся в сообщении подпоследовательностей знаков, называются **словарными**.

Передача кодов сообщения в канал всегда *происходит во времени*. При этом на передачу каждого знака кода требуется некоторый отрезок времени

На рисунке изображена ситуация, в которой элементарные сигналы — импульсы, соответствующие цифре «ноль», имеют *одинаковую* длительность с импульсами, передающими цифру «единица».



Импульсы, несущие 1 бит

В общем случае элементарные сигналы, передающие разные цифры, могут иметь *различную* длительность. В связи с этим выделяются коды с *равными* и с *неравными* длительностями элементарного сигнала.

Кодирование называется **равномерным**, если все коды, закрепляемые за знаками первичного алфавита, имеют одну и ту же длину, то есть состоят из одинакового количества двоичных цифр. Кодирование называется **неравномерным**, если разные коды имеют различные длины.

Кодирование называется **алфавитным**, если за каждым знаком первичного алфавита закрепляется отдельный код. Кодирование считается **блочным**, если отдельные коды выделяются группам (блокам) знаков первичного алфавита.

Определяющие характеристики способов кодирования:

- длительность (одинаковая или разная) элементарных сигналов, которые соответствуют знакам 0 и 1;
- длина кода (одинаковая или разная) для разных знаков первичного алфавита (равномерный и неравномерный коды).
- выделение отдельного кода для каждого знака первичного алфавита (алфавитное кодирование) или возможны коды для сочетаний знаков (блочное кодирование).

Пример: **Азбука Морзе** – *статистический*, *алфавитный*, *неравномерный* код с *неравными длительностями* знаков.

В 1838 году С. Морзе предложил систему кодирования для первой телеграфной линии, которая строилась между городами Балтимор и Вашингтон (США).

Эта система в нашей стране во время Первой мировой войны 1914–1918 г.г. получила название «морзянки» или «азбуки Морзе». Фактически азбука Морзе была *первым цифровым* способом передачи сообщений.

С. Морзе при разработке своей системы кодирования исходил из того, что энергозатраты и общая длительность передачи сообщения должны быть минимальными. Поэтому в азбуке Морзе буквы, которые чаще употребляются в английском языке, имеют более короткие коды, а встречающиеся реже — имеют более длинные коды.

В предложенной им системе каждой букве английского алфавита, а также десятичным цифрам поставлен в соответствие код, представляющий собой некоторую комбинацию длинных (тире) и коротких (точек) импульсов электрического тока по телеграфным линиям. Для передачи букв других алфавитов, в том числе русских букв использовались коды сходных латинских букв, а для букв, не имеющих латинских аналогов, выбирались незанятые комбинации точек и тире.

Азбука Морзе для русского алфавита

Буква	Частота	Код	Длина	Буква	Частота	Код	Длина
Пробел	0,174	0	2	я	0,018	• —• —	5
о	0,090	— — —	4	ы	0,016	—• — —	5
е	0,072	•	2	з	0,016	— —••	5
а	0,062	• —	3	ь,ъ	0,014	—• • —	5
и	0,062	••	3	б	0,014	—• • •	5
т	0,053	—	2	г	0,013	— —•	4
н	0,053	—•	3	ч	0,012	— — —•	5
с	0,045	•••	4	й	0,010	• — — —	5
р	0,040	• —•	4	х	0,009	••••	5
в	0,038	• — —	4	ж	0,007	••• —	5
л	0,035	• —••	5	ю	0,006	•• — —	5
к	0,028	—• —	4	ш	0,006	— — — —	5
м	0,026	— —	3	ц	0,004	—• —•	5
д	0,025	—••	4	щ	0,003	— —• —	5
п	0,023	• — —•	5	э	0,003	•• —• •	6
у	0,021	•• —	4	ф	0,002	•• —•	5

За единицу длительности импульса в системе С. Морзе принимается длительность передачи точки *t*. Длительность передачи тире равняется $3t$ – длительностям передачи трех точек.

Чтобы при приёме можно было различить два соседних знака, *входящих в код одной буквы или цифры*, между ними включается пауза с длительностью *t* передачи точки.

А для разграничения кодов двух соседних букв сообщения делается пауза длительностью $3t$. Промежуток между соседними словами задается паузой длительность $6t$.

Базовая длительность времени передачи точки *t* зависит от индивидуального «подчерка» радистов, от их возможностей и опыта. Радист средней квалификации может вести передачу/приём со скоростью 60–100 знаков ***текста*** в минуту.

В среднем на код одного знака требуется около $6t$. Отсюда базовая единица длительности *t* равна примерно 0,1–0,15 секунды.

Формирование и передача кодов с *неравными* длительностями элементарных сигналов с помощью *технических средств* наталкивается на целый ряд трудностей.

В связи с этим в настоящее время такие системы кодирования в технических системах, как правило, *не применяются*. У всех рассматриваемых в дальнейшем кодов длительности элементарных сигналов ***одинаковы***.

Записать алфавит азбуки Морзе.

Алфавит азбуки Морзе фактически состоит не из двух, а из трёх знаков {точка, тире, пауза} с длительностями (t, $3t$, t) соответственно.

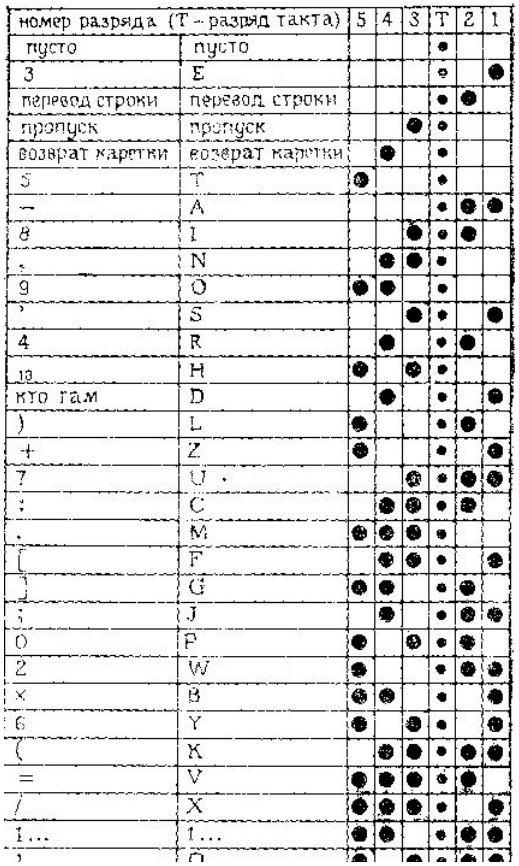
Азбука Морзе более века была важнейшей системой кодирования, повсеместно используемой в телеграфной связи, а впоследствии и в радиосвязи. В настоящее время её использование ограничено в основном любительскими сферами.

Равномерное алфавитное кодирование

Развитие средств связи и компьютерной обработки данных привело к широкому распространению алфавитных равномерных кодов, то есть таких способов кодирования, в которых все коды, закрепляемые за отдельными знаками первичного алфавита, имеют одну и ту же длину.
Это объясняется более простой технической реализацией *передачи* равномерных кодов. Кроме того, *обработка* равномерных кодов выполняется значительно проще и быстрее чем обработка неравномерных.

В связи с этими особенностями при обычной работе с текстовыми данными, которая занимает значительную долю времени компьютерных систем, применяется *алфавитное равномерное кодирование*.

Исторически первым кодом этой группы считается код, предложенный Э. Бодо в 1870 году. Это равномерный код, в котором для каждого знака первичного алфавита выделяется пять двоичных цифр. Следовательно, первичный алфавит может содержать не более чем $n = 2^5 = 32$ знака.



номер разряда (Т – разряд такта)		5	4	3	Т	2	1
пусто	пусто				•		
3	E				•		●
перевод строки	перевод строки				•	●	
пропуск	пропуск			●	•		
возврат каретки	возврат каретки		●		•		
5	T	●			•		
–	A				•	●	●
8	I			●	•	●	
,	N		●	●	•		
9	O	●	●		•		
'	S			●	•		●
4	R		●		•	●	
10	H	●		●	•		
кто там	D		●		•		●
)	L	●			•	●	
+	Z	●			•		●
7	U .			●	•	●	●
:	C		●	●	•	●	
.	M	●	●	●	•		
[	F		●	●	•		●
]	G	●	●		•	●	
;	J		●		•	●	●
0	P	●		●	•	●	
2	W	●			•	●	●
×	B	●	●		•		●
6	Y	●		●	•		●
(	K		●	●	•	●	●
=	V	●	●	●	•	●	
/	X	●	●	●	•		●
1...	1...	●	●		•	●	●
	Q	●		●	•	●	●

Этот код стал основой для международного телеграфного кода Бодо.

Международный компьютерный код ASCII

Кроме кода Бодо и других кодов, построенных на его основе, в технических устройствах середины XX века применялось много различных равномерных кодов, обладавших разными свойствами и возможностями. Однако по мере развития информационных технологий эти коды постепенно вытеснялись стандартными компьютерными способами кодирования.

К середине XX века в информационных системах разных стран мира использовалось более 60 различных способов кодирования алфавитов естественных языков, так что обмен сообщениями между различными компьютерами был затруднён или вообще невозможен.

В связи с этим остро встала проблема стандартизации способов кодирования текстовых данных. В 1963 году был опубликован первый вариант равномерного алфавитного кода **ASCII** (от англ. American Standard Code for Information Interchange — американский стандартный код для обмена информацией), который стал логическим развитием кода Бодо.

Однако фактическим международным стандартом это код стал только после 1981 года, когда кодировка ASCII была использована для представления текстов в программах для персональных компьютеров, выпущенных фирмой IBM.

В основе выбора системы равномерного кодирования для представления текстов в компьютерах лежали следующие соображения.

Алфавит для представления текстов на естественном языке содержит: 52 латинских буквы, десятичные цифры, знаки препинания, математические знаки, специальные знаки и т.д., всего примерно 150 символов.

Исходя из теоретических соображений, это требует для представления любого знака исходного алфавита при *равномерном алфавитном* двоичном кодировании $\log_2 150 \approx 7{,}2$ знаков, то есть семи или восьми двоичных цифр.

Первоначально кодировка ASCII была семиразрядной, что позволяло выделить коды максимум $2^7 = 128$ знакам. Но впоследствии было предложено использовать для кодирования одного знака группу из 8 двоичных цифр, что позволяет определить коды для $2^8 = 256$ различных знаков первичного алфавита.

Группа из 8 бит, используемая ***как единое целое***, например, для кодирования одного символа текста, называется ***байтом (byte – BinarY TErm)***.

В современном международном стандарте кодовой таблицы ASCII зафиксированы коды только для 128 различных знаков. Их список и соответствующие им восьмиразрядные (то есть состоящие из восьми двоичных цифр, разрядов) двоичные коды образуют **основную (базовую)** часть кодовой таблицы.

Неиспользованные в базовой части коды входят в **расширение таблицы ASCII**. Этими кодами в конкретных разновидностях кодировок можно распоряжаться достаточно произвольно. В частности, в разных странах мира такие расширения применяются для кодирования национальных алфавитов.

Для краткости речи расширения кодовой таблицы обычно называют просто кодовой таблицей (а иногда и **кодовой страницей**), с указанием некоторого дополнительного обозначения, например, говорят: «кодовая таблица Windows 1251», подразумевая под этим соответствующее расширение таблицы ASCII.

Фрагмент ГОСТ альтернативной кодовой таблицы ASCII.

1 знак → 8 бит, 2^8=256 знаков

Знак	2-й код	Знак	2-й код
А	1000 0000	Р	1001 0000
Б	1000 0001	С	1001 0001
В	1000 0010	Т	1001 0010
Г	1000 0011	У	1001 0011
Д	1000 0100	Ф	1001 0100
Е	1000 0101	Х	1001 0101
Ж	1000 0110	Ц	1001 0110
З	1000 0111	Ч	1001 0111
И	1000 1000	Ш	1001 1000
Й	1000 1001	Щ	1001 1001
К	1000 1010	Ъ	1001 1010
Л	1000 1011	Ы	1001 1011
М	1000 1100	Ь	1001 1100
Н	1000 1101	Э	1001 1101
О	1000 1110	Ю	1001 1110
П	1000 1111	Я	1001 1111

$1000\ 1010_2$ → К — ГОСТ альтерн.
→ Љ — Windows1251

10001010 10001110 10001100 10001111 10011100 10011110 10010001 10000101 10010000	
ГОСТ альтернативная	Windows 1251
КОМПЬЮТЕР	ЉЋЊЏњћ’ђ

Универсальная кодировка
UNICODE (UNIversal CODE)
1 символ → 16 бит
2^{16}=65 536

При кодировании текста каждому его знаку ставится в соответствие выбранный из кодовой таблицы конкретный восьмиразрядный двоичный код. Таким образом, для хранения кода каждого знака текста требуется *ровно один байт*.

И, следовательно, текст целиком занимает столько байтов, из скольких знаков он состоит (включая абсолютно все его знаки — пробелы, знаки препинания, специальные знаки перехода на новую строчку, на новую страницу и т. д.).

Понятно, что для представления текстовых данных, а в общем случае для кодирования любых сообщений, возможностей одного байта явно мало. Поэтому группы байтов *объединяются* для размещения в них кода отдельного слова, предложения или всего сообщения целиком.

Группа байт, совместно используемая для представления каких-либо данных, называется ***полем.*** Количество байтов в поле называется ***длиной поля.***

При хранении данных в памяти компьютера код одного знака занимает один байт памяти, и для обозначения необходимой для хранения текстовых данных памяти используются термины «объём данных» и *эквивалентный* термин «**объём памяти**»

Кроме того, когда речь идет о характеристике некоторого поля или участка памяти, используется ещё один термин **длина поля памяти**, которая также измеряется в байтах. Следует понимать, что понятия *объем памяти* и *длина поля памяти* представляют собой одну и ту же характеристику — количество байтов, из которых состоит (которые занимает в памяти) обсуждаемый объект.

Объемные единицы измерения количества информации (система СИ)

Единица	**Значение в байтах**	**Метрический аналог**
1 Кбайт	1024 байт (2^{10})	1 000 (10^3)
1 Мбайт	1024 Кбайт = 1 048 576 байт (2^{20})	1 000 000 (10^6)
1 Гбайт	1024 Мбайт = 1 073 741 824 байт (2^{30})	10^9
1 Тбайт	1024 Гбайт = 1 099 511 697 776 байт (2^{40})	10^{12}
1 Пбайт	1024 Тбайт = 125 899 978 522 624 байт (2^{50})	10^{15}
1 Эбайт	1024 Пбайт = 1 152 921 504 606 846 976 байт (2^{60})	10^{18}
1 Збайт	1024 Эбайт = 1 180 591 620 717 411 303 424 байт (2^{70})	10^{21}
1 Йбайт	1024 Збайт= 1 208 925 819 614 629 174 706 176 байт (2^{80})	10^{24}

Понятие формата

В общем случае ***формат*** понимается как *строго определенный*, *исчерпывающе полный* набор правил.

Конкретный способ кодирования (исчерпывающе полный набор правил) той или иной разновидности информации в компьютере принято называть **форматом данных.**

Текстовый формат определяет одну или несколько кодовых таблиц, которые используются для кодирования символов текста, а также полную совокупность возможностей и правил его оформления.

Некоторые текстовые форматы:

- **TXT** (TeXT — текст) основывается на одной из кодовых таблиц для представления символов и практически не содержит никаких элементов его оформления;
- **RTF** (Rich Text Format — богатый текстовый формат), содержит совокупность стандартных возможностей по оформлению текстов;
- **DOC** (DOCument — документ) содержит подавляющее большинство используемых в современной практике возможностей по оформлению текстов;
- **PDF** (Portable Data Format – переместимый формат данных) универсальный формат, воспринимаемый на компьютерах любого типа

Тема 5. Кодирование числовых данных

Основными разновидностями данных в информатике являются: ***текстовые***, ***числовые***, ***графические***. Представление ***звуковых*** данных базируется на числовых, а ***видеоданных*** — на числовых и графических.

Несмотря на то, что, запись любого числа может рассматриваться как часть текста, рассмотренные способы кодирования текстовых данных принципиально не могут использоваться для кодирования чисел.

Входящие в текст цифры и числа – это обычные знаки алфавита, возможными действиями для которых являются сравнение (совпадает или не совпадает с искомым знаком) и замена одного знака другим (например, для исправления ошибки).

Однако над числами приходится выполнять разнообразные *математические операции* — сложение, вычитание, умножение и т. д.

Для них применяются специальные способы кодирования, которые обеспечивают возможность выполнения указанных операций непосредственно *над кодами* чисел, с получением *математически правильного результата*.

Для обсуждения этих способов потребуются некоторые понятия, связанные с системами счисления.

Системы счисления

Любое число имеет название, значение и изображение, которое представляет собой последовательность знаков, используемую для записи числа.

Пример: различные изображения числа «девять»: IX, 1001, 11, 10, 9, и т.д.

Система счисления представляет собой совокупность правил записи и наименования чисел, а также получения значения чисел из изображающих их символов.

Каждая система счисления использует для изображения чисел свой собственный набор знаков — **алфавит.**

Примеры алфавитов:

Двоичный алфавит: {0,1}

Десятичный алфавит: {0,1,2,3,4,5,6,7,8, 9}

Шестнадцатеричный алфавит: {0,1,2,3,4,5,6,7,8, 9, A, B, C, D, E, F}

Алфавит римской системы счисления: {I,V,X,L,C,D,M}

Количество знаков в алфавите системы счисления обычно отражается в ее названии: двоичная, троичная, восьмеричная, десятичная, шестнадцатеричная и т. д. Вообще говоря, можно рассматривать системы счисления с *любым натуральным* количеством знаков в алфавите.

Система счисления всего с одним знаком в алфавите называется *унарной*. Её свойства и правила использования существенно отличаются от всех остальных систем

Знаки, входящие в алфавиты систем счисления традиционно называются цифрами. Таким образом, знаки A, B, C, D, E и F, входящие в алфавит шестнадцатеричной системы счисления, считаются цифрами.
В общем случае различают **непозиционные**, **позиционные** и **смешанные** системы счисления.

В позиционных системах значение, которое отдельная цифра вкладывает в общее значение числа, *зависит* от ее положения (*позиции*) в ряду других цифр, изображающих число, а в непозиционных – не зависит.

В компьютерах для представления числовых данных применяются только позиционные системы счисления, в которых используется *единый* закон получения значения числа из изображающих его цифр.

Закон разложения по степеням основания системы счисления.

Пусть $A = \{\alpha_1, \alpha_2, ..., \alpha_p\}$ — алфавит некоторой позиционной системы счисления

$$x = a_n a_{n-1} ... a_1 a_0 , a_{-1} a_{-2} ... a_{-m} \qquad a_i \in A \qquad -m \le i \le n$$

При использовании позиционных систем счисления позиции цифр в числе принято называть **разрядами**: разряд единиц, разряд десятков, разряд сотен и т.д.

Нумерация разрядов для целой части начинается с нуля и производится с возрастанием номеров *справа налево*, а нумерация для дробной части начинается с минус единицы и осуществляется в порядке убывания номеров *слева направо*.

Каждому разряду соответствует определенный зависящий от используемой системы счисления **весовой коэффициент**, на который умножается значение находящейся в этом разряде цифры. Это коэффициент в десятичной системе равен числу десять, *которое возведено в равную номеру разряда степень.*

Таким образом, для разряда единиц этот коэффициент равен единице, для разряда десятков — десяти, для разряда сотен — ста и т.д. Значение числа получается суммированием вкладов от каждой цифры, умноженной на соответствующий ее разряду весовой коэффициент.

$$\| x \| = \sum_{i=-m}^{n} a_i p^i = a_{-m} p^{-m} + a_{-m+1} p^{-m+1} + \ldots + a_{-1} p^{-1} + a_0 p^0 + a_1 p^1 + \ldots + a_n p^n$$

,

Пример: x=52,128 $\| x \| = 5 \times 10^1 + 2 \times 10^0 + 1 \times 10^{-1} + 2 \times 10^{-2} + 8 \times 10^{-3}$

$$= 5 \times 10 + 2 + \frac{1}{10} + \frac{2}{100} + \frac{8}{1000}$$

Целое число *p*, входящее в выражение, служащее для определения значения числа, называется **основанием системы счисления**. Это число *совпадает* с количеством цифр в алфавите позиционной системы счисления.

Все входящие в выражение весовые коэффициенты равны p^i — основанию системы счисления *p* в степени, равной номеру разряда *i*.

Общий принцип изображения целых чисел в позиционных системах счисления

- Процесс нумерации, то есть обозначения чисел в любой позиционной сис-теме счисления начинается с того, что за числом «нуль» закрепляется младшая цифра алфавита (цифра 0).
- При переходе к очередному целому числу в текущий разряд записывается *следующая* цифра алфавита: после 0→1, после 1→2, и т.д..
- После исчерпания всех возможностей по перебору цифр алфавита *во всех уже используемых разрядах* в них записывается младшая цифра 0, а в изображение числа слева добавляется еще один разряд, в который записывается вторая цифра алфавита — цифра «1». Например, в десятичной системе после 9 следует 10, после 99 — 100, после 999 следует 1000 и т.д. После чего возобновляется перебор цифр алфавита с самого *младшего* разряда числа: 100, 101, 102,… .
- После завершения перебора всех цифр алфавита в последнем или в одном из внутренних разрядов числа и во всех предшествующих ему, в нём и во всех предыдущих разрядах (если они есть) записывается младшая цифра алфавита 0 и происходит переход к следующей цифре в следующем старшем разряде. Например, в той же десятичной системе после 19 следует 20, после 119— 120, после 1199 — 1200 и т.д. Затем перебор цифр алфавита возобновляется с самого *младшего* разряда.

Максимальное значение *n* - разрядных *целых* чисел в системе счисления с основанием *p* равно $p^n - 1$. Например, максимально возможное 4-х *разрядное* десятичное число 9999 равно $10^4 - 1$.

В частности, максимально возможное *n* – разрядное целое двоичное число равно $2^n - 1$.

Двоичная система счисления

Число	Запись в двоичной системе счисления
0	0
1	1
2	10
3	11
4	**100**
5	101
6	110
7	111
8	**1000**
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Число	Запись в двоичной системе счисления
16	**10000**
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111

Запись целого неотрицательного десятичного числа в двоичной системе счисления называется **прямым двоичным кодом** этого числа.

Шестнадцатеричная система счисления

Число	Запись в 16-й системе счисления
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Число	Запись в 16-й системе счисления
16	**10**
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F

Шестнадцатеричная система счисления

Число	Запись в 16-й системе счисления
32	20
33	21
…	…
41	29
42	2A
43	2B
44	2C
45	2D
46	2E
47	2F
48	30
49	31
…	…
159	9F
160	A0
161	A1

Число	Запись в 16-й системе счисления
162	A2
…	…
169	A9
170	AA
171	AB
172	AC
173	AD
174	AE
175	AF
176	B0
177	B1
…	…
255	FF
256	**100**
257	101
…	…

Переходы между системами счисления

Перевод целых чисел из десятичной системы в систему с основанием P

1. Последовательно делить вначале заданное число, а затем полученные *частные* на основание *P*. Процесс деления продолжается до тех пор, пока очередное частное не окажется меньше *P*.
2. Последнее частное и остатки от делений записываются цифрами системы счисления с основанием *P* в порядке противоположном их получению.

Пример перевода в двоичную систему счисления

2005 → 1002 → 501 → 250 → 125 → 62 → 31 → 15 → 7 → 3 → 1

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

1 0 1 0 1 0 1 1 1 1

$2005_{10} \rightarrow 11111010101_2$

Пример перевода в шестнадцатеричную систему счисления

2005	16
16	125
40	112
32	13
85	
80	
5	

$7_{10} \rightarrow 7_{16}$ $13_{10} \rightarrow D_{16}$ $2005_{10} \rightarrow 7D5_{16}$

Перевод правильных дробей из десятичной системы в систему с основанием P.

1. Последовательно умножать вначале заданную дробь, а затем *дробные части произведений* на основание *P*. Процесс умножения продолжается до получения нулевой дробной части либо до достижения желаемой точности результата.
2. Целые части произведений записываются цифрами системы счисления с основанием *P* в порядке их получения как искомая дробная часть результата.

× 0,625 | 2	× 0,625 | 16
× 1,250 | 2	10,000
× 0,500 | 2	$10_{10} \rightarrow A_{16}$
1,000	
$0{,}625_{10} \rightarrow 0{,}101_{2}$	$0{,}625_{10} \rightarrow 0{,}A_{16}$

В теории погрешностей вычислений устанавливается, что погрешность вещественных чисел, полученных в результате округления, не превосходит половины значения последнего разряда.

Например, число 23,176 может быть получено как результат округления следующих чисел:

- 23,17648 – погрешность равна: Δ=|23,176-23,17648|=0,00048
- 23,17589 – погрешность равна: Δ=|23,176-23,17589|=0,00011
- 23,1757 – погрешность равна: Δ=|23,176-23,1757|=0,0003
- 23,17629 – погрешность равна: Δ=|23,176-23,17629|=0,00029

Отсюда $\Delta \le 0{,}0005$ или $\Delta \le 1/2 \times 10^{-3}$.

Пусть правильная дробь в системе счисления с основанием *q* содержит *n* цифр. Сколько цифр *m* в дробной части этого числа следует получить при переходе в систему счисления с основанием *p?*

$$\Delta_q \le q^{-n}/2; \quad \Delta_p \le p^{-m}/2; \quad \Delta_p \le \Delta_q; \quad p^{-m} \le q^{-n}; \; m \ge n \log_p q;$$

При переходе из десятичной в двоичную (10→2):

q=10; p=2; $m = n \times \log_2 10$, $m = 3{,}329 \times n$

Перевод смешанных чисел из десятичной системы в систему с основанием P.

Нужно осуществить раздельный перевод целой и дробной частей, а затем сложить полученные результаты.

Пример: $2005{,}625_{10} \rightarrow 111\ 1101\ 0101{,}101_2$ $2005{,}625_{10} \rightarrow 7D5{,}A_{16}$.

Перевод чисел из системы с основанием P в десятичную систему счисления

1. Записать число в виде разложения по степеням основания исходной системы счисления.
2. Заменить *p*-е цифры числа их десятичными эквивалентами.
3. Выполнить вошедшие в полученное выражение действия.

Пример: $111\ 1101\ 0101{,}101_2$; $p=2$

$$1\times2^0+0\times2^1+1\times2^2+0\times2^3+1\times2^4+0\times2^5+1\times2^6+1\times2^7+1\times2^8+1\times2^9+1\times2^{10}+1\times2^{-1}+0\times2^{-2}+1\times2^{-3}$$

$$1+2^2+2^4+2^6+2^7+2^8+2^9+2^{10}+2^{-1}+2^{-3}$$

$$=1+4+16+64+128+256+512+1024+1/2+1/8=2005{,}625_{10}$$

Пример: $7D5{,}A_{16}$; $p=16$

$$5\times16^0+D\times16^1+7\times16^2+A\times16^{-1}=5+13\times16+7\times256+10/16=2005{,}625_{10}$$

Переходы между двоичной и шестнадцатеричной системами счисления

$p_1=2$, $p_2=16$, $p_2=p_1^4$

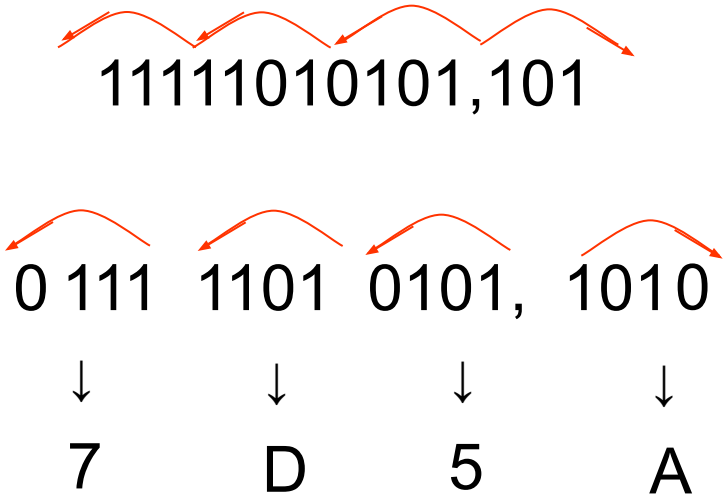
Каждую цифру алфавита шестнадцатеричной системы счисления представляют четверкой (обычно говорят **тетрадой**) двоичных цифр

16-я цифра	**Тетрада**	**16-я цифра**	**Тетрада**	**16-я цифра**	**Тетрада**	**16-я цифра**	**Тетрада**
0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

Переход из двоичной системы счисления в шестнадцатеричную

1. Разбить число на тетрады. Разбиение производится от запятой для целой части числа справа налево, а для дробной — слева направо.
2. Если в процессе разбиения на концах числа образуются неполные тетрады, их следует дополнить незначащими нулями.
3. Каждую тетраду заменить соответствующей шестнадцатеричной цифрой.

Пример: $11111010101{,}101_2$



$11111010101{,}101_2 \rightarrow 7D5{,}A_{16}$

Переход из шестнадцатеричной системы счисления в двоичную

Чтобы перевести шестнадцатеричное число в двоичную систему счисления нужно каждую шестнадцатеричную цифру числа заменить двоичной четверкой.

Пример: $7D5,A_{16}$

7	D	5,	A
↓	↓	↓	↓
0111	1101	0101	1010

$$011111010101,1010 \rightarrow 11111010101,101$$

$$7D5,A_{16} \rightarrow 11111010101,101_2$$

Таблица сложения двоичной системы счисления

0+0=0
1+0=1
0+1=1
1+1=(1)0

Сложение *многоразрядных* чисел производится также как и в десятичной системе: осуществляется поразрядное сложение с учетом возможных переносов из предыдущих разрядов. Разряды при этом перебираются *справа налево*.

Пример:

$$\begin{array}{r} 101001011011{,}0011_2 \\ +\ 111101010111{,}1111_2 \\ \hline 1100110110011{,}0010_2 \end{array}$$

Перенос (в «уме»): 1

Одноразрядное действие: 1+1 = 0 и 1 перенос (в «уме»)

Таблица вычитания двоичной системы счисления

$$0 - 0 = 0$$
$$1 - 0 = 1$$
$$1 - 1 = 0$$
$$10 - 1 = 1$$

Вычитание многоразрядных двоичных чисел производится также как и десятичных: осуществляется поразрядное вычитание с учетом возможности займа из старшего разряда. Если же уменьшаемое меньше вычитаемого и для старшего разряда выполнить заем неоткуда, то уменьшаемое и вычитаемое меняются местами, а к результату приписывается знак минус.

Пример:

$$\begin{array}{r} 1100110110011,0010_2 \\ -\ 101001011011,0011_2 \\ \hline 111101010111,11\ 11_2 \end{array}$$

заем

Одноразрядное действие: 0 – 1= 10 – 1 = 1

Таблица умножения двоичной системы счисления

$$0 \times 0 = 0$$
$$1 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 1 = 1$$

Умножение многоразрядных двоичных чисел производится по той же схеме, что и умножение десятичных:

1. первый сомножитель поразрядно умножается на каждую из цифр второго сомножителя;
2. каждое следующее произведение смещается влево на один разряд по отношению к предыдущему;
3. результаты всех умножений складываются.

Пример:

$$\begin{array}{r} \times\ 11011{,}11_2 \\ \quad 101{,}01_2 \\ \hline 1101111 \\ \quad 1101111 \\ 1101111 \\ \hline 10010001{,}1011_2 \end{array}$$

Фрагмент таблицы сложения шестнадцатеричной системы счисления

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
A	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

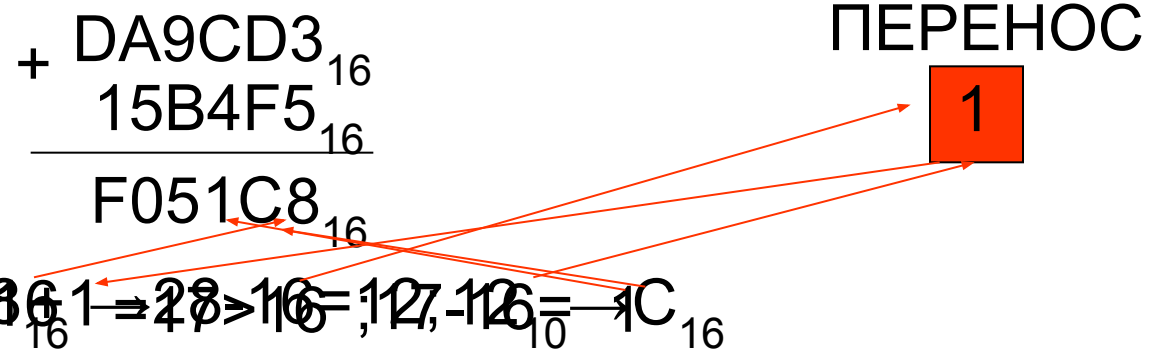
Сложение двух многозначных шестнадцатеричных чисел

1. Складываемые шестнадцатеричные цифры заменяются десятичными эквивалентами.
2. Выполняется сложение в десятичной системе счисления.
3. Если полученная сумма меньше 16-ти, то она считается результатом сложения в текущем разряде и записывается в виде шестнадцатеричного эквивалента.
4. Если результат сложения больше или равен 16-ти, то из суммы вычитается 16 и запоминается единица переноса в следующий разряд, а полученная разность записывается в виде шестнадцатеричной цифры как результат сложения в текущем разряде.

Пример:

$$\begin{array}{r} DA9CD3_{16} \\ +\ 15B4F5_{16} \\ \hline F051C8_{16} \end{array}$$

ПЕРЕНОС



Второй способ сложения многозначных шестнадцатеричных чисел

1. Сумма цифр *оценивается* по отношению к шестнадцатеричному десятку.
2. Если она *меньше*, чем 16, то выполняется прямое сложение и результат записывается виде шестнадцатеричной цифры.
3. Если сумма *не меньше* 16, то из двух складываемых цифр выбирается большая и определяется, сколько единиц ей не хватает до десятка (шестнадцатеричного).
4. Нужное количество единиц вычитается из меньшей цифры пары.
5. Разность записывается в шестнадцатеричном виде как результат сложения, при этом запоминается единица переноса в следующий разряд.

Пример:

$$\begin{array}{r} + \; DA9CD3_{16} \\ 15B4F5_{16} \\ \hline F051C8_{16} \end{array}$$

ПЕРЕНОС

1

$3+5=8<16$, $8_{10} \rightarrow 8_{16}$

D<F; F ближе к 10_{16}, не хватает 1; D-1=C;

Вычитание двух многозначных шестнадцатеричных чисел

1. Участвующие в вычитании цифры текущего разряда заменяются их десятичными эквивалентами.
2. Если уменьшаемое меньше вычитаемого, то из ближайшего слева ненулевого разряда занимается шестнадцатеричный десяток, и к уменьшаемому прибавляется 16.
3. Производится вычитание в десятичной системе.
4. Результат записывается в виде шестнадцатеричной цифры.
5. При переходе к следующим разрядам необходимо учитывать выполненные ранее займы.

Пример:

$$\begin{array}{r} F050C8_{16} \\ -\ DA9CD3_{16} \\ \hline 15B4F5_{16} \end{array}$$

заем 1

8-13=5 12-13=12+16-13=15$_{10}$ → F$_{16}$

Форматы кодирования числовых данных

Вспомним, что исчерпывающе полный набор правил кодирования той или иной разновидности данных принято называть *форматом этих данных*.

Для представления числовых данных в компьютерах используются два основных формата: формат с **фиксированной точкой** (запятой) и **формат с плавающей точкой** (запятой). В названиях форматов речь идет о знаке, с помощью которого целая часть числа оделяется от его дробной части.

Формат с фиксированной точкой предназначен для *абсолютно точного* представления *целых* чисел, которые появляются в задачах в результате выполнения операций типа пересчёт, например, подсчёт количества студентов, присутствующих на занятии. Результаты того рода операций всегда абсолютно точные целые числа. В языках программирования такие числа относятся к *целому типу*.

Формат с плавающей точкой используется для представления *нецелых*, точнее *приближенных* чисел. Такие числа возникают в задачах в результате различных измерений (например, измерений веса некоторого тела или его длины), которые, как известно, всегда выполняются с некоторой погрешностью, приближенно. В языках программирования их относят к *вещественному типу*.

Возможности одного байта для кодирования чисел достаточно малы, поэтому числа обычно занимают несколько соседних байтов, то есть *поле*, длина которого зависит от используемого формата и является важной его характеристикой.

Формат с фиксированной точкой

Формат с ***фиксированной точкой*** предназначен для представления ***целых*** чисел. Целые числа в этом формате представлены абсолютно точно. Используются поля длиной 1,2 и 4 байта. Существуют ***беззнаковое*** и ***знаковое*** представления формата.

Беззнаковое представление формата с фиксированной точкой

Используется прямой двоичный код, все биты поля содержат значащие цифры числа.

Точка, отделяющая целую часть числа от дробной, считается расположенной, *зафиксированной* справа от крайнего правого разряда.

Следовательно, под дробную часть числа отводится *нулевое* количество разрядов, поэтому в данном варианте кодирования возможна работа только с целыми числами. Постоянное расположение точки, фиксация её позиции дала название формату — «с фиксированной точкой».

Диапазон представимых чисел: от 0 до $2^N - 1$, N – длина поля в битах.

Название	Длина	Диапазон	
byte	1 байт	0÷255	$0 \div 2^8-1$
word	2 байта	0÷65 535	$0 \div 2^{16}-1$
*	4 байта	0÷4 294 967 295	$0 \div 2^{32}-1$

Порядок перехода к беззнаковому формату с фиксированной точкой:

1. перевести заданное целое положительное число в двоичную систему счисления;
2. выбрать поле, длина которого соответствует значению числа;
3. записать полученный код в это поле, при необходимости дополняя его слева незначащими нулями.

Примеры: **77_{10}→ 4D; 004D; 0000004D; 267_{10} → 010B; 0000010B**

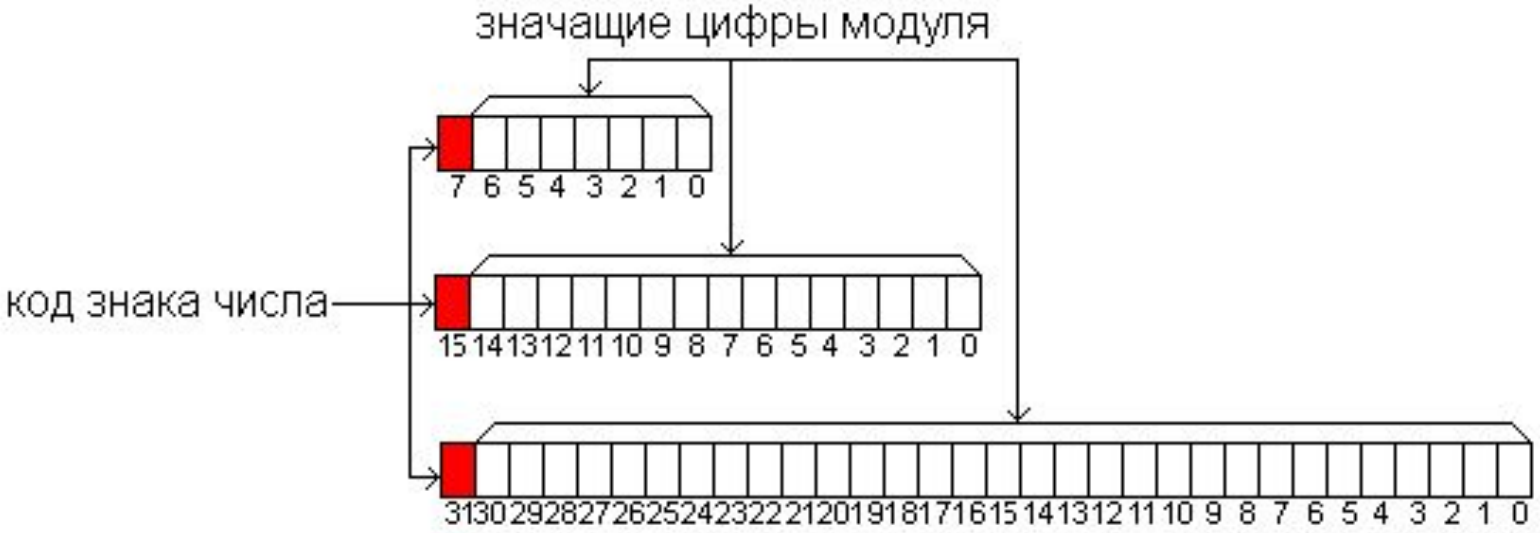
Определение числа по его коду (беззнаковый, фиксированная точка):

Чтобы определить значение числа, для которого задан его код в беззнаковом представлении формата с фиксированной точкой, достаточно, считая весь этот код двоичным (или шестнадцатеричным) *числом*, перевести его в десятичную систему счисления.

Знаковое представление формата с фиксированной точкой.
Система кодирования со знаком

Один бит поля выделяется под код знака: 0 — код знака +, 1 — код знака -; остальные биты поля — значащие цифры модуля, диапазон модулей представимых чисел: от 0 до 2^{n-1} -1, n — длина поля в битах.

Код знака числа принято размещать *в самом левом* разряде поля, который в связи с этим принято называть **знаковым битом**.



В знаковом представлении *различные* биты кода играют *разные* роли. Для однозначного выполнения кодирования и декодирования данных для каждого способа кодирования необходимо чётко определить и точно описать роль каждого бита поля. Описание закреплённых за разрядами поля конкретных функций по хранению различных элементов машинного кода принято называть структурой **разрядной сетки** формата или системы кодирования**.**

Попытка

использования системы кодирования со знаком (прямого двоичного кода со знаковым битом) для представления знаковых целых чисел:

$$|4|_{10} \rightarrow 000\ 0100_2; \quad +4 \rightarrow 0|000\ 0100_2; \quad -4 \rightarrow 1|000\ 0100_2;$$

$$|127|_{10} \rightarrow 111\ 1111_2; \quad +127 \rightarrow 0|111\ 1111_2; \quad -127 \rightarrow 1|111\ 1111_2;$$

Недостатки:

1. Неоднозначное представление нуля: $+0 \rightarrow 0|000\ 0000_2$; $-0 \rightarrow 1|000\ 000_2$;
2. Специальная арифметика:

$$\begin{array}{rl} & 0000\ 0001 \quad (+1) \\ + & 1000\ 0001 \quad (-1) \\ \hline & 1000\ 0010 \quad (-2) \end{array}$$

Дополнительный код целых чисел

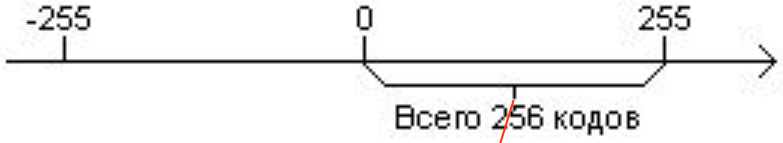
Код должен подчиняться правилам выполнения сложения и вычитания в двоичной системе: код следующего положительного числа получается прибавлением единицы, а код следующего отрицательного числа получается вычитанием единицы из кода текущего числа. Построенный таким образом код принято называть ***дополнительным***.

При четырех битах всего 16 кодов, диапазон при использовании прямого кода: от – 7 до +7, два кода заняты под нуль 0000_2 и 1000_2

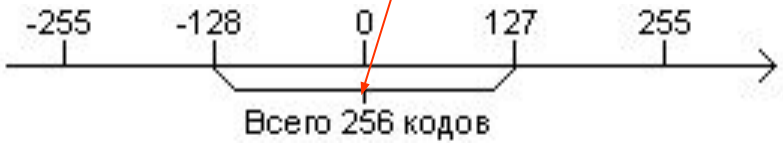
При четырех битах всего 16 кодов, диапазон при использовании дополнительного кода: от – 8 до +7, под нуль занят один код 0000_2, а код 1000_2 обозначает -8

+7 → 0111_2
+6 → 0110_2
+5 → 0101_2
+4 → 0100_2
+3 → 0011_2
+2 → 0010_2
+1 → 0001_2
+0 → 0000_2
-1 → 1111_2
- 2 → 1110_2
- 3 → 1101_2
- 4 → 1100_2
- 5 → 1011_2
- 6 → 1010_2
- 7 → 1001_2
- 8 → 1000_2

Для однобайтового поля всего 256 кодов, диапазон целых чисел для беззнакового представления



Для однобайтового кода всего 256 кодов, диапазон целых чисел для однобайтового поля знакового представления



Пусть $M = 2^n$, где n — разрядность поля, используемого для записи кода. С формальных позиций дополнительный код D отрицательного целого числа представляет собой *дополнение* прямого кода модуля P этого числа до числа n: $P+D=M$, откуда $D=M-P$. Отметим, что название «дополнительный» код возникло именно по этой причине.

Рассмотрим пример. Вновь возьмём $n=4$ и найдем код числа минус семь. В данном случае $M=2^4=10000_2$, прямой код модуля обсуждаемого числа $P=111_2$, и, следовательно, его дополнение до M равно $D = 10000_2 – 111_2 = 1001_2$, что совпадает с полученным прямыми рассуждениями дополнительным кодом этого числа.

Способ кодирования знаковых чисел, основанный на использовании *дополнительного* кода, устраняет все отмеченные выше недостатки применения системы кодирования со знаком.

Диапазон представления чисел при использовании дополнительного кода:
от -2^{N-1} до $2^{N-1}-1$

Название	Длина	Диапазон	
shortint	1	-128÷127	$-2^7 \div 2^7-1$
iInteger	2	-32768÷32767	$-2^{15} \div 2^{15}-1$
longint	4	-2 147 483 648÷2 147 483 647	$-2^{31} \div 2^{31}-1$

Порядок перехода к знаковому формату с фиксированной точкой:

1. Перевести модуль целого числа в двоичную систему счисления;
2. Если число отрицательное, то получить дополнительный код, инвертировав прямой код (получится обратный код) и добавив к результату 1;
3. Записать число в выделенное для него поле.

Примеры: $+77_{10} \rightarrow$ **0100 1101$_2$ или 4D$_{16}$;**

-77; 1) прямой код модуля: 77 $\rightarrow$ 4D$_{16}$= 0100 1101$_2$;
2) обратный код: 1011 0010$_2$;
дополнительный код: 1011 0011$_2$;
3) – 77 $\rightarrow$ 1011 0011$_2$ или B3$_{16}$;

Способы получения дополнительного кода

$$\begin{array}{r} 10000\ 0000_2 \\ -\ 0100\ 1101_2 \\ \hline 1011\ 0011_2 \end{array} \qquad \begin{array}{r} 100_{16} \\ -\ 4D_{16} \\ \hline B3_{16} \end{array}$$

Правило размножения знака при переходе к более длинным полям

Число: $+77_{10}$

Поле 1 байт: $0|100\ 1101_2$ или $4D_{16}$;

Поле 2 байта: $0|000\ 0000\ 0100\ 1101_2$ или $00\ 4D_{16}$;

Поле 4 байта: $0|000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0100\ 1101_2$ или $00\ 00\ 00\ 4D_{16}$

Дополнительные коды для -77

$$\begin{array}{r} 1\ 0000\ 0000\ 0000\ 0000_2 \\ -\quad 0100\ 1101_2 \\ \hline 1111\ 1111\ 1011\ 0011_2 \end{array}$$

$$\begin{array}{r} 1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 \\ -\quad 0100\ 1101_2 \\ \hline 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011\ 0011_2 \end{array}$$

$$\begin{array}{r} 1\ 00\ 00_{16} \\ -\quad 4D_{16} \\ \hline FF\ B3_{16} \end{array}$$

$$\begin{array}{r} 1\ 00\ 00\ 00\ 00_{16} \\ -\quad 4D_{16} \\ \hline FF\ FF\ FF\ B3_{16} \end{array}$$

Число: -77_{10}

Поле 1 байт: $1|011\ 0011_2$ или $B3_{16}$;

Поле 2 байта: $1|111\ 1111\ \ 1011\ 0011_2$ или $FF\ B3_{16}$;

Поле 4 байта: $1|111\ 1111\ 1111\ 1111\ 1111\ 1111\ \ 1011\ \ 0011_2$ или $FF\ FF\ FF\ B3_{16}$

При переходе к более длинному полю свободные слева биты поля заполняются кодом знакового бита 0_2 (0_{16}) для положительных и 1_2 (F_{16}) – для отрицательных

Порядок определения числа по его коду (знаковый, фиксированная точка):

1. По знаковому биту определить знак числа. Для отрицательных чисел в 2-ой системе счисления крайняя слева цифра равна 1_2, в 16-й ─ эта цифра не меньше (≥) 8_{16};
2. Если код дополнительный, то перейти к прямому коду модуля (по тем же правилам, что и переход от прямого к дополнительному);
3. Перевести число из 2-ой (или 16-ой) системы счисления в десятичную
4. Сформировать соответствующий знак числа.

Примеры:

$005A_{16}$ → $0000\ 0000\ 0101\ 1010_2$; >0; → +90

$FFE8_{16}$ → $1111\ 1111\ 1110\ 1000_2$; <0; 10000-FFE8=0018 → - 24

Нормализованные числа.

В естественных науках часто используются числа вида:- $2{,}9\times10^{18}$;$0{,}091\times10^{-30}$. Обычно этот способ используется для задания очень больших или очень маленьких по модулю чисел.

Например:

$-\ 2\ 900\ 000\ 000\ 000\ 000\ 000 \rightarrow -\ 2{,}9\times10^{18}$

$0{,}000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 091 \rightarrow 0{,}091\times10^{-30}$

Запись чисел таким способом обладает неоднозначностью:

$18{,}456 \rightarrow 1{,}8456\times10^{+1} \rightarrow 0{,}18456\times10^{+2} \rightarrow 184{,}56\times10^{-1} \rightarrow 1845{,}6\times10^{-2}$

Чтобы избавится от неоднозначности на значение мантиссы **m** накладывается какое-либо ограничение, например: **$0{,}1 \le m < 1$** или **$1 \le m < 10$**.

Числа, записанные в форме **$\pm\ m \times q^{\pm p}$**, где **$1 \le m < q^{+1}$** — мантисса числа, **q** — основание системы счисления, **p** — порядок, называются ***нормализованными***: Для десятичной системы счисления это ограничение имеет вид **$1 \le m < 10$**. Примеры нормализованных чисел: **$-\ 2{,}9\times10^{18}$**; **$1{,}8456\times10^{+1}$; $9{,}1\times10^{-32}$**

ФОРМАТ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Формат с плавающей точкой используется для кодирования нецелых чисел, обычно возникающих в результате выполнения всевозможных измерений. В информатике такие числа принято называть ***вещественными***. Числа в этом формате представлены принципиально неточно, то есть приближенно. **5.0 — вещественное, нецелое, приближенное ≠ 5 — целое, точное**.

Принцип кодирования в формате с плавающей точкой

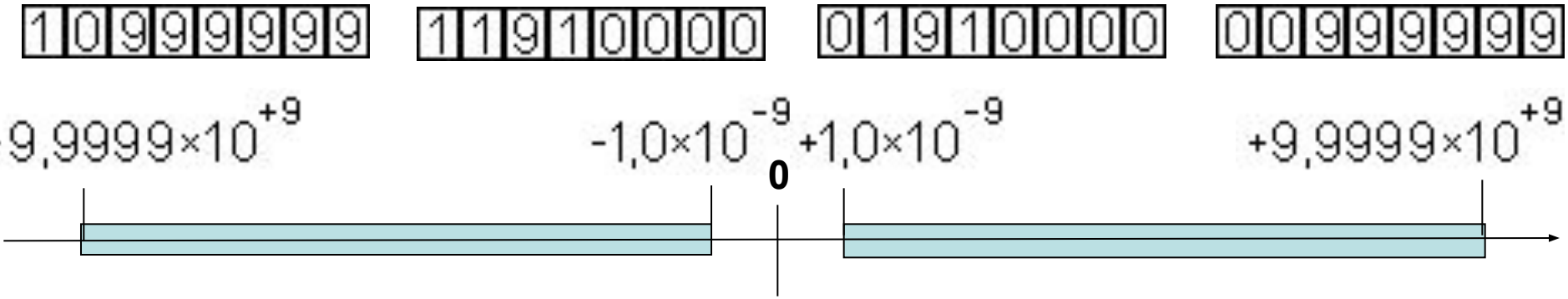
Код числа вида $\pm m \times q^{\pm p}$ должен содержать:

1. Код знака числа ± (0 или 1);
2. Код нормализованной мантиссы $1 \le m < 10$;
3. Код знака порядка ± (0 или 1);
4. Код порядка p.

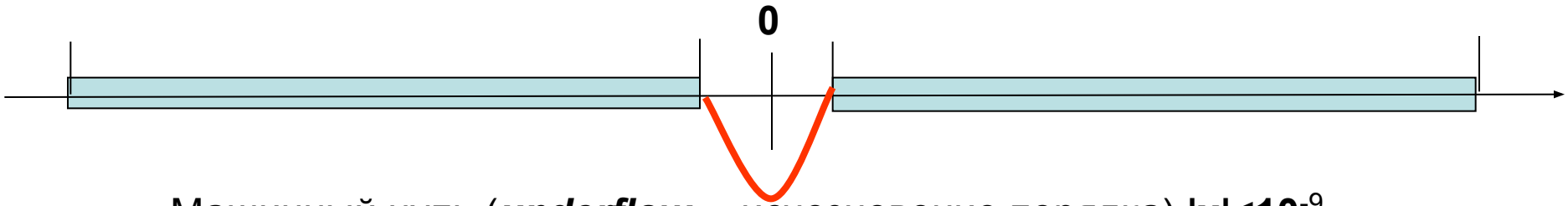
Возможный способ кодирования с помощью восьми десятичных цифр

0 0 5 3 4 7 0 8	$+3{,}4708\times10^{+5}$
0 1 5 3 4 7 0 8	$+3{,}4708\times10^{-5}$
1 0 5 3 4 7 0 8	$-3{,}4708\times10^{+5}$
1 1 5 3 4 7 0 8	$-3{,}4708\times10^{-5}$

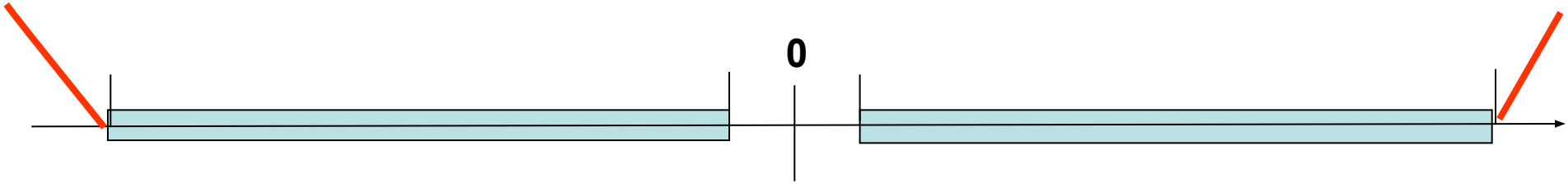
Диапазон представления чисел с плавающей точкой



$1{,}0\times10^{-9} \le |X| \le 9{,}9999\times10^{+9}$ и 0

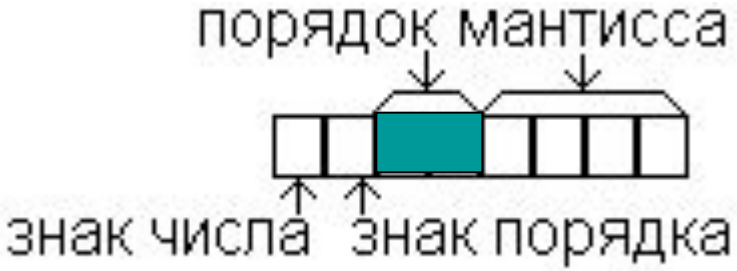


Машинный нуль (***underflow*** — исчезновение порядка) **$|x|<10^{-9}$**



Переполнение (***overflow*** — переполнение порядка) **$|x| > 9{,}9999\times10^{+9}$**

Влияние распределения ролей в разрядной сетке на диапазон



Минимальное по модулю число:

$+1{,}0\times10^{-9}$

$+1{,}0\times10^{-99}$

Максимальное по модулю число:

$+9{,}9999\times10^{+9}$

$+9{,}999\times10^{+99}$

При изменении количество разрядов, отводимых на порядок, изменяется диапазон представимых чисел. При изменении количества разрядов, отводимых на мантиссу, изменяется точность представления чисел

Денормализация и нормализация при выполнении арифметических операций над данными в формате с плавающей точкой.

Нормализация результата

$$+\begin{array}{l}1{,}6\times10^{-4}\\ 1{,}2\times10^{-4}\\ \hline 2{,}8\times10^{-4}\end{array}$$

$$+\begin{array}{l}8{,}6\times10^{-4}\\ 3{,}7\times10^{-4}\\ \hline 12{,}3\times10^{-4}\end{array}\rightarrow 1{,}23\times10^{-3}$$

$$-\begin{array}{l}1{,}6\times10^{-4}\\ 1{,}52\times10^{-4}\\ \hline 0{,}08\times10^{-4}\end{array}\rightarrow 8{,}0\times10^{-6}$$

Предварительная денормализация операнда

$$+\begin{array}{l}1{,}6\times10^{-4}\\ 1{,}5\times10^{-5}\end{array}\rightarrow +\begin{array}{l}1{,}6\times10^{-4}\\ 0{,}15\times10^{-4}\\ \hline 1{,}75\times10^{-4}\end{array}$$

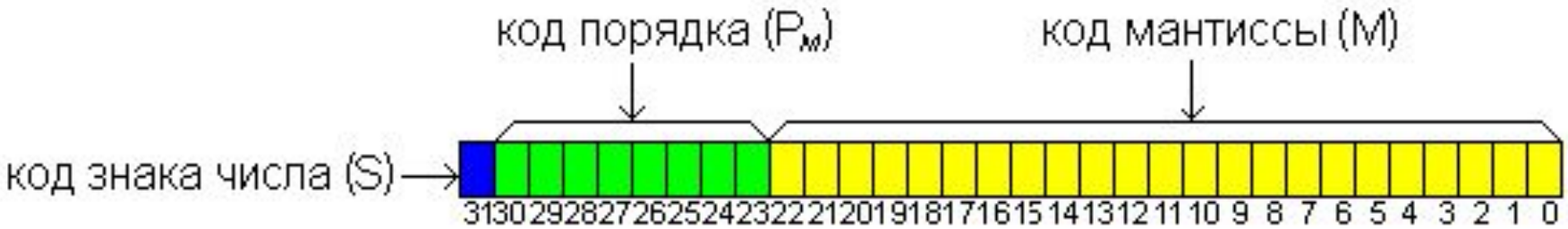
$$+\begin{array}{l}1{,}6\times10^{-4}\\ 1{,}5\times10^{-3}\end{array}\rightarrow +\begin{array}{l}1{,}6\times10^{-4}\\ 15{,}0\times10^{-4}\\ \hline 16{,}6\times10^{-4}\end{array}\rightarrow 1{,}66\times10^{-3}$$

Диапазон и точность представления чисел в формате с плавающей точной

Двоичные коды чисел в формате с плавающей точкой могут занимать поля длиной 4, 6, 8 и 10 байт.

Название	Длина	Диапазон	Точность
single	4	$1{,}5\times10^{-39}\div3{,}4\times10^{+38}$	7÷8
real	6	$2{,}9\times10^{-39}\div1{,}7\times10^{+38}$	11÷12
double	8	$5{,}0\times10^{-324}\div1{,}7\times10^{+308}$	15÷16
extended	10	$3{,}6\times10^{-4951}\div1{,}1\times10^{+4932}$	19÷20

Поля длиной 4 байта — короткие вещественные (одинарная точность)



Код знака числа S — 1 бит (31 бит)

Код машинного порядка $P_м$ — 8 бит (23-30 биты)

Код мантиссы M — 23 бита (0-22 биты).

S=0 для положительных чисел и S=1 — для отрицательных

Код машинного порядка $P_м$ равен порядку числа $P_и$ увеличенному на 127_{10}

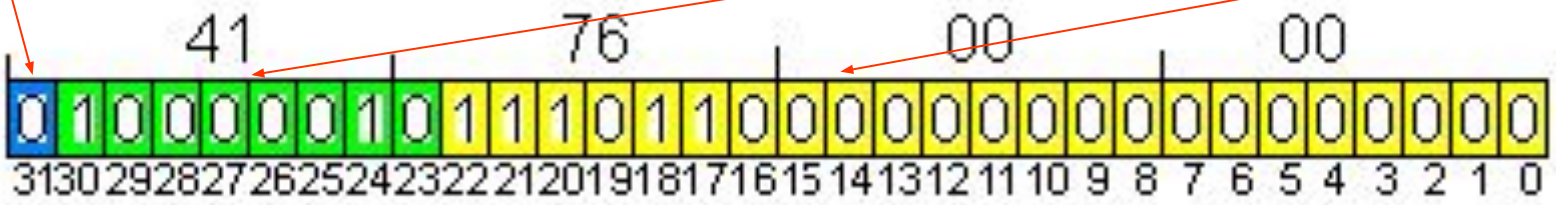
Код мантиссы равен мантиссе, уменьшенной на единицу

Порядок получения кода числа в формате с плавающей точкой

1. Перевести модуль числа из десятичной системы счисления в двоичную.
2. Нормализовать число.
3. Сформировать знаковый бит S (0 → код знака плюс ; 1 → код знака минус).
4. Выделить истинный порядок ($P_и$).
5. Получить код машинного порядка ($P_м = P_и + 127_{10}$ или $7F_{16}$ или $111\ 1111_2$).
6. Получить код мантиссы M, отбрасывая бит целой части.
7. Записать коды S, $P_м$, M в отведенные для них позиции.

Пример: **$+15{,}375_{10}$ → $1111{,}011_2$ → $1{,}111011 \times 10^{11}{}_2$**

S=0; $P_и = 11_2$; $P_м = P_и + 111\ 1111_2 = 1000\ 0010_2$; $M = 111011_2$



Код числа **$+15{,}375_{10}$** в формате с плавающей точкой в четырехбайтовом поле
$0100\ 0001\ 0111\ 0110\ 0000\ 0000\ 0000\ 0000_2$ или **$41\ 76\ 00\ 00_{16}$**

Порядок получения числа по его коду в формате с плавающей точкой

1. Получить запись числа в двоичной системе счисления.
2. Выделить код знака S и определить знак числа.
3. Выделить код машинного порядка $P_м$.
4. Получить истинный порядок $P_и$ ($P_и = P_м - 127_{10}$ или $7F_{16}$ или $111\ 1111_2$).
5. Выделить код мантиссы M и получить мантиссу, дописывая бит целой части.
6. Получить число в нормализованной форме.
7. При необходимости денормализовать число, перевести его в десятичную систему счисления и дописать нужный знак.

Пример: **$C2B7AE00_{16}$** **$\rightarrow$1100 0010 1011 0111 1010 1110 0000 0000**

$\rightarrow$1 10000101 01101111010111 S=1; $P_м = 10000101_2 = 85_{16}$;

$\rightarrow P_и = (85-7F)_{16} = 6_{16} = 110_2$ **M = 01101111010111**

$\rightarrow -1{,}01101111010111 \times 10^{+110} \rightarrow -1011011{,}11010111_2 \rightarrow -5B{,}D7_{16} = -91{,}83984375_{10}$

Поля длиной 6 байтов — вещественные (обычная точность)

Код знака числа S — 1 бит (47 бит)
Код машинного порядка $P_м$ — 8 бит (39-46 биты)
Код мантиссы M — 39 бита (0-38 биты).

Правила прямого и обратного переходов те же самые, что и при использовании четырехбайтовых полей

Поля длиной 8 байтов — длинные вещественные (двойная точность)

Код знака числа S — 1 бит (63 бит)
Код машинного порядка $P_м$ — 11 бит (52-62 биты)
Код мантиссы M — 52 бита (0-51 биты).

Правила прямого и обратного переходов те же самые, что и при использовании четырехбайтовых полей, но машинный порядок отличается от истинного на 1023_{10}

Пример: **$+15{,}375_{10} \rightarrow 1111{,}011_2 \rightarrow 1{,}111011\times10^{11}{}_2$**

$S=0;\quad P_и = 11_2;\ P_м = P_и + 11\ 1111\ 1111_2 = 100\ 0000\ 0010_2;\quad M = 111011_2$

Код числа **$+15{,}375_{10}$** в формате с плавающей точкой в восьмибайтовом поле
$0100\ 0000\ 0010\ 1110\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
или **$40\ 2E\ C0\ 00\ 00\ 00\ 00\ 00_{16}$**

Поля длиной 10 байтов — расширенные вещественные

Код знака числа S	—	1 бит	(79 бит)
Код машинного порядка $P_м$	—	15 бит	(64-78 биты)
Код мантиссы M	—	64 бита	(0-63 биты).

Правила прямого и обратного переходов те же самые, что и при использовании четырехбайтовых полей, но машинный порядок отличается от истинного на $16\,383_{10}$ и у кода мантиссы бит целой части не отбрасывается

Пример: **$+15{,}375_{10} \rightarrow 1111{,}011_2 \rightarrow 1{,}111011\times10^{11}{}_2$**

S=0; $P_и = 11_2$; $P_м = P_и + 11\ 1111\ 1111\ 1111_2 = 100\ 0000\ 0000\ 0010_2$; $M = 1111011_2$

Код числа **$+15{,}375_{10}$** в формате с плавающей точкой в десятибайтовом поле
$0100\ 0000\ 0000\ 0010\ 1111\ 0110\ \ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$
или **$40\ 02\ F6\ 00\ 00\ 00\ 00\ 00\ 00\ 00_{16}$**

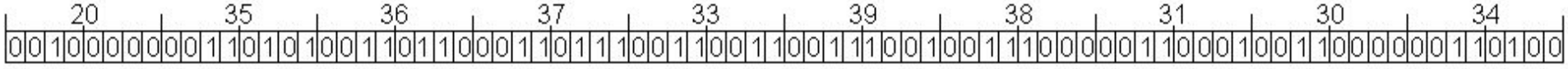
Двоично-десятичный формат (BCD – формат — binary coded decimal)

Разновидности BCD кодирования

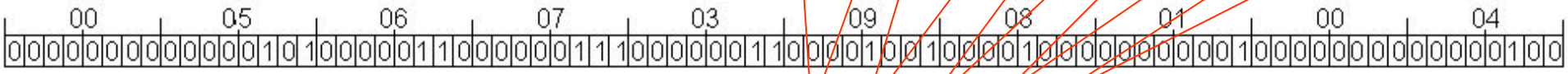
- Неупакованный BCD формат.
- Упакованный BCD формат.
- Знаковый BCD формат.

Пример: число: **+567 398 104**

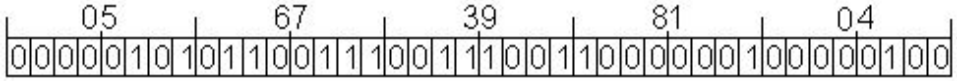
Текстовый код ASCII: **20 35 36 37 33 39 38 31 30 34$_{16}$**



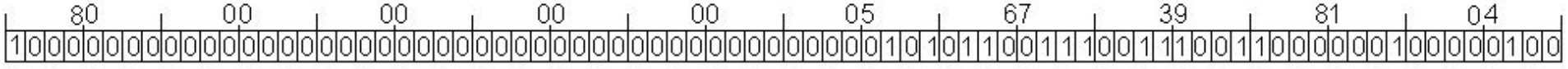
Неупакованный BCD формат: **00 05 06 07 03 09 08 01 00 04$_{16}$**



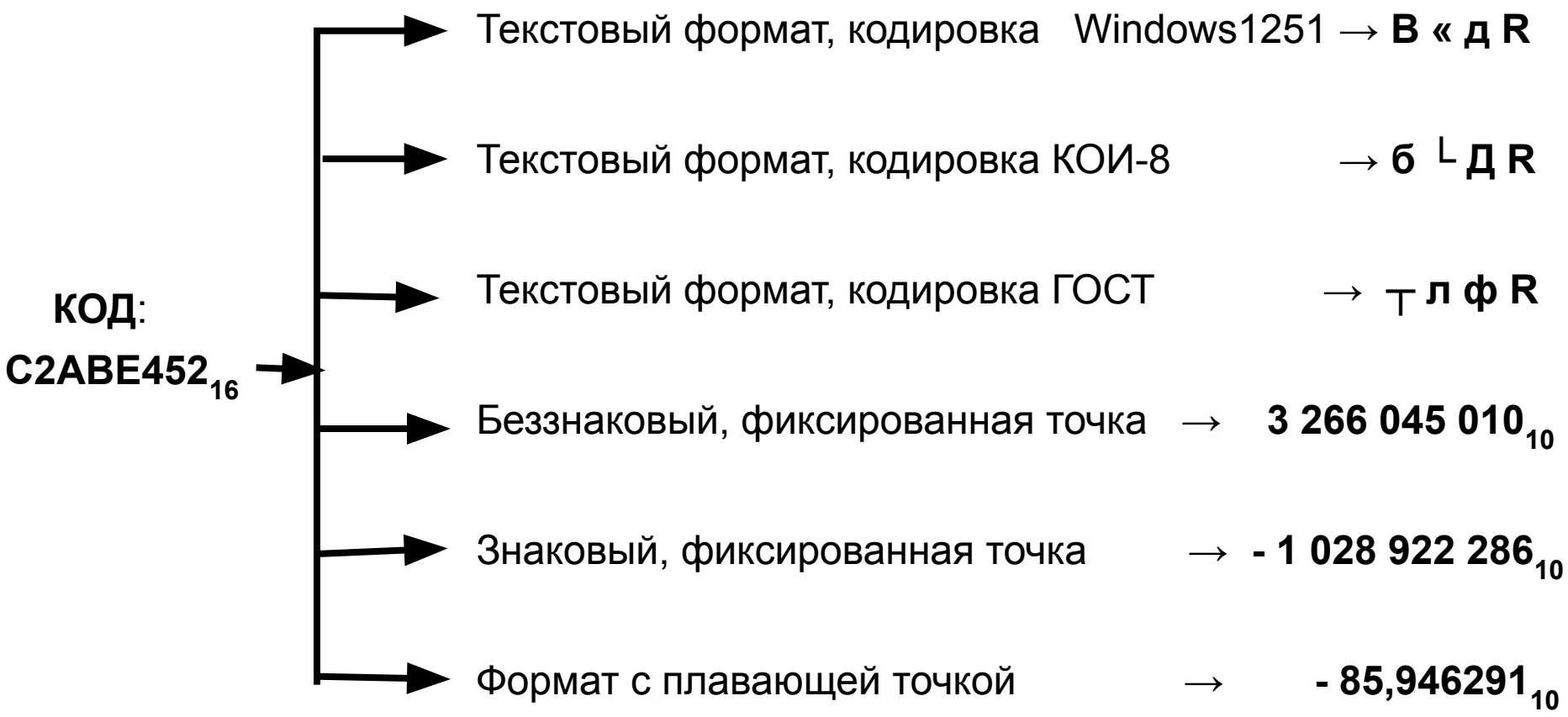
Упакованный BCD формат: **05 67 39 81 04$_{16}$**



Знаковый BCD формат (для **- 567398104**) : **80 00 00 00 00 05 67 39 81 04$_{16}$**



Один и то же код может восприниматься и как код цепочки символов и как код целого числа в формате с фиксированной точкой и как код числа в формате с плавающей точкой, а также в любых других смыслах



Кодирование графических данных

Необходимость в кодировании графических данных возникла в связи с появлением в шестидесятых годах XX века в составе компьютеров устройств на основе электроннолучевых трубок — **дисплеев** (англ. display — показывать, демонстрировать).

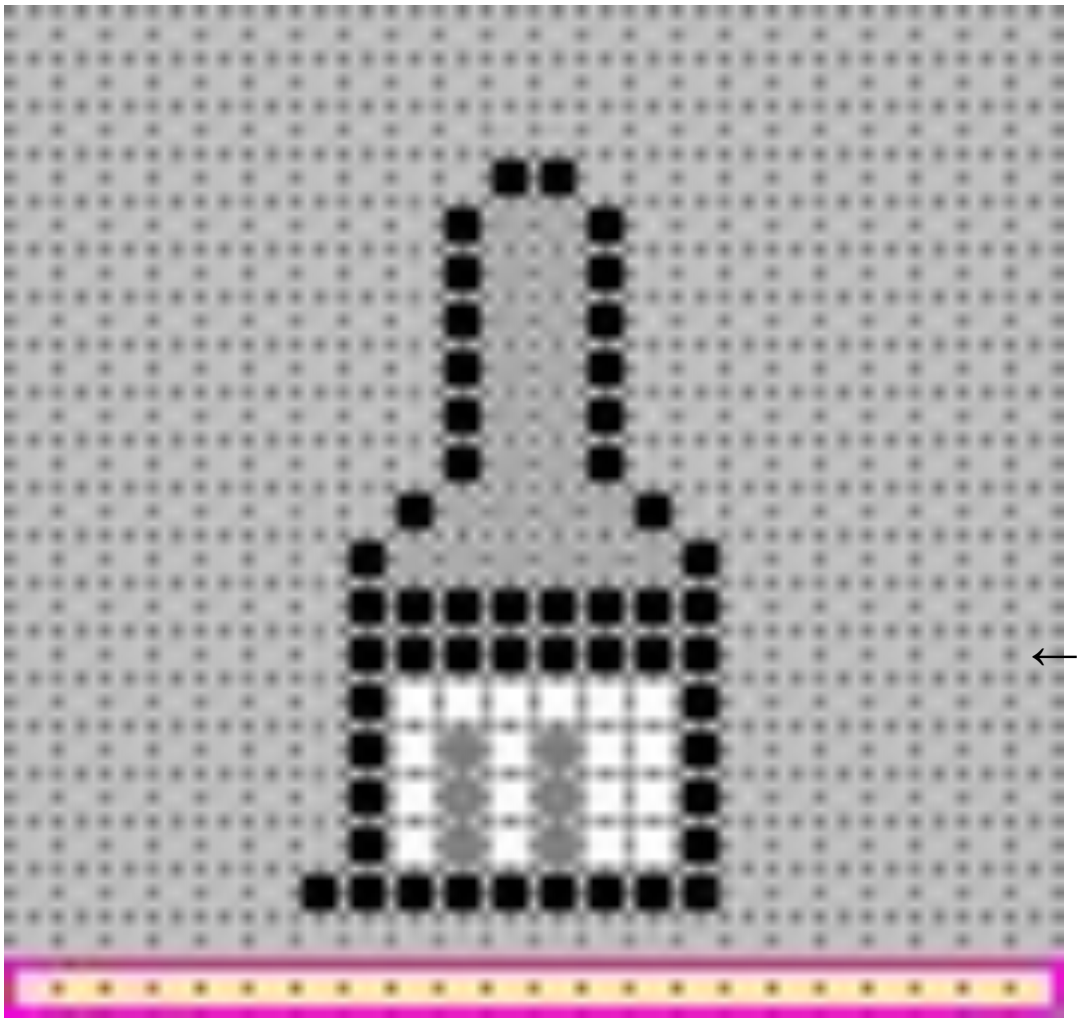
Под графическими данными можно понимать рисунок, чертеж, картинку в книге, фотографию, изображения на экране телевизора или в кинозале и т. д.

Основные виды компьютерной графики в настоящее время: ***растровая***, ***векторная***, ***фрактальная***, ***flash – графика***.

Основы кодирования растровой графики

Изображение строится из горизонтальных линий — строк. А каждая строка в свою очередь состоит из элементарных мельчайших единиц изображения — точек, которые принято называть ***пикселями*** (picsel — PICture'S ELement — элемент картинки).

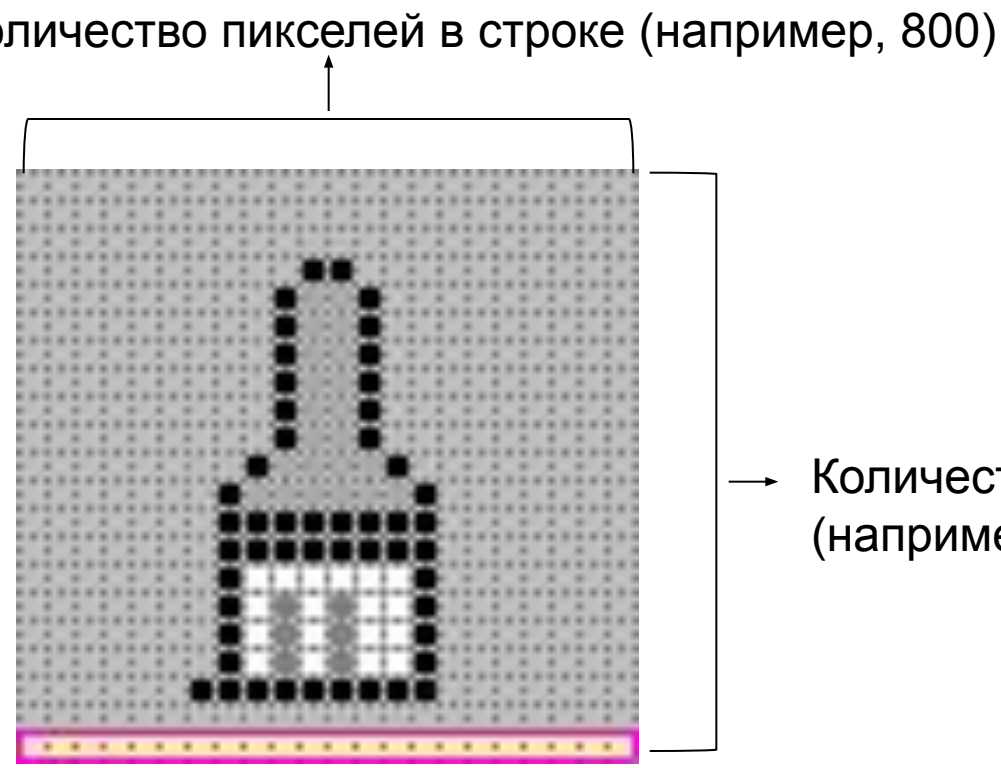
Изображение на экране → Фактическая структура изображения:



Всю совокупность элементарных единиц изображения называют ***растром*** (лат. rastrum — грабли).

Степень четкости изображения зависит от количества строк на весь экран и количества пикселей в строке, которые представляют ***разрешающую способность*** экрана или просто ***разрешение*.** Например**,** разрешение 800×600, то есть 800 точек на строку и 600 строчек на экран.

Стандартно используемые разрешения:
800×600,
1024 × 768,
1152 × 864,
1280 × 1024,
1600 × 1200,
1600 × 1280,
1920 × 1200,
1920 × 1600,
2048 × 1536.



Строки, из которых состоит изображение, принято просматривать сверху вниз друг за другом, как бы составив из них одну сплошную линию. Такой способ работы со строками называется ***строчной разверткой***, или ***сканированием*** (scan — бегло просматривать; разлагать, развертывать изображение).

Кодирование черно-белых изображений

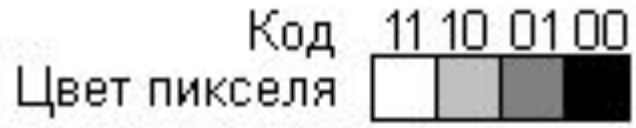
1. В изображении используется два цвета: белый и черный. Состояние пикселя кодируется одним битом.

Пример кодирования одной строки изображения.

На строчку, состоящую из 800 пикселей (точек), требуется 100 байтов памяти, а на весь растр 800×600 (на всё изображение) — 60 000 байтов.

Недостаток: получается чрезвычайно контрастное изображение

2. Более реалистический вариант — в изображении используются четыре цвета: белый, светло-серый, темно-серый и черный. На один пиксель требуется два бита.



На растр 800×600 требуется 120 000 байтов.

3. Стандартный способ — в монохромном изображении используются 256 оттенков серого цвета от белого до черного.

На кодирование одного пикселя требуется один байт, а на весь растр 800×600 — 480 000 байтов.

Для формирования цветных изображений используются несколько различных способов, которые принято называть **цветовыми системами**, **цветовыми моделями** или **цветовыми схемами**. Наиболее известную цветовую систему, которая называется **RGB**, предложил Д. Максвелл в 1860 году.

Кодирование цветных пикселей в методе RGB основано на том, что белый цвет можно представить как сумму трех основных цветов: красного (***R***ed), зеленого (***G***reen), синего (***B***lue), а их смешение в различных пропорциях дает остальные цвета.

В конечном счете, любой конкретный цветовой оттенок определяется долей присутствия в нём каждого из основных цветов. В разных цветовых системах такая доля задается различными способами, но в любом случае она представляет собой *целое число*, которое считается **цветовой координатой** соответствующего основного цвета.

По аналогии с тремя координатами точки в обычном трёхмерном геометрическом пространстве, совокупность цветовых координат основных цветов определяет некоторый цвет — **цветовой вектор**, **цветовую точку** в **цветовом пространстве**, то есть во множестве цветов, представимых в конкретной цветовой системе.

Цветовая система представляет собой способ представления цвета в виде набора чисел (цветового вектора) из трёх (иногда четырёх) значений, которые называются **цветовыми координатами** или **цветовыми компонентами**. В цветовую систему входят также правила определения соответствий между числовыми значениями координат и конкретными цветовыми оттенками, условия для воспроизведения изображения и т.д. Множество всех возможных цветов, представимых в цветовой системе, называется её **цветовым пространством**.

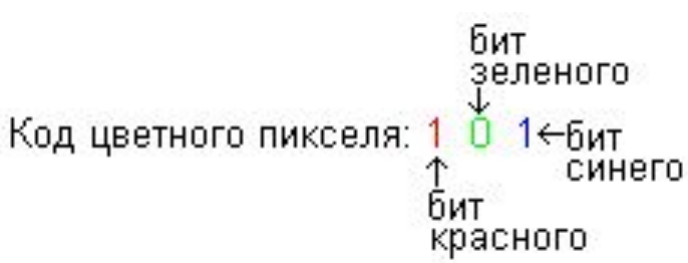
В соответствии с этими представлениями, для формирования цветного пикселя изображения в одну и ту же точку экрана направляется не один, а сразу три цветных (красный, зелёный и синий) луча, совместное воздействие которых на эту точку придаёт ей нужный цвет.

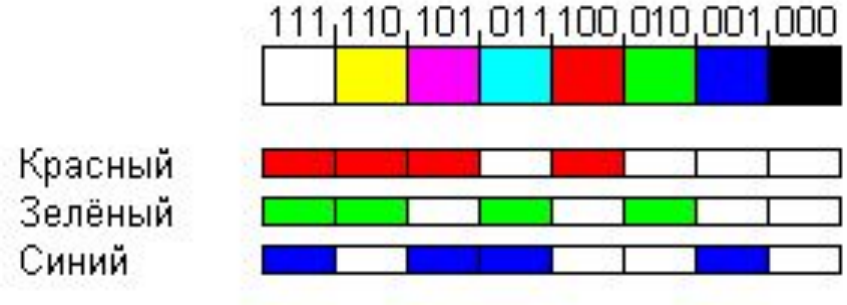
Вначале вновь будем считать, что для кодирования каждого из основных цветов достаточно одного бита, следовательно, каждая из цветовых координат может принимать только одно из двух значений 0 или 1.

При этом нулевое значение означает, что в суммарном цвете данный основной цвет *отсутствует*, а единичное — *присутствует*.

В этом случае для кодирования одного *цветного* пикселя потребуется три бита — по одному на каждый цвет.

Пусть первый бит соответствует красному цвету, второй — зеленому и третий — синему. Тогда код (цветовой вектор) 101_2 обозначает сиреневый цвет — красный есть, зеленого нет, синий есть, а код 110_2 — желтый цвет — красный есть, зеленый есть, синего нет. В этой системе кодирования цветовое пространство состоит из восьми цветов, которыми может быть окрашен любой пиксель изображения.





На кодирование растра 800×600 необходимо 180 000 байтов.

2. ***Полноцветный*** режим (режим ***True Color*** — истинный цвет). Для кодирования каждого из основных цветов выделяется один байт.

Можно задать 256 оттенков каждого из основных цветов — всего 16 777 216 различных цветов. На кодирование одного пикселя требуется три байта, на растр 800×600 — 1 440 000 байтов.

В целях экономии памяти разрабатываются различные графические режимы и форматы, которые немного *хуже* передают цвета и изображение в целом, но требуют меньше памяти. Например:

3. Режим ***High Color*** (high color — богатый цвет). Для кодирования цвета одного пикселя используется два байта. Можно передать 65 535 цветовых оттенков. Для растра 800×600 требуется 960 000 байтов.

4. ***Индексный*** режим. Для кодирования цвета одного пикселя используется один байт. Базируется на заранее созданной для данного рисунка таблице используемых в нем цветовых оттенков. Затем нужный цвет пикселя выбирается из этой таблицы с помощью номера — индекса, который занимает всего один байт памяти. Для растра 800×600 требуется 480 000 байтов.

Конкретный способ кодирования всей требуемой при записи изображения в память компьютера информации (цвет, яркость, контрастность, разрешение) образует ***графический формат***. Форматы кодирования графической информации, основанные на передаче цвета каждого отдельного пикселя, из которого состоит изображение, относят к группе ***растровых*** или ***BMP*** (**B**it **M**a**P** — битовая карта) форматов.

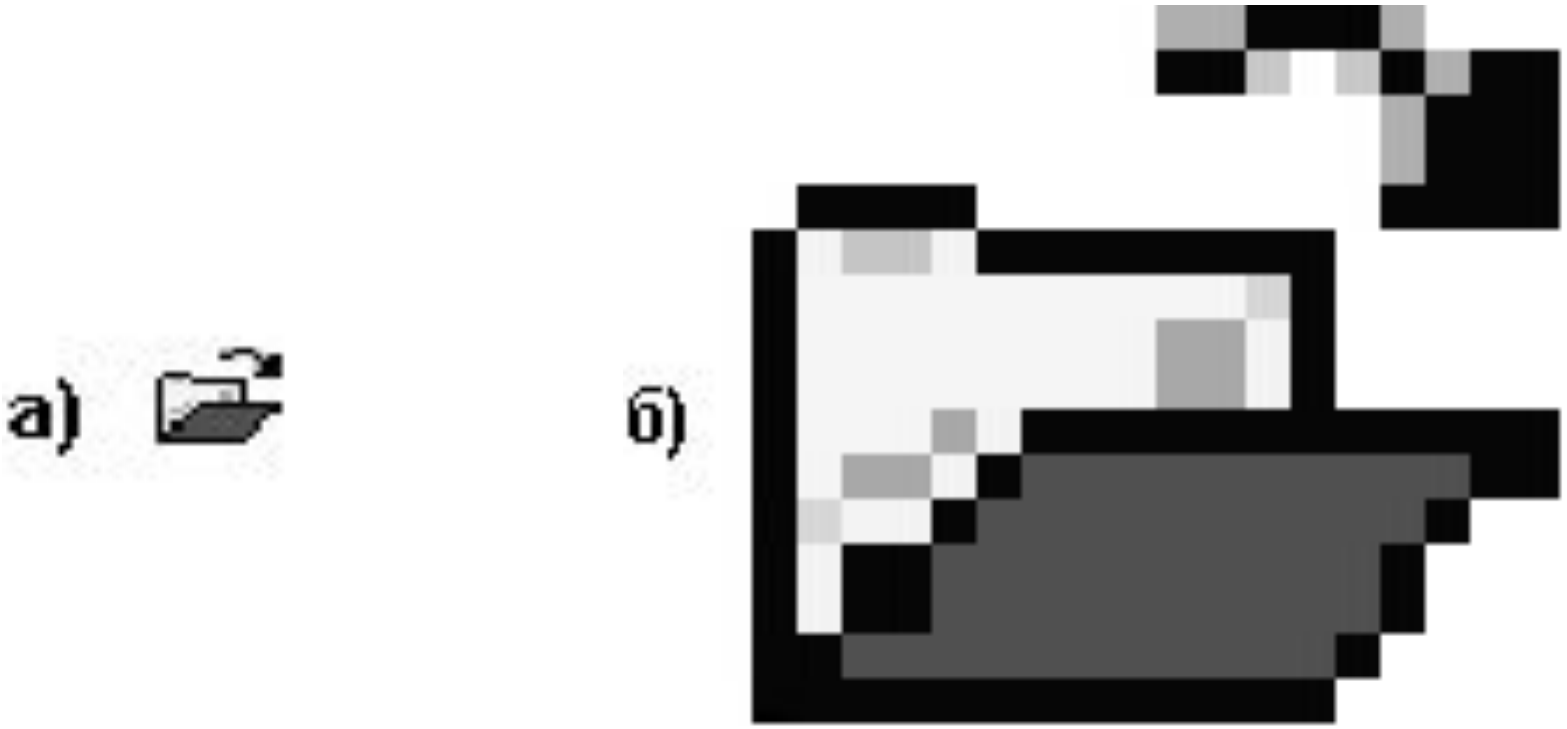
- Метод ***CMYK*** (от ***C***yan — голубой, ***M***agenta — пурпурный, ***Y***ellow — желтый и blac***K*** — черный) основан на использовании цветов дополнительных к трем основным. Дополнением к красному является сочетание зеленого и синего — голубой. Дополнением к зеленому является сочетание красного и синего — пурпурный цвет. Дополнение к синему является сочетание красного и зеленого — голубой. Этот режим также относится к полноцветным. Для кодирования цвета одного пикселя требуется четыре байта памяти. Может быть передано 4 294 967 295 различных цветов.

- Формат ***bmp*** (или ***dib*** от ***D***evice ***I***ndependent ***B***itmap — независимый от устройства bitmap). Задается цветность всех пикселей изображения. При этом можно выбрать монохромный режим, использующий 256 градаций серого цвета, или цветной, использующий 16, 256 или 16 777 216 цветов. Изображения в этом формате требует много памяти.

- Формат ***gif*** (от ***G***raphics ***I***nterchange ***F***ormat — графический формат обмена). Используются специальные методы *сжатия* кода, поддерживается только 256 цветов. Качество изображения немного хуже, чем в формате ***bmp***, зато изображение занимает в десятки раз меньше памяти.

- Формат ***jpeg*** (***J***oint ***P***hotographic ***E***xperts ***G***roup — объединенная группа экспертов по фотографии) использует методы сжатия, основанные на удалении «избыточной» информации и приводящие к потерям некоторых деталей. Поддерживает передачу 16 777 216 цветов. Достаточно высокое качество изображения. По требованиям к памяти формат ***jpeg*** занимает промежуточное положение между форматами ***bmp*** и ***gif***.
- Формат ***png*** (***P***ortable ***N***etwork ***G***raphics — компактная графика для сети) представляет собой улучшение формата ***gif*** в области цветопередачи и сжатия кода. Создан для передачи изображений в Интернете. Не подходит для печати изображений.
- Формат ***tiff*** (***T***ag ***I***mage ***F***ile ***F***ormat — формат файла образа, изображения, tag — используемый в реализации формата тэг, этикетка, ярлык). Представляет собой универсальный формат для хранения растровых изображений. Широко используется в издательских системах. По качеству сжатия он близок к форматам png и gif.
- Формат ***pdf*** (***P***ortable ***D***ocument ***F***ormat — компактный формат документов) Основное достоинство этого формата в том, что текст и графика, заключенный в электронном документе этого формата одинаково точно воспроизводятся на любой аппаратной платформе, что особенно важно в компьютерных сетях.

Растровая графика дает высококачественные изображения, но имеет существенный недостаток — плохо переносит масштабирование, то есть изменение размеров.



а) исходное изображение, б) то же изображение, увеличенное в 8 раз

Другие виды компьютерной графики

- ***Векторная*** графика. Базовым объектом является не точка, а линия. При этом изображение формируется из описываемых математическим, векторным способом отдельных отрезков прямых или кривых линий, а также геометрических фигур — прямоугольников, окружностей и т. д., которые могут быть из них получены.

- ***Трехмерная*** графика. Является особой разновидностью векторной графики, в которой специальными средствами фактически плоского рисунка добиваются визуальных эффектов объемности изображений.

- ***Фрактальная*** графика. Формирование изображений целиком основано на математических формулах, уравнениях, описывающих те или иные фигуры, поверхности, тела. При этом само изображение в памяти компьютера фактически не хранится — оно получается как результат обработки некоторых данных. Таким способом, например, могут быть получены довольно реалистичные изображения природных ландшафтов.

- ***Flash*** графика. В 1996 году компания Macromedia разработала стандарт *flash* графики (flash — мгновение, короткая телеграмма). Основное назначение этой технологии работы с графикой — создание высококачественных анимационных изображений для веб страниц Интернета.

Некоторые графические векторные форматы

- Формат ***wmf*** — ***W***indows ***M***eta ***F***ile — формат метафайлов операционной системы Windows. Используется для хранения созданных в приложениях операционной системы Windows изображений (например, в Microsoft Office).

- Формат ***cdr*** — ***C***orel ***DR***aw — формат хранения изображений, используемый в мощном графическом редакторе Corel Draw.

- Форматы ***ps*** — PostScript (от poster script — сценарий описания плакатов, объявлений, афиш) и ***eps*** — Encapsulated PostScript— инкапсулированный, то есть скрытый, заизолированный PostScript. Используются для описания как векторных, так и растровых изображений, а также разнообразных текстовых шрифтов.

Кодирование звуковых и видео данных

Развитие способов кодирования звуковых данных, а также движущихся изображений **анимации** (фр. animation — оживление, одушевление) и **видео** (лат. video — смотрю, вижу) — происходило со значительным запаздыванием относительно развития способов кодирования текстовых, числовых и графических данных.

Компьютерная анимация — последовательный показ подготовленных на компьютере рисунков, а также имитация движения с помощью изменения и перерисовки объектов или показа последовательных рисунков с соответствующими фазами движения.

Приемлемые возможности для хранения и воспроизведения с помощью компьютера звука и видеозаписей появились только в девяностых годах XX века. Разработанные в это время методы, способы, приёмы работы со звуком и видео получили название **мультимедийных технологий**.

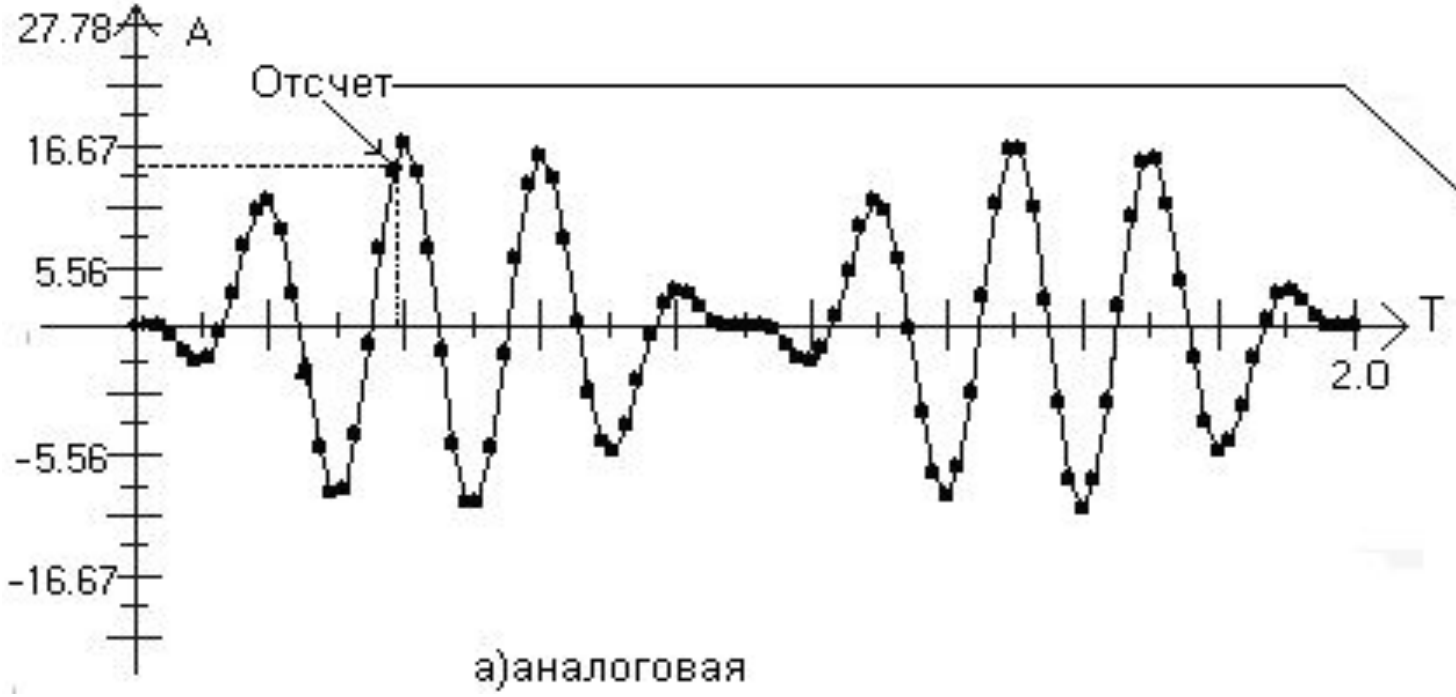
В связи с большими объёмами памяти, требующейся для хранения графики, видео и звука, эти разновидности данных практически всегда кодируются с помощью *сжимающих методов с потерей информации*.

С точки зрения физики звук представляет собой достаточно сложное, непрерывное колебание воздуха. Это значит, что с точки зрения информатики звук является разновидностью *непрерывных сообщений*.

Поэтому для представления в компьютере непрерывное сообщение необходимо преобразовать в дискретную форму — выполнить развертку по времени и квантование. Осуществление этих операций во время кодирования звука называется **преобразованием в цифровую форму** или **оцифровыванием**.

Более точно: замена ***непрерывного*** звукового сигнала ***дискретным*** набором его значений — ***отсчетов*** — в некоторые последовательные моменты времени называется ***дискретизацией***, ***преобразованием в цифровую форму*** или просто ***оцифровыванием*.**

Заметим, что получающиеся при оцифровывании значения *отсчётов* в литературе иногда называют **сэмплами** (англ. sample — образец). Но для обозначения *отдельного отсчёта* это название некорректно потому, что под термином «сэмпл» в практике оцифровывания звука понимается относительно небольшой звуковой *фрагмент*, а не отдельный отсчёт.



T	A
0,31	-0,51345
0,32	6,24756
0,33	14,71281
0,34	17,82943
0,35	19,24682
0,36	15,02881
0,37	5,98843
0,38	-1,21784
0,39	-4,95674
0,40	-13,52471

а)аналоговая

б) цифровая

На рисунке показана развертка по времени звукового сигнала, который длится 2 секунды и заменяется на 100 *отсчётов* (жирные точки на графике сигнала). Количество отсчетов сигнала в единицу времени, которое принято называть **частотой дискретизации**, является очень важной характеристикой оцифровывания звука. Чем выше частота дискретизации, тем лучше качество оцифрованного звука.

Чему равна частота дискретизации в рассматриваемом примере?

50 герцам.

В общем случае значение частоты дискретизации выбирается исходя из требований теоремы В. А. Котельникова.

При записи звука в мультимедийных технологиях применяются частоты 8, 11, 22, 44 и более килогерц. Для получения высококачественного звука используются частоты до 192 килогерц. Например, частота дискретизации 44 килогерца означает замену одной секунды непрерывного звучания набором из сорока четырех тысяч отдельных отсчетов сигнала.

Качество преобразования звука в цифровую форму определяется не только частотой дискретизации, но и количеством битов памяти, отводимых на запись кода одного отсчета. При этом каждый отсчет во время *квантования* кодируется в одном из разобранных выше числовых форматов.

Этот параметр, который принято называть **разрядностью преобразования**, определяет количество различных уровней сигнала, представимых при его квантовании. Чем больше количество битов, тем более качественно и точно происходит воспроизведение оцифрованного звука. Обычно используются разрядности 8, 16 и 24 бит.

Таким образом, кодирование и обработка звука фактически сводится к кодированию и обработке *числовых данных.*

На обсуждаемых стандартных способах оцифровывания звука основан, так называемый, волновой формат кодирования звука ***wav*** (от ***WAV***eform-audio — волновая форма аудио)**.**

Получить запись звука в этом формате можно от подключаемых к компьютеру микрофона, проигрывателя, магнитофона, телевизора и других стандартно используемых устройств работы со звуком.

Формат ***wav*** требует очень много памяти. Так, при записи стереофонического звука с частотой дискретизации 44 килогерца и разрядностью 16 бит — параметрами, дающими достаточно хорошее качество звучания — на одну минуту записи требуется около десяти миллионов байтов памяти.

В связи с особенностями программ, осуществляющих кодирование звука, оцифровывание удобнее характеризовать одним параметром, который называется **битрейтом** (англ. bit rate — битовая скорость).

По своей сути битрейт это — количество битов сформированного кода, требующихся на одну секунду звучания, и его значение получается как произведение частоты дискретизации на разрядность преобразования.

Единицей измерения битрейта является бит в секунду (бит/с или bps — bit per second). Широко используются также кратные единицы — килобиты в секунду (кбит/с, kbit/s или kbps) и мегабиты в секунду (мбит/с, mbit/s или mbps). Во избежание путаницы отметим, что здесь приставки кило- и мега- означают 1 000 и 1 000 000 соответственно.

Для рассмотренного выше случая оцифровывания с частотой дискретизации 44 килогерца и разрядностью 16 бит битрейт равен 44 000 (сек$^{-1}$)×16 (бит) = 704 000 бит/с или 704 кбит/с. А для стереозвука, то есть для кодирования двух каналов битрейт оказывается равным 1408 кбит/с или примерно 1,4 мбит/с. Этот битрейт считается стандартным для *несжатого* звука формата **CD-Audio**, который применяется для записи музыкальных произведений на компакт-дисках.

На самом деле, точное значение стандартного битрейта равно 1411,2 кбит/с, так как частота дискретизации выбирается равной 44 100 герц — немного выше, чем в примере.

Для записи звука широко применяется формат ***midi*** (***M***usical ***I***nstruments ***D***igital ***I***nterface — цифровой интерфейс музыкальных инструментов). Представляет собой набор инструкций, команд ***музыкального синтезатора —*** устройства, которое имитирует звучание реальных музыкальных инструментов. Команды синтезатора фактически являются указаниями на высоту ноты, длительность ее звучания, тип имитируемого музыкального инструмента и т. д.

Кодирование видеоинформации осуществляется с помощью дискретизации непрерывных изображений. Основано на свойстве человеческого глаза воспринимать быструю смену последовательности картинок с небольшими изменениями одна относительно другой как одну картинку с движением. Этот принцип используется в кино, в мультипликации и анимации. Стандартной частотой дискретизации для видеоизображений является 30 изображений (кадров) в секунду. Кодирование требует не только дискретизации непрерывных движений, но и синхронизации изображения со звуковым сопровождением. В настоящее время для этого используется формат, которой называется ***avi*** (***A***udio-***V***ideo ***I***nterleaved — чередующееся аудио и видео).

Форматы, основанные на сжатии информации

Стандарты сжатия звука и видео ***MPEG*** (***M***oving ***P***ictures ***E***xperts ***G***roup — группа экспертов по движущимся изображениям) включают:

- стандарт ***MPEG-1*** описывает форматы записи звука. Например, формат ***MP-3*** при практически том же качестве звука требуется в десять раз меньше памяти, чем при использовании формата ***WAV.***
- стандарт ***MPEG-2*** описывает методы сжатия видеозаписей, которые обеспечивают телевизионное качество изображения и стереозвуковое сопровождение и имеют приемлемые требования к памяти.
- стандарт ***MPEG-4*** описывает методы сжатия видеозаписей, которые обеспечивают минимальные требования к памяти. Например, с его помощью можно полнометражный цветной фильм со звуковым сопровождением записать на стандартный компакт-диск. Разработан в 1999 году.

АЛГОРИТМЫ

Вопросы, связанные с представлением сообщений, кодированием и хранением данных в памяти компьютера, интересны не сами по себе. Необходимо уметь не только хранить данные, но и что-то с ними делать.

Выделенная из принятого сообщения информация так или иначе обрабатывается и возможно запоминается. Обработка информации может происходить осознанно или на уровне подсознания.

Но в любом случае эта обработка производится по каким-то правилам, при этом выполняется некоторая последовательность действий. В результате получается, вообще говоря, некоторая новая информация, которая может быть представлена в форме сообщения. Эти правила обработки, эту последовательность действий принято называть **алгоритмом.**

Понятие алгоритма, в отличие от большинства других понятий информатики, имеет довольно длинную историю. Происхождение этого термина обычно связывают с именем узбекского учёного аль-Хорезми, написавшего около 825 года книгу «Об индийском счёте», в которой описаны *правила выполнения действий* над числами в десятичной позиционной системе счисления.

Имя автора трактата в европеизированной форме (Algorithmus) стало обозначением десятичной арифметики.

В настоящее время *считается*, что именно так появился современный термин *алгоритм*. Кроме того *предполагают*, что именно аль-Хорезми впервые использовал цифру 0 (нуль) для обозначения пропущенной позиции в записи числа в позиционной системе счисления. Её индийское название переведено как *sifr*, отсюда произошли нынешние термины «цифра» и «шифр».

Аль-Хорезми также считается основателем алгебры. Он написал книгу «Ал-китаб ал мухтасар фи хисаб ал-джабр ва-л-мукабала» («Книга о восполнении и противопоставлении») в которой описано несколько разновидностей квадратных уравнений и приведены способы их решения, в частности, с помощью приведения подобных, переноса в другую часть уравнения с переменой знака и т.д. От части названия книги «ал-джабр» произошло слово «алгебра».

Постепенно под термином алгоритм стали понимать любые чётко заданные предписания, правила выполнения каких-либо действий.

В 30–50-х годах XX века в связи с формированием теории алгоритмов произошло кардинальное изменение в понимании их роли и значения в научной сфере и обычной жизни человека. В это время термин алгоритм приобрел чёткое научное определение.

Одновременно выяснилось, что огромное большинство своих действий в повседневной жизни и в профессиональной деятельности: умывание, приготовление различных кулинарных блюд, переход улицы, выпечку хлеба, выплавку стали, выращивание винограда, и т.д. человек осуществляет по некоторым вполне определённым *алгоритмам*.

В настоящее время понятие алгоритм является сложным и многоплановым, тем не менее, можно выделить три основных аспекта в его использовании:

- интуитивный;
- теоретический;
- фундаментальный.

Интуитивное понятие алгоритма связано с бытовым представлением об алгоритме как о способе решения некоторой задачи, как о последовательности действий, которую исполнитель алгоритма должен совершить, чтобы достичь поставленной цели: дойти до требуемого пункта в городе, приготовить кофе, сварить суп, решить математическую задачу и т.д.

При этом исполнителем алгоритма может быть дрессированное животное, человек или некоторый механизм, выполняющий действия самостоятельно, *автоматически* или совместно с человеком.

В этом смысле термин алгоритм является совершенно понятным, хотя и нестрогим. В повседневном применении он не требует никаких дополнительных пояснений и тем более каких-либо определений.

Интуитивного понятия алгоритма вполне достаточно не только в быту. В настоящее время указанное понимание устраивает специалистов большинства профессий, включая инженеров и даже программистов.

Заметим однако, что для программистов ещё необходимо чтобы алгоритм обладал некоторой, обсуждаемой далее совокупностью свойств.

Вместе с тем, для решения более сложных задач, которые оказались в центре внимания математиков в первой трети XX века, потребовалось более точное **теоретическое понимание** алгоритма.

В это время математики столкнулись с целым рядом важных проблем и задач, относительно которых возникли существенные подозрения в принципиальном отсутствии у них решений.

В связи с появлением доказательств принципиальной невозможности получить решения некоторых задач был сделан естественный вывод о том, что необходимо своевременное выявление неразрешимых проблем с тем, чтобы не затрачивать бесполезно время и усилия.

То есть, необходимо уметь доказывать *отсутствие алгоритма* (способа) решения той или иной задачи.

Как оказалось, задача доказательства *отсутствия* решения совершенно отличается от задачи *нахождения* какого-либо решения.

Для доказательства существования алгоритма решения задачи достаточно, используя интуитивное представление об этом термине, найти (можно даже просто угадать) некоторый способ её решения.

В то время как для доказательства *отсутствия* алгоритма необходимо абсолютно точно знать, что же именно не существует, а для этого сначала требуется определить, что такое алгоритм. Следовательно, требуется *строгое* и *точное* определение этого понятия.

Возьмем в качестве иллюстрации этих рассуждений геометрическую задачу о делении любого угла на две равные части с помощью циркуля и линейки, причём линейка может использоваться *только* для проведения прямой линии через две точки.

А вот внешне аналогичная *задача о трисекции угла* (о делении произвольного угла на три равные части) при тех же ограничениях решения не имеет, и никогда не будет иметь. Чтобы установить этот факт учёным понадобилось более 2 000 лет.

Вместе с тем, в процессе поиска решения задачи о трисекции было найдено много вариантов её решений, которые существуют при некоторых *ослаблениях на допустимые действия*. Так ещё Архимед предложил метод решения, в котором разрешались дополнительные операции с линейкой: нанесение на неё двух точек и такое перемещение линейки, когда одна из точек скользит вдоль прямой, а вторая — вдоль окружности.

Получается, что наличие или отсутствие решения прямо связаны с исполнителем алгоритма, с теми действиями, которые он может выполнять для достижения результата. Отсюда прямо вытекает, что доказательство отсутствия алгоритма не может быть осуществлено без абсолютно точной фиксации в определении алгоритма исполнителя и всех его возможных действий.

Задача точного определения понятия алгоритм стала центральной математической проблемой в двадцатых годах XX века. Её решение было получено в 30–50-х годах Д. Гильбертом, К. Гёделем, А. Черчем, С. Клини, Э. Постом, А. Тьюрингом, Н. Винером А. А. Марковым и А.Н. Колмогоровым.

В их работах были развиты строгие и точные **классические теории алгоритмов:** теория рекурсивных функций, теория машин Поста, теория машин Тьюринга, теория нормальных алгоритмов Маркова.

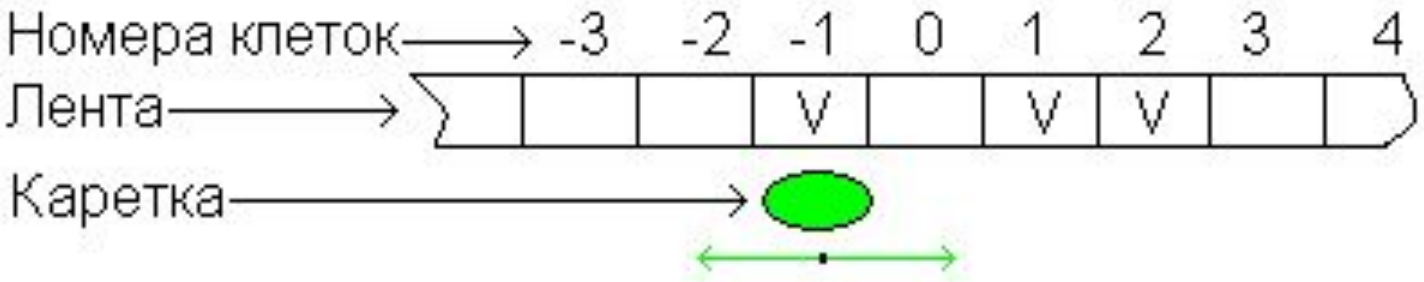
Классические теории алгоритмов иногда объединяют под общим названием **логической теории алгоритмов**.

Наиболее общим является **фундаментальный аспект** алгоритмов. В современной математике алгоритм является одним из тех немногих понятий, которые наряду с понятием множество привлекаются для *обоснования математики* как научной дисциплины, для создания фундамента всего математического здания.

С этой точки зрения понятие алгоритма считается *принципиально неопределяемым*. Обсуждение фундаментального аспекта понятия алгоритм выходит за рамки предмета информатика, поэтому в дальнейшем мы с ним сталкиваться не будем.

Машина Поста

Машина Поста представляет собой ***идеальное, воображаемое устройство***, состоящее из неограниченной в обе стороны ленты и каретки, способной перемещаться вдоль ленты. Лента разбита на клетки. Любая клетка может быть отмеченной или нет. За один шаг каретка может переместиться на одну клетку вправо или на одну клетку влево. Каретка может определить отмечена текущая клетка или нет, а также поставить в текущую клетку отметку или же стереть её.



Машина Поста задается:

1. начальным распределением отмеченных клеток;
2. начальным положение каретки;
3. программой, определяющей последовательность действий каретки.

Командой машины Поста называется указание исполнителю (каретке) выполнить единственное действие из набора возможных действий.

Система команд машины Поста

Операция	Действие
⇒	Шаг вправо к следующей клетке ленты
⇐	Шаг влево к предыдущей клетке ленты
V	Поставить отметку (отметить) текущую клетку
ξ	Стереть отметку в текущей клетке
	Условный переход
Стоп	Завершение работы машины

Структура команды машины Поста:

<порядковый номер команды>. <операция> <номер следующей команды>

Пример команды: ***5. ⇒ 9***

Пятая команда. Шаг вправо, перейти к выполнению команды с номером 9

Невыполнимые команды:

- стереть отметку, если текущая клетка не отмечена;
- поставить отметку, если текущая клетка уже отмечена.

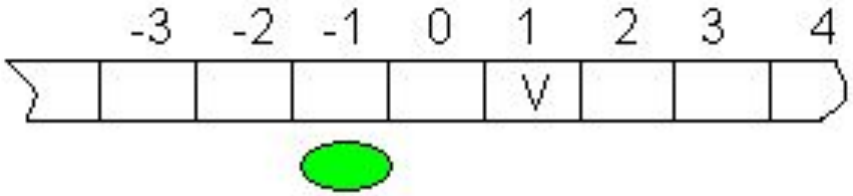
Выполнение программы может:

- не завершится никогда (безостановочная работа);
- завершится безрезультатно на невыполнимой команде;
- завершится результативно на команде ***Стоп***.

Требование к программе машины Поста

1. Порядковые номера команд программы должны образовывать возрастающую арифметическую последовательность, начинающуюся с номера 1 и с шагом 1.
2. Используемые в командах номера должны обозначать присутствующие в программе команды.

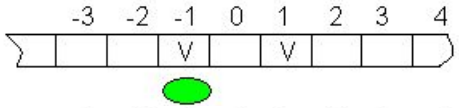
Пример программы машины Поста



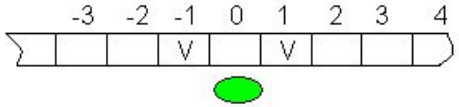
1. V 4
2. ξ 3
3. ⇐ 2
4. ⇒ 5
5. ? 4 3
6. Стоп

Выполнение программы

1. V 4



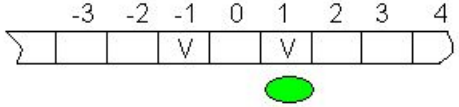
4. ⇒ 5



5. ? 4 3

Клетка пуста, переход к 4 команде

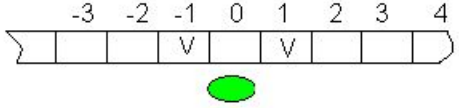
4. ⇒ 5



5. ? 4 3

Клетка не пуста, переход к 3 команде

3. ⇐ 2

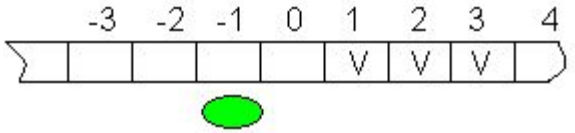


2. Стоп

Результатная остановка

Изображение целого неотрицательного числа ***n*** в машине Поста представляет собой последовательность из ***n+1*** подряд расположенных отмеченных клеток, которые ограничены с двух сторон неотмеченными клетками. Положение каретки безразлично.

Пример: изображение числа 2 в машине Поста



Программа для машины Поста прибавления единицы

1. ?↗ 2 / ↘ 3
2. ⇒ 1
3. ⇐ 4
4. V 5
5. Стоп

Интуитивное понятие алгоритма

Упомянутые выше классические теории, дающие точное определение алгоритма, связаны с очень *узким* классом алгоритмов. В то время как на практике приходится использовать значительно более сложные алгоритмы обработки данных.

С одной стороны, реально применяемые алгоритмы соответствуют точным классическим определениям, но при этом оказываются очень трудно реализуемыми на базе соответствующих этим теориям подходов.

С другой стороны для таких алгоритмов совершенно не нужен такой высокий уровень строгости как у классических теорий.

Использование классических подходов для возникающих на практике алгоритмов полностью аналогично доведению каждого математического доказательства по степени детализации до уровня алгебраических или геометрических аксиом. Такая степень подробности совершенно излишняя.

Поэтому для решения подавляющего большинства практически значимых задач можно обойтись *интуитивным*, нестрогим понятием алгоритма, его понятием в, так сказать, житейском смысле.

С точки зрения теоретических подходов интуитивное понятие алгоритма является некоторым его *объяснением*, которое опирается на бытовые аналогии, интуитивные ощущения и представления, а также общепризнанную терминологию.

Основными особенностями интуитивного представления об алгоритме является неопределённость терминов «исполнитель», «действие», «цель» и т.д. Именно поэтому существует несколько десятков «определений» алгоритма, которые на самом деле являются не более чем «объяснениями на пальцах».

Выдающийся советский математик А.Н. Колмогоров пишет, что «Алгоритм — это всякая система вычислений, выполняемых по строго определённым правилам, которая после какого-либо числа шагов заведомо приводит к решению поставленной задачи».

Создатель одной из классических теорий алгоритмов А.А. Марков приводит следующее объяснение: «Алгоритм — это точное предписание, определяющее вычислительный процесс, идущий от варьируемых исходных данных к искомому результату».

Алгоритмом называется понятное и точное предписание, указание исполнителю совершить последовательность действий, направленных на достижение указанной цели на решение поставленной задачи.

Алгоритм представляет собой совокупность правил, инструкций для исполнителя, выполняя которые он за конечное число шагов добьется искомого результата.

Алгоритм — это последовательность действий, либо приводящая к решению задачи, либо поясняющая, почему это решение получить нельзя;

Проанализировав все эти трактовки, можно заметить, что общими для них являются следующие моменты:

1. наличие некоторого исполнителя алгоритма;
2. существование точно определенных правил, инструкций, в которых строго определены необходимые действия и последовательность или порядок их выполнения;
3. конечность количества действий, которые должен совершить исполнитель для достижения цели, результата.

Исполнителем алгоритма называется субъект или объект, осуществляющий фактическое исполнение алгоритма.

Под исполнителем алгоритма может пониматься человек или животное, это может быть также некоторое неодушевлённое устройство: компьютер, токарный станок, швейная или стиральная машина и т.д.

Исполнение (выполнение) алгоритма представляет собой точную реализацию предписанных в алгоритме действий в конкретных условиях решаемой задачи.

Алгоритм есть описание способа решения задачи. Таким образом, ***исполнение алгоритма*** есть фактическое решение задачи по заданному в алгоритме способу.

Форма задания правил, инструкций и порядка их выполнения существенно зависят от исполнителя алгоритма: для дрессированного животного это короткие условные слова или жесты, для человека — всевозможные инструкции по применению и выполнению, а для неодушевлённых устройств — это особые для каждого типа устройств специальные последовательности команд.

И основным моментом, конечно же, является получение желаемого результата за конечное время. Более того, неявно подразумевается, что это время должно быть разумным, поскольку ждать от исполнителя требуемого результата сотни или даже десятки лет, обычно не может считаться приемлемым.

Алгоритмы являются:

- формой изложения научных результатов;
- руководством к действию при решении ранее изученных проблем;
- необходимым этапом при автоматизации обработки информации и решения различных задач.

Следует подчеркнуть, что алгоритмы обладают замечательным качеством: исполнитель алгоритма, если он последовательно и четко выполняет указанные, зафиксированные в алгоритме действия, предписания, инструкции может достигнуть желаемого результата даже не имея ни малейшего представления о том почему и зачем требуется выполнять предписанные действия.

От исполнителя требуется полное отсутствие любой инициативы, он должен действовать, формально, механически, ни на грамм, ни на миллиметр не отступая от предписаний алгоритма.

Именно это качество алгоритмов позволяет использовать их для *автоматизации* обработки данных, поручать такую обработку вычислительным машинам (компьютерам). Естественно, для этого алгоритмы должны быть заранее сформулированы, построены и записаны в форме, понятной компьютеру

Алгоритм, записанный в понятной компьютеру или другому исполнителю форме, принято называть **программой**

Компьютеры, действуя с огромной скоростью, могут с высокой надежностью выполнять переданные им в виде программ алгоритмы и тем самым решать самые разные задачи математики, экономики, управления, информационно-поисковые задачи и т.д. Поэтому построение алгоритма является необходимым этапом при *автоматизации* обработки данных и решения различных задач.

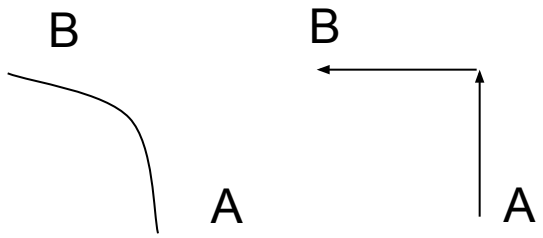
Процесс разработки, создания алгоритма является, вообще говоря, глубоко творческим делом. На первоначальную разработку алгоритмов могут уходить годы и десятилетия, может потребоваться объединение усилий десятков специалистов и ученых.

В то время как исполнение алгоритма не требует никакого интеллекта, его можно поручить неодушевлённому устройству, например, компьютеру, при этом собственно исполнение в подавляющем большинстве случаев может быть осуществлено за сравнительно короткое время.

Для того, чтобы некую совокупность правил, инструкций, указаний можно было объявить алгоритмом, и тем более для того, чтобы передать этот алгоритм для исполнения компьютеру, указанная совокупность должна обладать рядом свойств, удовлетворять определенным требованиям. Поэтому интуитивное понятие алгоритма кроме простого объяснения обычно подкрепляется перечислением характерных черт, свойств, которыми должен обладать алгоритм.

Свойства, которыми должен обладать алгоритм

Дискретность. Вообще говоря, один и тот же результат может быть получен как с помощью *непрерывного*, так и с помощью *дискретного процесса*



Но компьютер представляет собой *дискретное устройство*, поэтому в информатике всегда используются способы решения задач, опирающиеся на дискретные процессы.

Обсуждая свойство дискретности алгоритма А.Н. Колмогоров пишет, что «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 определённому закону (программе) из системы величин, имевшихся в предыдущий момент времени».

Исполнение алгоритма *всегда* представляет собой дискретный процесс идущий в дискретном времени.

Это значит, что преобразование исходных данных в результат осуществляется поэтапно. Следовательно, алгоритм должен представлять процесс решения задачи как последовательное выполнение некоторых простых этапов. явно различимых, чётко отделенных друг от друга шагов, этапов. И для любого этапа должна существовать возможность указать один или два *соседних.*

Конечность алгоритма означает, что результат его исполнения требуется получить за конечное, и как правило относительно небольшое время. Абсолютно бессмысленно говорить о необходимости затратить на получение результата *бесконечное* время.

Чтобы исполнитель алгоритма завершил работу и выдал результат за конечное время требуется, чтобы в его описании имелось только *конечное* количество *явно заданных* этапов и при этом на выполнение любого отдельного этапа затрачивался лишь конечный отрезок времени.

Это утверждение требует, чтобы общее количество этапов в алгоритме было конечным.

Пример: правило вычисления значения 1+1/4+1/9+…— задает бесконечное количество этапов, а правило 1+1/4+1/9+1/16+1/25+1/36+1/49+1/64+1/81+1/100 — задает конечное количество этапов.

Кроме того, нигде не сказано, что этап или группа этапов может быть выполнена только один раз.

Следовательно, какая-либо последовательная группа этапов алгоритма может выполняться *неоднократно*, и требование конечности алгоритма ***исключает*** случай *бесконечного количества выполнений любой группы этапов*.

Рассмотрим пример, в котором последовательность предписанных действий необходимо выполнить несколько раз. Врач выписал больному лекарство, которое: 1)следует купить в аптеке и принимать каждый день 2)утром, 3)в обед и 4) вечером, а после выздоровления больному следует 5)пройти контрольное обследование.

Что можно сказать об этом примере?

Явно видна дискретность этого предписания и конечность количества этапов, из которых оно состоит.

Но из пяти этапов первый и пятый должны быть выполнены только один раз, а вот этапы 2, 3 и 4 необходимо выполнять неоднократно, в течении нескольких дней. Про такие этапы программисты говорят, что они образуют цикл.

Внимательный анализ обсуждаемого предписания показывает, что оно составлено некорректно. В чём этап некорректность?

Если по прошествии нескольких (3-5) дней больной выздоровеет, то согласно указанию врача он может переходить к пятому пункту — прохождению контрольного обследования и в этом случае результат окажется полученным за ***конечное*** количество выполнений 2, 3 и 4 этапов.

А что если больному лучше не станет? Если врач неправильно поставил диагноз?

Если пунктуально подходить к исполнению предписаний алгоритма (а только так и следует к ним подходить), то несмотря на то, что лекарство не помогает и больной не выздоравливает, он должен продолжать приём выписанного лекарства, вообще говоря, *до бесконечности*.

Пациент должен каждый день выполнять этапы два, три и четыре и *никогда* не сможет перейти к выполнению пятого этапа, поскольку условие завершения приёма лекарства не выполнено — выздоровление пациента не произошло.

Такую ситуацию принято называть **бесконечным** циклом. Отсюда следует, что корректный алгоритм не может быть сформулированным таким образом, чтобы возникал бесконечный цикл, чтобы какая-то группа этапов выполнялась *бесконечно* много раз.

Любые современные компьютерные системы требуют не только конечного, но и *явного* задание всех этапов алгоритма.

Пример неявного задания этапов: 1+1/4+1/9+…+1/100 — подразумеваются действия по аналогии.

Детерминированность (определённость, однозначность) алгоритма означает, что алгоритм должен быть построен так, чтобы любой его исполнитель, выполняя действия алгоритма в одних и тех же исходных условиях, всегда получал один и тот же конечный результат.

Не допускается любая неопределенность, неоднозначность, случайность в выполнении действии или определении порядка их выполнения. Каждый следующий этап исполнения, а также все действия следующего этапа алгоритма должны *однозначно* определятся состоянием на текущем этапе.

Рассмотрим в качестве примера алгоритма инструкцию по начислению заработной платы. Допустим, что два разных бухгалтера, *абсолютно точно следуя инструкции* и не совершив ни одной ошибки, выполнили расчёты зарплаты для одного и того же коллектива за один и тот же период и получили при этом разные результаты, различные суммы выплат одному и тому же сотруднику.

Это может означать только одно: инструкция начисления зарплаты составлена таким образом что некоторые её правила воспринимаются разными людьми различным образом. Совершенно очевидно, что такая ситуация неоднозначности недопустима как для инструкций по начислению зарплаты, так и для любых других алгоритмов.

В связи с обсуждением определённости, однозначности алгоритмов, необходимо Отметить, что существуют особые разновидности алгоритмов, которые называются *вероятностными* и *недетерминированными.*

Вероятностными называются алгоритмы, к которых имеются действия, связанные со случайными событиями.

Недетерминированными называются алгоритмы, в которых на некотором этапе их выполнения возникает *несколько различных* вариантов действий. После чего каждый из вариантов выполняется *отдельным исполнителем*.

Потенциальная выполнимость (**результативность**, **направленность**, **завершаемость**) алгоритма подразумевает *направленность*, нацеленность алгоритма на получение результата. Более того, имеется в виду, что исполнение алгоритма всегда должно *завершаться* получением *результата*.

Это в свою очередь означает, что алгоритм может содержать только такие действия, которые исполнитель *в состоянии выполнить*.

Если всё же по смыслу решаемой задачи, или по способу её решения предполагается, что появление невыполнимого действия возможно, то такая ситуация должна специальным образом *контролироваться* и в алгоритме должно быть указано, что в этом случае следует считать результатом исполнения алгоритма.

Например, алгоритм не должен содержать этапов, на которых от исполнителя потребуется выполнить деление на нуль, вычислить логарифм отрицательного числа и т.д.

Наличие, скажем, действия деления в том случае, если знаменатель всё таки может оказаться равным нулю, означает, что в таком алгоритме должна быть *предусмотрена проверка* знаменателя на равенство нулю.

И при обнаружении такого равенства должно быть определено, что считать результатом, а также должны быть указаны дальнейшие действия.

Обычно, в таких ситуациях в алгоритме предусматривается выдача диагностического сообщения с указанием места появления невыполнимого действия, после чего выполнение алгоритма завершается. Результатом исполнения алгоритма при этом считается диагностическое сообщение.

Алгоритм не содержит ошибок, если он даёт правильные результаты для *любых* допустимых исходных данных

Возникновение в алгоритме *неконтролируемых* действий, которые исполнитель не может выполнить, означает наличие в алгоритме ошибок. Кроме того, алгоритм содержит ошибки, если приводит к получению неправильных результатов либо *вообще не даёт результатов*.

Итак, исполнение алгоритма может завершиться за конечное время с получением требуемого результата только в том случае, если: 1) алгоритм не содержит ошибок, 2) алгоритм применяется к корректным, допустимым входным данным и 3) в алгоритме предусмотрена реакция на возникновение невыполнимых действий.

В противном случае исполнение алгоритма за конечное время может не завершится, будет получен неверный результат, либо результат вообще не будет получен, так как исполнение алгоритма прервется невыполнимым действием.

Понятность (**элементарность**) алгоритма предполагает что на каждом этапе правило выполнения действий должно быть относительно простым и локальным, то есть касающимся только данного этапа.

Это требование также означает, что любой этап алгоритма должен содержать только такие действия, способ выполнения которых *известен* исполнителю. Иначе говоря, все предусмотренные в алгоритме действия должны входить в *систему команд* исполнителя.

Кроме того, алгоритм должен быть задан в форме, которая известна исполнителю. Например, для исполнителя-компьютера алгоритм должен быть в конечном счете задан в двоичном алфавите. Получается, что требование понятности, элементарности *относительно*, оно всегда зависит от исполнителя алгоритма.

Массовость (**универсальность**) **алгоритма** означает, что набор исходных величин алгоритма может выбираться из некоторого, возможно, *бесконечного* множества наборов.

Для каждого алгоритма существует некоторое множество наборов величин, которые *допустимы* в качестве исходных данных. Например, в алгоритме деления вещественных чисел делимое может быть *любым* числом, а делитель не может быть равен нулю.

Алгоритм служит, как правило, для решения не одной конкретной задачи, а для решения некоторой группы однотипных задач. Так, алгоритм сложения применим к *любой паре* натуральных чисел. В этом и выражается свойство массовости, то есть возможности применять многократно один и тот же алгоритм для решения любой *аналогичной* задачи.

Интуитивного понятия алгоритма, применяемого совместно с перечисленными требованиями, оказывается достаточно для большинства практических проблем информатики — разработки алгоритмов решения самых разных задач и написания работоспособных программ для компьютера.

С другой стороны, система правил, инструкций, указаний, которая не удовлетворяет этим требованиям, не обладает описанными свойствами, *не может считаться алгоритмом*. Такую систему правил невозможно записать в форме программы, пригодной для исполнения на компьютере.

Алгоритмизация

Использование компьютеров для решения различных задач, обработка любых данных на компьютере могут быть осуществлены только в том случае, если имеется алгоритм решения задачи, алгоритм обработки данных.

Следовательно, возникает необходимость построения алгоритма, который служит для решения поставленной задачи. После того, как алгоритм построен, соответствующую ему программу можно реализовать на любой конкретной модели компьютера.

Изучение и развитие методов и приёмов построения алгоритмов являются содержанием отдельного раздела информатики, который принято называть **алгоритмизацией**.

Построение алгоритма на основе методов предлагаемых алгоритмизацией является *первым* и *обязательным* этапом в процессе разработки программ и решении задач на компьютере.

Модель компьютера или язык программирования заранее могут быть неизвестны, кроме того, может понадобиться перенос реализации с одной модели на другую или же с одного языка на другой.

Поэтому разработку алгоритма предпочитают не привязывать ни к моделям компьютеров, ни к языкам программирования, и построение алгоритма определяется в основном решаемой задачей.

Именно в связи с этим возникло понятие *алгоритмизации*, когда сначала разрабатывается, строится алгоритм, а затем на его основе как результат **программирования** (часто говорят **кодирования**) получают программу для любой модели компьютера или на любом желательном языке.

Способы задания алгоритмов

Построенный алгоритм необходимо некоторым образом фиксировать, записывать просто для его сохранения, для передачи его другому человеку или для предоставления во всеобщее использование, а также для передачи конкретному исполнителю, например, компьютеру, чтобы получать решения конкретных задач.

Алгоритм может быть сформулирован устно, в виде письменного сообщения, задан в виде картинки или какой-либо их последовательности и т.д. Но в любом случае алгоритм представляет собой *сообщение*, зафиксированное в некотором алфавите, а его запись подчиняется определённым правилам.

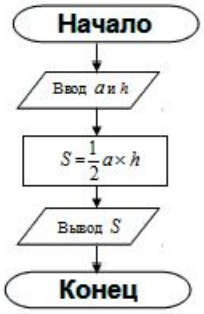
В информатике предъявляются особые требования к заданию алгоритмов в связи с тем, что их исполнителем должен быть компьютер — неодушевлённое устройство, которое не может (*и не должно!*) думать, догадываться, проявлять сообразительность или инициативу.

Поэтому используемые в информатике способы задания алгоритмов должны наделять записываемые алгоритмы всеми необходимыми свойствами, обеспечивать удовлетворение всех предъявляемых к ним требований

Наиболее важные для информатики способы:

- словесный;
- двоичное кодирование;
- графические;
- словесно-формульные.

а)	б)		в)	г)

Чтобы вычислить площадь треугольника S необходимо измерить (взять) длину одной из его сторон a и высоту h, проведенную из вершины противоположного угла к этой стороне. Искомая площадь S равна $\frac{1}{2}a \times h$.	1000 1010 0000 0001
	1111 0110 0000 0001
	1101 0001 0010 1000
	0100 0110
	1000 1001 0000 0001
		mov AX, a
		mul h
		shr AX, 1
		inc di
		mov S, AX
				var a, h, S: real;
				begin
				read(a, h); S:=a*h/2.0; write(S)
				end.

Словесный способ задания алгоритма состоит в его формулировке средствами естественного языка — русского, английского, китайского и т.д. Основное достоинство способа: наиболее приемлемая для человека, естественная форма. Важнейший недостаток — неоднозначность конструкций, предложений, слов естественного языка.

В общем случае алгоритм есть сообщение, заданное в некотором алфавите. Известно, что его можно закодировать в ***двоичном алфавите***. В этом случае может быть получен алгоритм, который может быть использован для *непосредственного* исполнения на компьютере.

Более точно: для исполнения на компьютере алгоритм ***должен*** быть представлен в специальной форме, в которой его этапы имеют вид *двоичных* машинных команд, принадлежащих системе команд процессора этого компьютера.

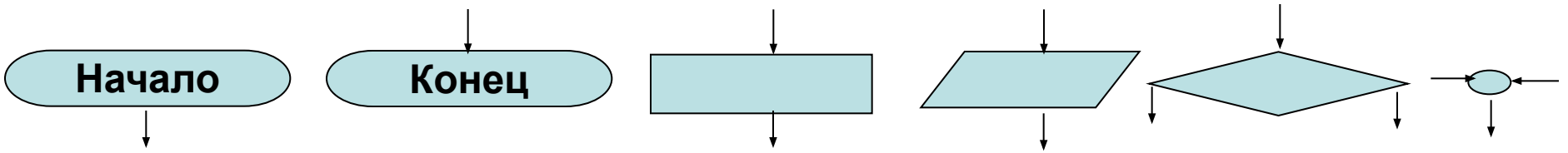
Такую форму записи, обеспечивающую свойство *понятности* алгоритма исполнителю — компьютеру, принято называть **машинным языком** (**машинным кодом**). Алгоритм, представленный на машинном языке, называют **машинной программой**.

Процессор компьютера ничего кроме машинного языка «не понимает».

Недостаток — непосредственное восприятие человеком алгоритма, заданного в двоичном алфавите, затруднено.

Блок-схемой называется графическое изображение структуры алгоритма. Блок – схема представляет собой группу геометрических фигур строго определенного начертания, соединенных стрелками. Каждая фигура изображает некоторое действие, а стрелки, соединяющие фигуры, определяют последовательность действий.

Основные элементы блок-схем



Основными достоинствами этого способа является его простота, гибкость, высокая степень наглядности. Однако, задание достаточно сложных алгоритмов на этом уровне затруднено из-за существенно возрастающей громоздкости. Кроме того, алгоритмы, заданные в таком виде непосредственно не воспринимаются компьютером.

Выявленные особенности блок-схем определяют характер их применения: в настоящее время блок-схемы являются *вспомогательным* средством описания алгоритмов. Они используются, в основном, с целью обучения основам алгоритмизации, в отдельных случаях для проработки особенно сложных участков алгоритмов, с иллюстративной целью, а также для документирования в некоторых технологиях программирования.

Практически сразу после того, как обнаружились проблемы и сложности программирования на машинном языке, а также неприспособленность блок-схем для автоматического перевода на машинный язык специалисты начали такие искать способы задания алгоритмов, которые:

1. более удобны для людей, чем сам машинный язык;
2. не зависят от системы команд конкретных моделей компьютеров;
3. обеспечивают возможность *автоматического* преобразования алгоритмов на машинный язык.

Вначале на этом пути появились очень близкие к машинным языкам **автокоды** (от автоматические коды) и **ассемблеры** (от англ. assemble — сборка, соединение). В этих системах задания алгоритмов для записи команд программы используются не двоичные строки, а некоторые эквивалентные им *словесные* обозначения. Пример таких обозначений можно найти на рисунке, содержащем способы задания алгоритмов.

Основным достоинством таких способов задания алгоритмов является то, что преобразование на машинный язык программ, написанных с использованием словесных обозначений, производится самим *компьютером*, который осуществляет это перевод ***автоматически*** во время выполнения специальных программ-переводчиков.

Автокоды и ассемблеры всегда связаны с машинным языком конкретной модели компьютера, и как только происходит появление новой более мощной модели, приходится, вообще говоря, разрабатывать и новые языки-ассемблеры, и новые программы-переводчики.

Поэтому довольно быстро возникло желание иметь способы задания алгоритмов, *независящие* от систем команд конкретных моделей компьютеров, но сохранившие при этом возможность автоматического перевода на машинный язык *любой модели* компьютера.

Кроме автоматизации перевода выявилась возможность значительно упростить разработку алгоритмов, в которых требуется выполнять множество вычислений по различным математическим формулам. Затем появились другие проблемные области такие как, физика, обработка текстов, экономика, в которых обнаружилась возможность применения компьютеров.

Всё это учитывалось специалистами, и по мере развития способов задания алгоритмов и расширения их возможностей, эти способы всё больше удалялись от машинных языков и приближались к естественным и профессиональным языкам, используемым в различных проблемных областях. Таким образом возникло множество вариантов *словесно-формульного* способа задания алгоритмов, которые впоследствии получили название *алгоритмических языков.*

Словесно-формульный способ базируется на естественном или узко-профессиональном языке. Чтобы избежать неоднозначности, а также обеспечить алгоритм всеми необходимыми свойствами, на запись алгоритма накладывается жесткая система правил. Такие системы правил, которые полностью регулируют все возможности записи алгоритмов образуют ***алгоритмические языки.***

Алгоритмическим языком называется формальная система представления (записи) алгоритмов, которая может быть предназначена для изучения алгоритмов и их свойств, или же обеспечивает возможность *автоматического* преобразования на машинный язык. Алгоритм, записанный на каком-либо алгоритмическом языке, обычно называют **программой**.

Программы, которые осуществляют автоматический перевод, получили название **компиляторов** (англ. compile — компилятор, переводчик), часто их называют ещё и **трансляторами** (англ. translator — переводчик).

Автоматическое преобразование на машинный язык программы, которая перед этим сформулирована на алгоритмическом языке, называется **трансляцией** или **компиляцией**.

Разберём подробнее общую схему использования алгоритмических языков. Для любого использования компьютера должна быть подготовлена программа на *машинном* языке, которая задаёт алгоритм решения задачи, алгоритм обработки данных. Назовём такую программу **целевой**.

Записывать алгоритмы сразу же на машинном языке очень сложно. Поэтому сначала специалист разрабатывает и записывает алгоритм на некотором более удобном, чем машинный *алгоритмическом языке*, получая тем самым **исходную** программу.

Исходную программу компьютер выполнить не может, она написана не в двоичной системе, поэтому сначала её необходимо преобразовать на машинный язык. Для такого преобразования применяются программы-переводчики — *трансляторы*, которые должны быть предварительно установлены на компьютер.

Подчеркнём, что трансляторы — это специализированные машинные программы, то есть программы, которые *уже представлены на машинном языке* и потому они могут непосредственно выполняться процессором компьютера.

Транслятор запускается на выполнение и на его вход передается исходная программа на алгоритмическом языке. Во время выполнения трансляции исходная программа преобразуется (переводится) на машинный язык. Результатом такого *автоматического* перевода как раз и является формируемая на *выходе* транслятора *целевая* машинная программа, которая уже может быть воспринята, «понята» и выполнена процессором компьютера.

Для выполнения целевой программы её необходимо определённым образом передать компьютеру и предоставить ей на вход исходные данные. Результаты выполнения целевой программы в той или иной форме возвращаются специалисту. Такой процесс использования целевой программы можно осуществлять *многократно* для обработки *различных* наборов исходных данных.

Не следует путать *алгоритмический язык* с родственным ему понятием **язык программирования**. Язык программирования это *любой* язык для представления программ, которые сразу же или после ряда преобразований могут быть исполнены компьютером.

Таким образом, все машинные языки автоматически считаются языками программирования. Далее из этого определения следует, что языками программирования могут считаться только те алгоритмические языки, для которых существует возможность их автоматического перевода на машинный язык некоторого компьютера. Это, например, такие языки как Паскаль, Delphi, Си, Java и т.д.

Однако не любой алгоритмический язык может считаться языком программирования, и точно так же не любой язык программирования относится к алгоритмическим.

Можно указать на группу так называемых **императивных** языков программирования, которые *одновременно* являются алгоритмическими языками.

При этом существует много других групп языков программирования: функциональных, логических, скриптовых и т.д., которые построены на абсолютно других принципах по сравнению с алгоритмическими языками, но при этом обеспечивают возможность получения полноценной машинной программы.

Ещё на начальном этапе развития в качестве основы для построения алгоритмических языков были выбраны естественный и математический языки. Близость к естественному языку предоставляет необходимый уровень удобства человеку, а математическая строгость — возможность последующего *автоматического* преобразования на машинный язык.

В этой форме алгоритмы и программы записываются как некоторые, довольно близкие к естественному языку тексты, как последовательности слов и математических формул, поэтому читать, да и создавать их значительно проще, чем машинные программы.

Самое главное в этом способе задания алгоритмов состоит в том, что: текст может состоять только из знаков, входящих в фиксированный алфавит языка, а на запись текста программы накладывается *строгая система правил*.

Важно подчеркнуть, что правила алгоритмических языков, в отличие от грамматических правил естественных языков не *допускают никаких исключений*.

В общем случае система любого языка программирования включает в себя как обязательные элементы:

1. *алфавит*, содержащий множество знаков, которые могут использоваться при записи текста алгоритма (программы);
2. множество правил, обеспечивающих дискретность, конечность, однозначность и понятность алгоритма, а также возможность перевода на машинный язык;
3. комплексы *программ*, которые выполняют *автоматический* перевод исходных программ на машинный язык.

Отметим терминологическую разницу в применении в информатике слов «алгоритм» и «программа». Слово «программа» обычно используется в тех случаях, когда алгоритм записан на машинном языке или на некотором языке программирования. Слово «алгоритм» можно использовать *для всех способов* задания алгоритмов, но обычно этот термин применяется, когда алгоритм задан на естественном языке или на языке блок-схем.

Завершая обсуждение способов задания алгоритмов, сравним запись алгоритма на алгоритмическом языке с другими способами задания:

- по сравнению с естественными языками, алгоритмические языки обеспечивают наличие у алгоритмов всех необходимых свойств, в том числе однозначность, дискретность и т.д., а также возможность автоматического перевода на машинный язык;
- в отличие от машинного языка программа на алгоритмическом языке непосредственно процессором компьютера не воспринимается, нужен дополнительный этап трансляции на машинный язык, но писать и читать программы на алгоритмических языках несоизмеримо легче, чем на машинных языках;

- в отличие от ассемблеров написание программы на высокоуровневых языках ориентировано не на систему команд конкретной модели компьютера, а на проблемную область. Это делает программы машинно-независимыми — одна и та же программа может быть переведена на машинные языки самых разных моделей компьютеров, однако необходимо, чтобы для каждой такой модели была разработана программа-транслятор;
- по сравнению с блок-схемами запись на алгоритмическом языке отличается компактностью, удобочитаемостью, и что самое главное, возможностью автоматического перевода на машинный язык.

Типы алгоритмов

- ***Линейные алгоритмы***. Отличительным свойством линейных алгоритмов является выполнение этапов алгоритма в той последовательности в которой они заданы в алгоритме, без каких-либо условий и повторов.
- ***Ветвления***. Отличительным свойством является наличие в алгоритме хотя бы одного этапа, на котором происходит выбор одного из нескольких возможных дальнейших вариантов выполнения действий. В простейшем случае ветвления в алгоритме предусматривается два варианта.
- ***Циклы***. Отличительным свойством является наличие этапа или группы этапов алгоритма, которые выполняются неоднократно.

Начальные понятия алгоритмических языков

Алгоритмический язык представляет собой систему, которая включает в себя *алфавит*, то есть множество знаков, которые можно использовать при записи текста алгоритма, а также *множество правил*, определяющих **синтаксис** и **семантику** всех возможных конструкций алгоритма.

Из этого определения следует, что в системе правил, совокупность которых, собственно говоря, и образует алгоритмический язык, можно выделить две группы: синтаксические правила, которые определяют возможные способы записи текстов алгоритмов, и семантические правила, связывающие конструкции алгоритма с их смыслом и способом выполнения.

Синтаксис (греч. syntaxes — составлять) языка представляет собой множество правил, определяющих комбинации знаков алфавита, которые считаются для этого языка правильно записанным алгоритмом или же некоторым его правильным фрагментом

Семантика (греч. semanticos — обозначающий) языка представляет собой множество правил, объясняющих смысл и свойства конструкций этого языка.

Следует напомнить, что наличие у алгоритма свойства *однозначного* задания действий, является важнейшим требованием к алгоритму и способу его записи. И как раз семантические правила обеспечивают при исполнении любого такого алгоритма *однозначное* воспроизведение процесса обработки заданных исходных данных.

Каждое *синтаксическое* правило алгоритмического языка представляет собой точное определение, которое должно обеспечивать *однозначное* понимание любых возможных способов построения и записи соответствующей языковой конструкции.

Но сами эти правила также необходимо каким-то образом формулировать, причём способ их задания должен обеспечивать однозначность понимания самих правил. Другими словами нужен язык для описания синтаксиса алгоритмических языков. Такие языки существуют и их принято называть *метаязыками*.

Метаязыком называется система правил описания синтаксиса алгоритмических языков. Эти правила определяют все возможные способы записи любых допустимых конструкций в программах на описываемом алгоритмическом языке

Одним из самых распространённых метаязыков является универсальный язык **БНФ** (от Бекуса Нормальная Форма) или в другом варианте названия язык **формул Бекуса–Наура**. Мы будем использовать этот метаязык в упрощённом виде.

В метаязыке БНФ правило, определяющее способ построения конструкции алгоритмического языка, состоит из двух частей, которые отделяются друг от друга знаком . Это знак понимается как «есть по определению». Слева от него в *уголковых* скобках указывается название определяемой конструкции или понятия, а справа — собственно определение:

<имя понятия> → *определение 1* | *определение 2 …*

Текст определения может состоять из знаков используемого алфавита и других понятий, которые *должны быть* определены в каком-либо другом синтаксическом правиле языка.

При наличии нескольких вариантов определения одного и того понятия варианты отделяются друг от друга вертикальной чертой, которая понимается как союз «*или*».

Во всех правилах действует общее соглашение: названия определяемых или уже определённых понятий заключаются в уголковые скобки, а знаки алфавита — не заключаются.

Разберём пример определения:

<*цифра*>→ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Здесь определяемым является понятие *цифра*, а справа десять различных вариантов его определения. Понимается этот текст следующим образом: «Цифра есть по определению знак 0, либо знак 1, либо знак 2, либо знак 3 и т.д.».

Таким образом, римские цифры I, V, X, а также «цифры» шестнадцатеричной системы счисления A, B, C, D, F и E с точки зрения этого определения не являются цифрами.

Комментарии

Текст программы на любом алгоритмическом языке предназначен не только для передачи его транслятору. Этот текст приходится неоднократно читать, во-первых, самому автору, причём это может потребоваться спустя некоторое, возможно значительное время, когда многое из того, что было сделано ранее уже забылось, а, во-вторых, другим программистам, заказчику работы, а также во многих других случаях.

Поэтому в тексты алгоритмов и программ целесообразно включать различные *пояснения* на естественном языке, которые предназначены *читающим эти тексты людям*.

Формой включения пояснений в текст программы являются **комментарии**, которые представляют собой записанную в соответствии с правилами конкретного алгоритмического языка произвольную цепочку знаков.

В различных языках правила включения комментариев в текст программы различны. Но в любом случае комментарий представляет собой произвольную цепочку знаков, которая некоторым образом выделена из остального текста алгоритма.

Существует два основных способа включения комментариев: однострочный и многострочный:

- //так выглядит однострочный комментарий и в Паскале и в Си
- { так оформляется многострочный комментарий в Паскале }
- (* другой способ оформления многострочного комментария в Паскале *)
- /* а это оформление многострочного комментария в Си */

Присутствие комментариев в алгоритме или программе *не является обязательным*. Однако полное отсутствие комментариев или их включение в избыточном количестве является признаком низкой квалификации программиста. Умение поставить комментарий в нужном месте программы, точно, ясно и коротко его сформулировать говорит о хорошем профессиональном уровне программиста.

Алфавит языка

Алфавит любого языка программирования, как и вообще любой алфавит, представляет собой точно зафиксированное множество знаков

Следует иметь ввиду, что в отличие от естественных языков, в алгоритмических языках, во-первых, знаки алфавита в целом неупорядочены, и, во-вторых, являются знаками в некотором, обобщенном смысле, например, ключевое слово for в алгоритмических языках Паскаль и Си считается одним знаком алфавита.

Различные языки имеют, естественно, различные алфавиты, но в целом, практически любой алгоритмически язык содержит одни и те же *группы* знаков.

Например, в языке Паскаль элементы алфавита называются **основными символами**:

<основной символ>→<буква>|<цифра>|<специальный знак>

Это определение понимается следующим образом: основной символ это либо *<буква>*, либо *<цифра>*, либо *<специальный знак>*. Фактически понятие «основной символ» сводится к другим перечисленным справа понятиям, каждый из них, следовательно, должен иметь своё отдельное определение.

<буква>→ a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

<специальный знак>→<знак арифметической операции>|<знак отношения>|<разделитель>|<ключевое слово>

<знак арифметической операции>→ + | - | * | /

<знак отношения >→ = | < | <= | > | >= | <>

разделитель>→ ␣ (знак пробела) | . | , | ; | : | (|) | [|] | { | } | ↑ | ‘ | :=

<ключевое слово>→ and | array | begin | case | const | div | do | downto | else | end | for | function | goto | if | in | label | mod | nil | not | of | or | packed | procedure | program | record | repeat | set | then | to | type | until | var | while | with

Как видно из этих определений в программах на Паскале невозможно использование каких либо надстрочечных и подстрочечных знаков. Вообще, ни один знак, кроме тех, которые перечислены в этих определениях, не может появиться в программе на этом языке. Однако, что в комментариях, которые предназначены исключительно для человека, можно использовать любой символ, имеющийся на клавиатуре компьютера.

Имя (идентификатор)

Чтобы иметь возможность как-то указывать в алгоритмах и программах, а также в операционных системах, в базах данных и в других информационных системах на нужные объекты и конструкции, их принято определенным образом обозначать, используя для этого *имена.*

Имя или **идентификатор** (от англ. identification — опознание) служит для *обозначения* объектов и конструкций в алгоритмах, программах и различных информационных системах. Имя представляет собой произвольную последовательность букв или цифр, которая обычно начинается с буквы.

В качестве имён могут использоваться, например, такие последовательности знаков: a, i, counter, z56yfw9q% .

Более конкретные правила образования имен регламентируются в описании языков, и в различных языках и информационных системах эти правила различны. В основном эти правила *уточняют* приведённое выше общее определение имени. Подробнее о выборе имен см. параграф 7.4.

Данные

В алгоритмах и программах всегда используются какие-либо данные: исходные данные вводятся, затем они как-то обрабатываются и возникают некоторые промежуточные данные, выходные данные — результаты — выводятся: отображаются на дисплее, печатаются на бумаге и т.д.

Данными называются дискретные сообщения в форме, определяемой правилами используемого способа задания алгоритмов (конкретного языка программирования) или же в форме, понятной исполнителю алгоритма. В важнейшем частном случае данные имеют вид двоичных кодов в одном из используемых в компьютере форматов.

К данным в алгоритмах и программах относятся:

- *непосредственные значения — неизменяемые* объекты, которые представляют *сами себя*;
- *константы* — имена, закрепленные за непосредственными значениями и используемые вместо них;
- *переменные* — объекты, значения которых при необходимости могут быть изменены;
- *значения выражений* и *функций*, которые можно с определённым упрощением рассматривать как математические *соотношения* и *формулы*, определяющие способ вычисления этих значений.

Основная разновидность непосредственных значений — числа, например, -5, 0, 3.1415926 и т.д. Кроме них к непосредственным значениям обычно относятся заключённые в кавычки одиночные знаки из алфавита языка: 'a', '!', … , образованные из таких знаков строки: ‘Введите количество слагаемых', а также логические значения true (истина), false (ложь).

Использование непосредственных значений в алгоритмах может вызвать некоторые неудобства. Если, например, одно и то же значение неоднократно встречается на различных участках алгоритма, его приходится каждый раз его заново записывать, при этом не исключено появление ошибок.

В таких случаях целесообразно закрепить за значением какое-либо имя и указывать это имя везде, где требуется использовать значение. Например, чтобы каждый раз не записывать число 3.1415926, следует закрепить за ним имя, скажем, pi и указывать его вместо числа. Такие имена принято называть **константами**.

Переменные

В действиях, которые задаются в алгоритме, кроме непосредственных значений или заменяющих их констант обычно участвуют величины с *изменяемыми* значениями: скорость движения автомобиля, температура вещества или среды, объём продаваемого товара и т.д.

В информатике для выполнения действий с такими величинами вводится понятие *переменной*.

Переменной называется программная конструкция, которая может: 1) *принимать* значение, 2) *сохранять* его сколько угодно долго без изменения, 3) при необходимости это значение может быть *изменено* требуемым образом.

Переменные в алгоритмах рассматриваются как некоторые *носители значений*, как своеобразные *сейфы* для их хранения.

Чтобы запомнить некоторое значение его закрепляют за выбранной для этой цели переменной — кладут в «сейф».

Переменная сохраняет значение в неизменном состоянии столько, сколько это требуется алгоритму.

Как только по ходу выполнения алгоритма значение почему-либо следует изменить, над переменной выполняют определённое действие, в результате которого её значение заменяется новым — старое значение из «сейфа» удаляется, а вместо него кладётся новое.

Единственный способ сохранить в алгоритме некоторое значение, а потом его как-то использовать в вычислениях состоит в его закреплении за некоторой переменной в качестве текущего значения. Про такое действие говорят: **присваивание** значения переменной.

Из этого обсуждения видно, что понятие переменной в информатике практически аналогично соответствующему математическому понятию. Но в математике рассматриваются, как правило, *числовые* переменные, а вот в информатике природа значений переменных может быть *любой*. Например, множествами возможных значений переменных могут быть названия дней недели, месяцев в году, названия товаров и их цветовых оттенков и т.д.

Переменные в алгоритмах обозначаются именами, характеризуются типом и *имеют*, либо *не имеют* текущего значения

Переменные, не имеющие текущего значения, называются **неопределенными**, и их использование в алгоритмах существенно ограниченно

Тип переменной

Любые данные в компьютере представлены некоторым двоичным кодом и для процессора все они являются *неструктурированной последовательностью* двоичных цифр — бит. Для того чтобы подчеркнуть что ни количество бит в этой последовательности, ни их смысл процессору неизвестны используется термин **поток бит**.

Однако с точки зрения программиста данные в алгоритмах и программах всегда наделены некоторым смыслом, определённым образом *интерпретируются* (англ. interpretation — толкование, объяснение). Скажем, одно и то же число может рассматриваться и как вес товара и как расстояние до некоторой точки.

Любая конкретная интерпретация двоичного кода определяется формой и структурой его представления в программе, которые выбраны программистом. А этот выбор, в конечном счёте, зависит от решаемой задачи и удобства выполнения действий над данными, которые, как видно из примера, могут иметь самую разную природу.

Так возникает проблема выбора формы представления данных в алгоритме, которая в наилучшей степени соответствует поставленной задаче. Решение этой задачи привело к появлению сначала понятия типа данных, а впоследствии и *целостной концепции* типа данных.

Концепция (от лат. *conceptio* — понимание, система) представляет собой определённый *способ понимания* (трактовки, восприятия) какого-либо предмета, явления или процесса, точку зрения на предмет, руководящую идею для его систематического освещения, комплекс связанных между собою и вытекающих один из другого взглядов.

Основной принцип этой концепции заключается в том, что любые используемые в алгоритмах данные, то есть любое значение, любая константа и переменная, а также все используемые выражения и функции *относятся к некоторому вполне определённому* типу

Тип представляет собой важнейшую характеристику данных, используемых в алгоритмах или программах. Тип определяет: 1)множество допустимых значений переменой, 2)множество операций, которые могут выполняться над этими значениями, 3)структуру значения (скаляр, вектор и т.д.), а также 4) способ машинного представления значения.

В какой-то степени тип данных в алгоритмах и программах похож на базовые характеристики физических тел, такие как геометрические размеры и масса, поскольку тип фактически определяет всё, что с данными можно делать, а также как они изображаются, записываются в алгоритмах, в программах и в памяти компьютера.

Различные типы данных имеют различные множества допустимых значений. Так, целые числа в беззнаковом формате с фиксированной точкой могут находиться только в интервале [0, 4294967295], а множество логических значений содержит всего два элемента {true, false}. Этот аспект понятия тип вполне аналогичен множеству допустимых значений аргумента функции.

Над данными разных типов допускается выполнение различных операций. Например, действия сложение и умножение допустимы над числовыми данными, а действия отрицание, конъюнкция и дизъюнкция — над логическими. При этом логические значения складываться и умножаться не могут, а для чисел не определены логические операции.

Значения разной природы имеют различную структуру, они по-разному устроены. Например, такие величины как давление, время, температура полностью задаются одним числом, а чтобы задать вектор, скажем в пространстве, необходимо указать три компоненты, три его координаты, каждое из которых фактически является вещественным числом. Без указания структуры значения невозможно обеспечить правильность выполнения действий над ним.

И, наконец, данные разной природы по-разному кодируются для их представления в памяти компьютера.

Все эти четыре аспекта интегрируются, объединяются в характеристике, которая и называется *тип данного*.

В различных алгоритмических языках используются самые разные наборы типов. Для каждого из них в языках вводятся имена и точно определяются все их характеристики. Четыре обсуждаемых ниже типа целый, вещественный, символьный и логический в той или иной форме включены практически во все современные алгоритмические языки.

Тип *boolean (char, bool)*

Два логических значения **false** и **true**. Причем, **false<true.**
Машинное представление — нулевое и единичное значение бита: false кодируется 0, true — 1. Операции: **¬, ∨, ∧, =, <** и т.д.

В качестве машинного представления этих значений можно было бы использовать состояние одного бита: нулевое значение бита естественно считать кодом значения false, а единичное — кодом значения true. Однако минимальной доступной процессору *самостоятельной* единицей памяти является байт. В качестве кода значения false выбрано нулевое значение байта $0000\ 0000_2$, а кодом значения true считается *любой ненулевой восьмибитный код*.

Логические выражения, то есть выражения, которые имеют значение логического типа, играют важную роль в организации ветвлений и циклов, поскольку входящие в них условия с точки зрения алгоритмического языка как раз и представляют собой *выражения логического типа*.

Тип *integer, int*

В информатике считается, что данные целого типа всегда задаются *абсолютно точно*.
Как правило, в алгоритмических языках предусматривается не один целый тип, а *семейство* целочисленных типов, которые отличаются друг от друга форматом и длиной поля в машинном представлении, и, как следствие, различными диапазонами возможных значений. Эти диапазоны всегда являются подмножествами *математического* множества Z.

Так у целого типа с названиями byte (Паскаль), unsigned char (Си) длина поля равна одному байту, а значение представлено в *беззнаковом* формате с фиксированной точкой, поэтому множество возможных значений принадлежит интервалу [0, 255].

У стандартного целого типа integer (Паскаль), int (Си) длина поля равна двум байтам и используется *знаковое* представление формата с фиксированной точкой, отсюда диапазон значений [–32768, 32767].

Целые числа в программах на алгоритмических языках имеет вид последовательности цифр, перед которой может стоять знак плюс или минус, например: 73, -98, 5, 0, 19674.

Над данными целых типов определены арифметические операции сложения (+), вычитания (-), умножения (*), целочисленного деления (div — Паскаль, / — Си) и остатка от деления (mod — Паскаль, % — Си).

У операций целочисленного деления и остатка от деления оба операнда *обязаны* быть целыми, результат также получается целый.

Для данных целых типов определены все операции сравнения. Результат имеет логический тип.

Тип *real, float*

Важнейшим для информатики является представление о вещественных данных как о результатах всевозможных измерений или же математических операций над измеренными значениями. В связи с этим вещественные данные считаются *принципиально* приближенными, неточными. Именно поэтому множество R^* вещественных чисел информатики *не содержит* целых, которые по своей природе являются точными: множество Z не пересекается с множеством R^*.

Для значений этого типа выделяются поля длиной 4 байта, точность значений равна 7–8-ми десятичным цифрам, а их модули принадлежат диапазону ($1{,}4\times10^{-45}$; $3{,}4\times 10^{38}$).

Внешним признаком вещественного числа в алгоритмических языках является наличие в его записи порядка и/или десятичной точки, которая разделяет целую и дробную части числа. При одновременном отсутствии и порядка и точки число считается целым.

Примеры записи вещественных чисел: 7.3 → 7,3; -5.0→ –5,0; 19E6 → 19×10^{6}, -1.34E-20 → $-1{,}34 \times 10^{-20}$.

Для данных вещественного типа определены все четыре арифметические операции. Результат получается вещественного типа.

Для вещественных типов определены все операции сравнения, но их использование имеет одну особенность.

Несмотря на то, что для вещественных данных разрешается использование операций сравнения «равно» и «не равно», делать это *не рекомендуется*.

Применение отношений «равно» и «не равно» для вещественных данных *некорректно*, так как сравнение приближенных чисел в отношении «равно» обычно даёт результат false, а сравнение в отношении «не равно», как правило, имеет результат true.

Вместо строгих неравенств «равно» и «не равно» для вещественных данных рекомендуется применять нестрогие «больше или равно» и «меньше или равно».

Кроме того, вместо сравнения вещественных переменных на равенство более корректной является проверка их *близости*. Например, вместо равенства x=y лучше использовать неравенство вида |x-y|<eps, где eps — требуемая точность совпадения значений.

Во всех арифметических операциях и сравнениях целые и вещественные данные могут использоваться совместно. Это значит, что один из операндов может быть вещественным, а другой — целым. Эта возможность требует осторожности, так как пренебрежение принципиальным различием между целыми и вещественными данными, которое вызвано их точным и приближённым характерами, влечёт за собой неоднозначность и может привести к серьёзной ошибке во время исполнения алгоритма.

Возьмём, например, отношение 5<>5.0 (Паскаль), 5!=5.0 (Си). обсуждаемое соотношение должно всегда иметь значение false, но скрытые детали реализации трансляторов могут дать *любой* результат вычислений, что влечёт за собой неоднозначность алгоритма, поэтому таких операций следует всемерно *избегать*.

Тип *char*

Непосредственными значениями символьного типа по определению являются одиночные символы текстов. Чтобы отличить однобуквенные *имена*, которые могут использоваться в алгоритмах, от непосредственных значений символьного типа, последние, заключаются в одиночные апострофы. Примеры: **‘a’, ‘!’, ‘5’.**

Множество значений определяется кодовой таблицей и возможностями клавиатуры. Операции: **+, =, <,** и т.д. Машинное представление: текстовый формат ASCII, Unicode.

Структурированные типы данных

Величины, используемые в различных предметных областях, в том числе в технических применениях, математике, физике и т.д., могут иметь различную внутреннюю структуру.

Самыми простыми являются скалярные величины, такие как давление, температура, время. Значения этих величин не имеют *внутренней структуры*, они *неделимы* на составные части. Чтобы полностью задать скалярную величину остаточно указать *только одно* число.

Более сложную структуру имеют, например, вектора. Значение любого вектора состоит из нескольких компонентов — его координат. Например, вектор

$$\bar{x} = (x_1, x_2, x_3)$$

представлен тремя составляющими его значение координатами x_1, x_2 и x_3.

Действия могут выполняться над вектором целиком, то есть над совокупностью всех его координат. Вместе с тем каждая координата может участвовать в вычислениях и *как самостоятельная величина*, а не только вместе со всем вектором

Пусть, например, требуется умножить вектор на некоторое число, тогда в записи операции следует указать только его имя:

$$a\bar{x}$$

Эта запись означает, что на заданное число необходимо умножить *все* координаты вектора .

А можно, например, выяснить знак у какой-то конкретной координаты вектора, тогда с помощью номера указывается только требуемая координата: $x_3>0$, и действие сравнения с нулём выполняется только над ней.

Необходимо понимать, что вектор может иметь только *одно* текущее значение, но это значение состоит из *нескольких* составляющих его элементов, которые могут использоваться в вычислениях как независимо друг от друга, так и совместно

Чтобы иметь возможность работать со структурированными величинами в алгоритмах необходимо точно ***описать*** *внутреннюю структуру* их значения: сколько элементов образует значение, какой тип они имеют, как можно выделить отдельный элемент. Такое описание связано с определением соответствующего типа данных.

Различают **простые** (**скалярные**, **неструктурированные**) и **сложные** (**составные, структурированные**) типы данных.

Значения простых типов данных не имеют никакой внутренней структуры, это неделимые, атомарные значения, в них невозможно выделить какие-либо составные части, которые могут самостоятельно интерпретироваться и использоваться.

Все рассмотренные стандартные типы (целый, вещественный, символьный и логический) относятся к простым, скалярным типам данных, поскольку их значения состоят из *одного* элемента — числа, знака алфавита, логического значения. Переменные простых типов обычно для краткостти называют **простыми переменными**.

Значения сложных, структурированных типов данных *составлены* из *нескольких* значений каких-либо простых типов, они имеют некоторую внутреннюю структуру. Со значением составного, сложного типа можно работать как с единым целым, кроме того каждую его составную часть можно использовать самостоятельно, отдельно, независимо от всех остальных частей.

Значение скалярного типа представлено *ровно одним* компонентом (время, температура), значение структурированного типа представлено *более чем одним* компонентом (вектор, матрица).

Примеры структурированных значений:

$$\overline{x} = \{x_1, x_2, x_3\}; \overline{w} = (w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9); A = \begin{bmatrix} a_{11} a_{12} a_{13} \\ a_{21} a_{22} a_{23} \\ a_{31} a_{32} a_{33} \end{bmatrix}$$

День	Месяц	Год
9	Мая	1945
12	апреля	1961

Структуры аналогичные векторам и матрицам в информатике принято называть ***массивами***, а структуры аналогичные строке таблицы называют ***записями***.

Различные структуры отличаются друг от друга количеством компонентов в значении, способом их обозначения и выборки.

В алгоритмах переменные структурированных типов, как и простые переменные, обозначаются именами. Имя переменной структурированного типа обозначает всю совокупность элементов, из которых состоит значение переменной — все координаты вектора или всю строку таблицы.

Например, имена x и A обозначают весь вектор и всю матрицу соответственно.

Если требуется выполнить действие над элементом значения, его следует указать, выделить.

Для выделения отдельного компонента массива в квадратных скобках справа от названия указывается его номер или несколько номеров через запятую: **w**[5]; w[i+2]; *A*[1,2].

В языке Си, а также во многих родственных языках при указании отдельного элемента матрицы необходимо каждый индекс записывать в *отдельной паре* квадратных скобок: элементу матрицы A_{23} соответствует обозначение *A*[2][3].

Для выделения элементов массивов могут использоваться не только числа, поэтому конструкцию, которая определяет отдельный элемент массива, называют **индексом**. В качестве индекса в общем случае могут использоваться непосредственные значения, *переменные* или *выражения* целого, символьного или логического типов, например, x[k], w[j+3].

Данные типа запись представляют собой *набор значений* возможно разных типов, помещаемых в различные столбцы таблицы. При этом каждый столбец принято называть **полем записи**.

Например, значение даты состоит из трёх полей, названия которых указаны в заголовке таблицы: День, Месяц, Год.

В тех случаях, когда речь идёт о *полной* дате некоторого события следует использовать имя Дата.

А для обозначения отдельных полей, отдельных составляющих записи применяются **составные имена**, которые образуются из нескольких обычных имен *отделяемых друг от друга точкой*.

В рассматриваемом примере чтобы указать какой месяц входит в дату, применяется составное имя Дата.Месяц, а имя Дата.Год служит для указания на входящий в дату год. Для первой строки таблицы имени Дата соответствует полное значение «9 Мая 1945», а составное имя Дата.Месяц имеет своим значением «Мая».

Описания

Еще одним важным принципом концепции типа данных является требование того, чтобы тип непосредственного значения, константы, переменной, выражения или функции определялся по внешнему виду объекта или по его описанию *без выполнения какихлибо вычислений*.

Тип непосредственных значений всегда можно определить по способу их записи в алгоритме. Все допустимые способы их записи регулируются исчерпывающими и точными правилами алгоритмических языков.

Тип констант, вообще говоря, также можно определить по их значениям, но обязательно требуется указание на то, что конструкция является константой.

Для переменных и функций тип должен указываться явно. Для такого рода указаний в алгоритмах используются конструкции, которые называются **описаниями**. В важных частных случаях описание связывает имя константы и её значение, а также имя переменной или функции и её тип.

Описанием называется конструкция алгоритма или программы, в которой зафиксировано имя программного объекта и в зависимости от его природы указаны важнейшие его атрибуты, такие как значение, тип и т.д.

Для паскалеподобных языков характерно размещение всех описаний в начале программы, до первой конструкции, задающей какое-либо действие. В Си-подобных языках описания могут размещаться практически в любом месте программы, потому константы и переменные обычно описывают в той точке программы, в которой они впервые появляются.

В описании константы необходимо указать выбранное для неё имя и значение. Для этого используется следующая конструкция:

<*описание константы*> const <*имя*>=<*непосредственное значение*> (Паскаль)

<*описание константы*> const <*тип*><*имя*>=<*непосредствен. значение*>(Си).

Пусть, например, имеются описания:

const pi=3.1415926 // Па, const float pi=3.1425926 // Си.

Теперь в любом месте алгоритма имя pi представляет число 3.1425926. Удобство применения констант в алгоритмах в том, что для любого непосредственного значения можно подобрать краткое и запоминающееся имя, которое в любом месте представляет это значение, причём изменить указанное в описании значение константы *невозможно*.

Заметим, что в этих определениях предполагается, что *существуют* и определения понятий <*непосредственное значение*>, <*имя*>, <*тип*>. Если их нет, то определение <*описание константы*> окажется некорректным.

Описания переменных простых типов в Паскале имеет вид:

<описание простого типа> <список имен через запятую>: <имя типа>

Эта формула понимается следующим образом: «Описание простого типа есть по определению конструкция, состоящая из одного имени или перечисленных через запятую имён переменных, относящихся к одному и тому же типу, за которым через двоеточие указывается имя типа».

Описания простых переменных в Си определяются немного по-другому:

<описание простого типа> <имя типа> <список имен через запятую>

И в Паскале и в Си несколько подряд расположенных описаний отделяются друг от друга точкой с запятой.

```
y: real;  b: char;  i, j, n, counter: integer;  // Па
```

```
float y;  char a;  int  i, j, n, counter;  // Си
```

В Паскале описание массивов состоит из списка имен тех массивов, которые имеют одинаковую структуру, а за списком после двоеточия записывается ключевое слово array (массив).

Далее в квадратных скобках для вектора указывается диапазон изменения одного индекса, а для матрицы — диапазоны изменений двух индексов через запятую, причем первый диапазон соответствует строкам матрицы, а второй — её столбцам. Диапазон изменения состоит из минимального и максимального значений индекса, которые отделяются друг от друга двумя подряд расположенными точками.

После квадратных скобок следует ключевое слово of (из) и завершается описание указанием типа элементов.

```
x,y: array[1..3] of real; w: array[1..9] of real; A, B, C: array[1..3, 1..3] of integer // Па
```

В языке Си описание массива состоит из имени типа элементов массива, за которым расположено имя массива. Далее для вектора в квадратных скобах указывается количество компонент вектора, а для матрицы записываются две пары таких скобок, в первой паре задается количество строк, а во второй — количество столбцов матрицы.

```
float x[3], w[9]; int A[3][3];  // Си
```

Внимание! Нумерация элементов массивов в языке Си всегда начинается с нуля.

Описания играют очень важную роль, так как транслятор использует их, во-первых, для определения форматов машинного представления данных и, во-вторых, для контроля возможности и правильности выполнения задаваемых в алгоритме операций над ними. Кроме того, транслятор выделяет поле памяти соответствующей типу длины для хранения связанного с именем значения.

Выражения

Основным способом вычисления новых значений в алгоритмах является использование *функций* и *выражений*. Понятие функции в алгоритмических языках практически совпадает с одноименным математическим понятием

Для наиболее часто используемых на практике *элементарных* функций в алгоритмических языках определены имена, типы аргументов и результатов, а также способы вычисления их значений. Такие функции принято называть *стандартными*. Например, для вычисления квадратного корня из некоторого числа можно применить стандартную функцию с названием sqrt (англ. square — квадрат).

Любую стандартную функцию программист может использовать в своих программах без каких-либо дополнительных пояснений. Для этого требуется указать её название, а затем записать её аргумент или список аргументов в круглых скобках. Аргументами функций могут быть непосредственные значения, константы, переменные и выражения.

Имя функции вместе с заключённым в круглые скобки аргументом или списком аргументов называется **указателем функции**

Например: sqrt — это имя функции, а sqrt(4.8) — это указатель функции.

Про результат, получающийся при вычислении значения функции, говорят: **функция возвращает результат**.

Выражения в алгоритмических языках являются практически полным аналогом хорошо известных арифметических и алгебраических выражений. **Выражением** называется текст, задающий правило вычисления значения

В выражения могут входить: непосредственные значения, константы, переменные и указатели функций, соединённые знаками различных операций.

При вычислении значений выражений действует порядок старшинства операции, который в разных алгоритмических языках определяется по значительно отличающимся друг от друга правилам.

Но в любом случае эти правила обеспечивают стандартный математический порядок старшинства — сначала выполняются действия умножения и деления, а потом умножения и сложения.

Операции одинакового старшинства всегда выполняются слева направо. При необходимости изменить порядок действий, диктуемый правилами старшинства операций, используются *круглые скобки*, причём глубина вложения скобок друг в друга не ограничена.

Правила записи выражений в разных алгоритмических языках достаточно сильно разнятся, но вместе с тем действуют некоторые общие для всех языков правила:

1. запрещаются «многоэтажные» конструкции, а также любые надстрочечные и подстрочечные знаки;
2. знак операции умножения * пропускать и заменять его точкой ·, как это делается в математических текстах, нельзя;
3. аргументы функций заключаются в круглые скобки;
4. в выражении должен соблюдаться баланс скобок, то есть количество открывающих скобок должно быть в точности равно количеству закрывающих скобок и т.д.

Пусть требуется записать в программе выражение соответствующее дроби:

$$\frac{a+b}{c-d}$$

Поскольку «многоэтажные» конструкции в алгоритмических языках запрещены, соответствующее обсуждаемой дроби выражение должно быть записано в одну строчку: a+b/c+d.

Но эта запись ошибочна, так как с учётом старшинства операций получим, что эта строка задаёт такие вычисления:

$$a + \frac{b}{c} - d$$

Необходимый порядок действий можно задать только применив круглые скобки: (a+b)/(c-d).

Ещё одно важное требование: значения всех переменных, входящих в выражение, должны быть определены до начала вычисления его значения.

При записи выражений необходимо также следить за типами, входящих в выражение операндов: они должны обеспечивать возможность выполнения вычислений. Например, запись 'a'+4 является ошибочной, так как невозможно сложить букву с целым числом.

Отношение в алгоритмических языках представляет собой *два* выражения одного и того же *простого* типа, которые связаны *одним* знаком отношения.

Примеры отношений: k > i–1, j<=3, x >= sqrt(r*r – y*y)

При записи в алгоритмических языках неравенств вида $0<x\leq 1$ довольно часто допускаются ошибки, вызванные стремлением записать полностью аналогичное логическое выражение: 0<*x*<=*1*. В чём здесь ошибка?

Ошибка здесь в том, что отношение может включать только *два* выражения и только *один* знак отношения

Неравенства вида следует трактовать как два неравенства $x>0$ и $x\leq 1$, которые должны выполняться *одновременно*, то есть, связаны логической операцией конъюнкция: $x>0 \wedge x\leq 1$.

Правильный вариант записи: (x>0) and (x<=1) //Па (x>0) && (x<=1) //Си

Операторы и управляющие конструкции

Алгоритм задает, определяет *действия*, которые необходимо выполнить для достижения цели, а также фиксирует требуемую *последовательность их выполнения*.

Основной конструкцией, с помощью которой в алгоритмах и программах задаются любые действия, является **оператор**.

Оператором называется самостоятельная, независимая от других конструкция, определяющая некоторый набор действий.

Все рассмотренные выше понятия и конструкции в этом смысле играют вспомогательную роль, так как они либо являются составными частями операторов, либо служат для правильной интерпретации действий, задаваемых в операторах.

В различные алгоритмические языки включены разные наборы операторов. Но этот набор всегда подбирается таким образом, чтобы с их помощью можно было построить любые возникающие при решении практических и теоретических задач алгоритмы.

Для определения требуемой *последовательности действий* в алгоритмах и программах используются управляющие конструкции.

Конструкция алгоритмического языка или блок-схемы, которая обеспечивает требуемый *порядок* выполнения действий, называется **управляющей конструкцией**.

Собственно говоря, существует три типа управляющих конструкций, ровно столько же сколько существует различных типов алгоритмов: это конструкции, которые применяются при построении линейных алгоритмов, алгоритмов с ветвлением и циклов.

Отметим, что управляющие конструкции являются более общим понятием, чем оператор. Одна и та же управляющая конструкция может быть задана несколькими различными операторами.

Присваивание

Закрепление за переменной некоторого текущего значения, также как и любое изменение этого значения, осуществляется в алгоритмах с помощью действия, которое называется ***присваиванием***.

Присваивание нового значения выполняется независимо от того, определена или нет переменная, которой производится присваивание.

Если переменная была определена, то есть у неё было *старое* значение, то оно безвозвратно теряется, и установить каким оно было невозможно.

Синтаксис оператора присваивания в *простейшем* случае определяется следующим образом:

<*оператор присваивания*> → <*Имя*> := <*Выражение* > (Паскаль)

<*оператор присваивания*> → <*Имя*> = <*Выражение* > (Си)

В общем случае в *левой* части оператора присваивания может находиться не только имя простой переменной, но и элемент массива, а также некоторые другие связанные со сложными типами данных конструкции, которые мы будем для простоты называть **элементами структуры**.

Справа в операторе присваивания может находиться любое выражение, в том числе непосредственное значение, переменная, указатель функции и т.д.

Между левой и правой частью оператора находится знак присваивания, в качестве которого в Паскале используется знак :=, а в Си — знак =.

Заметим, что присваивание может изображаться и другими знаками, в частности, в языке Бейсик для этого используется ключевое слово Let, а в языках ассемблера — mov (от англ. move — движение, передача в другие руки).

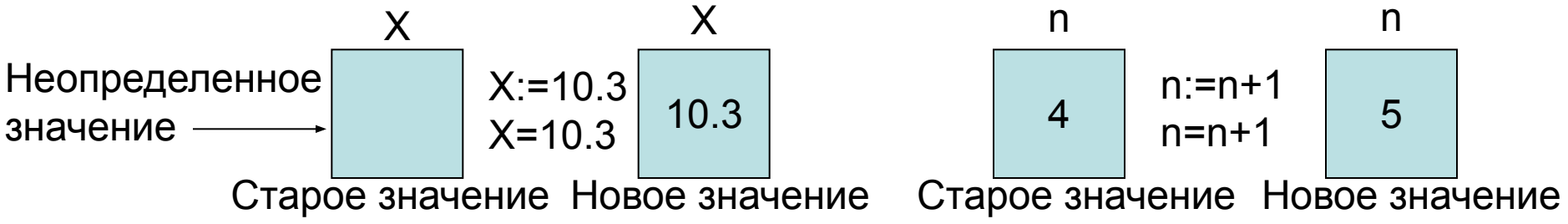
Примеры присваиваний:

counter:=0; X:=10.3;Y:= 2.0*X+1.0; n:=4; n:=n+1; w[n+2]:=0.8; //Па

counter=0; X=10.3;Y= 2.0*X+1.0; n=4; n=n+1; w[n+2]=0.8; //Си

В действии присваивания необходимо различать два состояния одно –до начала выполнения действия, второе – после его завершения. Никакие промежуточные состояния не вводятся и не обсуждаются.

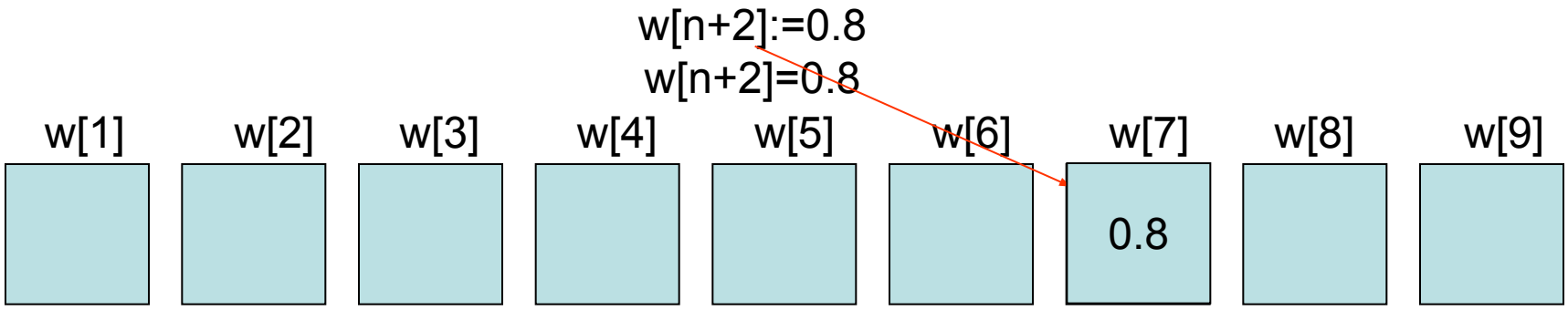
1. До начала действия переменная из левой части имеет старое значение или не имеет никакого;
2. После завершения действия она всегда имеет новое текущее значение.



Порядок выполнения присваивания

1. вычисляется значение выражения в правой части, при этом всегда берутся значения входящих в выражение переменных, которые они имели *до начала выполнения* присваивания;
2. для сложных типов данных определяется, какой именно элемент структуры задан в левой части, возможно, при этом вычисляются значения индексных выражений;
3. вычисленное значение закрепляется за указанной в левой части переменной или за элементом структуры в качестве нового текущего значения.

Действие, задаваемое оператором n:=n+1// Па, n=n+1; // Си, состоит в *увеличении текущего значения переменной* n *на единицу*. Такой оператор и аналогичные ему встречаются в алгоритмах очень часто, поэтому в языке Си имеется короткая эквивалентная форма его записи n++;



Выполнение действия существенно зависит от текущего значения ***n***. Так при ***n=5*** значение индексного выражения определяет, что получить значение ***0.8*** должен седьмой элемент массива ***w***.

Правила задания присваивания

1. Тип переменной в левой части и тип значения в правой должны соответствовать друг другу, например, совпадать. Имеются и другие случаи соответствия, зависящие от используемого языка программирования.
2. Все переменные в правой части должны иметь типы, обеспечивающие возможность вычисления значения выражения.
3. Все переменные, используемые в правой части, а также в индексных выражениях должны быть определены к моменту выполнения присваивания.

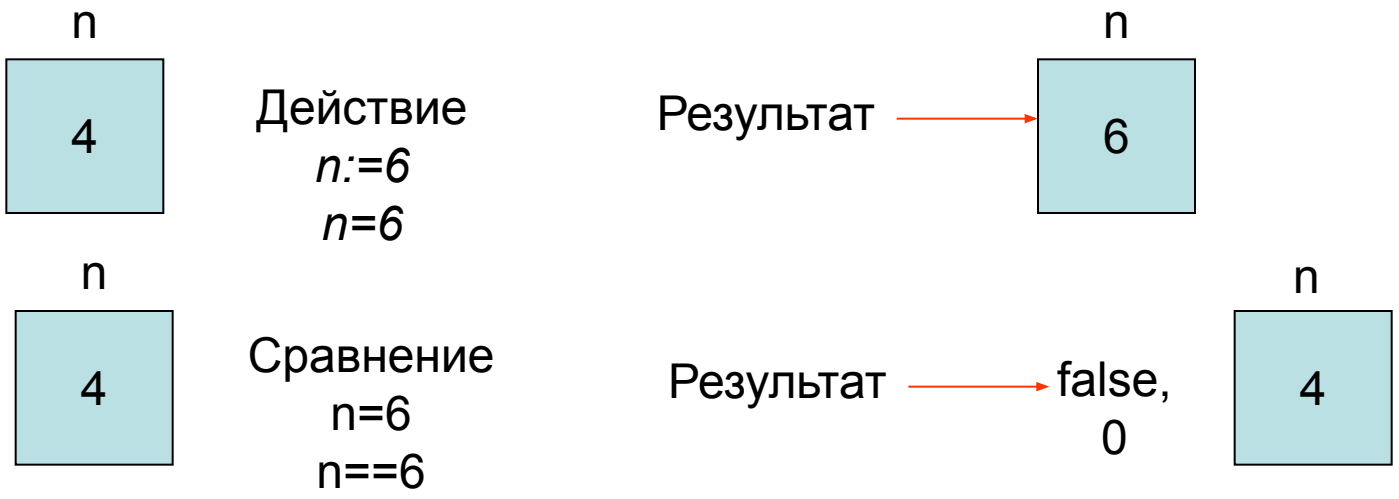
Неопределенные переменные или элементы структуры можно использовать только в левой части оператора для присваивания им начального значения.

X : real: C : char; I,J : integer; X:=0.8; C:='&'; I:=6; X:='&'; C:=6; I:='&'; X:=C+I;

J:=2*J+I

Отличия действия присваивания от операции сравнения

Присваивание это действие, в котором переменная изменяет значение, ***сравнение*** это операция, которая не изменяя значений устанавливает факт равенства или не равенства правой и левой частей.



В операции сравнения левая и правая части равноправны, а в действии присваивания — не равноправны. В присваивании выражение слева писать нельзя, а в сравнении — можно.

n:=6; 6:=n; y:=2*x+1; 2*x+1:=y; n:=5; j:=8; n:=j; n:=5; j:=8; j:=n

n=6; 6=n; y=2*x+1; 2*x+1=y; n:=5; j:=8; n=j; n:=5; j:=8; j=n

Организация обмена

В подавляющем большинстве случаев исполнителем алгоритма является компьютер, а результаты его работы требуются людям. Поэтому для полноценного формирования алгоритма кроме задания собственно вычислительных действий в нём необходимо предусмотреть действия, связанные с передачей исполнителю значений всех исходных величин, а также действия, связанные с получением от исполнителя искомых результатов.

Первая группа действий называется *вводом исходных данных*, а вторая — *выводом результатов*. А когда безразлично, о чем идёт речь используется термин *обмен*.

Организация действий по обмену данными *в реальных программах* довольно сложна, поскольку программисту требуется множество различных возможностей для обеспечения эффективного и удобного ввода, а также хорошо оформленного вывода

Тем более, что любые действия по обмену данными связаны с конкретными особенностями аппаратных и программных средств модели компьютера, на которой фактически должно производиться выполнение вычислений.

Чтобы учитывать такую связь, обмен в алгоритмических языках осуществляется с помощью особой разновидности операторов, которые называются **процедурами обмена**, а точнее **операторами вызова процедур обмена**.

Эти процедуры как раз и связаны и с аппаратурой компьютера и с языковыми конструкциями программы.

Действия по обмену данными *в алгоритмах* будем записывать с помощью *условных* процедур, которые имеют некоторое *сходство* с реальными. Так, для ввода данных мы будем применять несуществующие в реальных языках процедуры:

read(список вводимых величин) // Па, scan(список вводимых величин) // Си

а для вывода процедуры —

write(список выводимых величин) // Па, print(список выводимых величин) // Си

в которых вводимые/выводимые величины представлены в виде списка, состоящего из имён этих величин, перечисляемых через запятую.

Предполагается, что вводимые значения набираются на клавиатуре компьютера, а результаты выводятся на его дисплей.

Например, процедура read(y) scan(y) требует ввода одного вещественного числа — значения переменной y, для процедуры read(i, j) scan(I,j) на клавиатуре следует набрать два целых числа, а для процедуры read(A) scan(A)— девять вещественных чисел, являющихся значениями элементов матрицы A.

Общая структура программы

Структура программы — алгоритма, заданного на алгоритмическом языке, — в значительной мере зависит от используемого языка.

Программа на языке Паскаль в простейшем случае имеет вид:

var O1; O2;…;Om begin S1; S2; …; Sn end. // Па

То есть она содержит последовательность описаний O1;O2;…;Om используемых в программе переменных, и последовательность операторов S1;S2; …;Sn, которые задают нужные действия, здесь O и S — условные обозначения описания и оператора соответственно.

Любые два рядом стоящие описания или оператора разделяются точкой с запятой. Перед последовательностью описаний находится ключевое слово var (от англ. variable — переменная), последовательность операторов находится между ключевым словом begin (англ. — начало) и ключевым словом end (англ. — конец), а завершается текст программы точкой.

Программа на языке Си в простейшем случае имеет вид:

main ( ) { S1; S2; …; Sn; } // Си

То есть она состоит из заключённой в фигурные скобки *одной* последовательности, которая включает в себя и описания, и операторы.

Более того, в языке Си *описания считаются разновидностью операторов*, поэтому описания могут размещаться вперемешку с операторами, но всё-таки большая часть переменных описывается в начальной части программы.

В простейшем варианте написания программы на языке Си перед фигурными скобками указывается имя main (англ. — главный) и пара круглых скобок, конструкции, являющейся частным случаем так называемого **заголовка функции**, который состоит из имени и заключаемого в круглые скобки **списка параметров**

Попутно отметим, что пара ключевых слов begin end в Паскале и пара фигурных скобок { } в Си называется **операторными скобками**, а конструкции begin S1; S2; …; Sn end и { S1; S2; …; Sn; } — **составным оператором**.

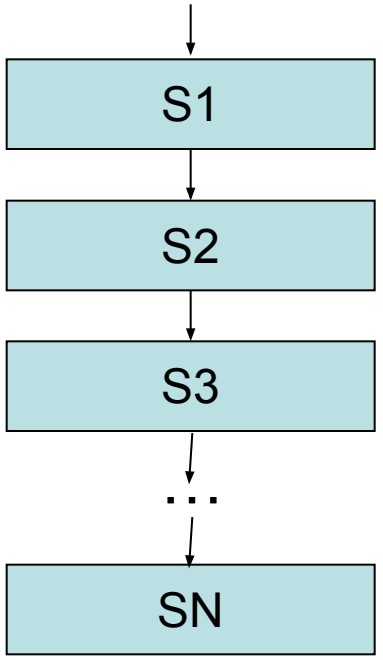
Важно! Заключение последовательности операторов в операторные скобки превращает эту последовательность с точки зрения синтаксических правил языка в *один* оператор

Процедуры ввода/вывода могут помещаться в любом месте внутри последовательности S1;S2; …;Sn операторов программы. Однако в стандартном случае процедуры ввода помещают вначале, а процедуры вывода – в конце этой последовательности.

Линейные алгоритмы. Управляющая конструкция следование.

Для реализации линейных участков алгоритмов используется управляющая конструкция ***следование***, представляющая собой линейную последовательность действий присваивания или других действий, которые выполняются без каких бы то ни было условий или повторений.

S1; S2; S3; … ;SN

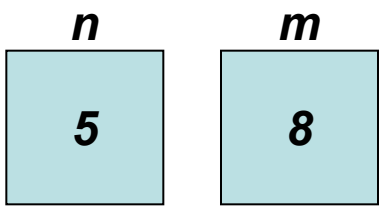


Построение линейных алгоритмов обычно не вызывает затруднений, и состоит из следующих шагов:

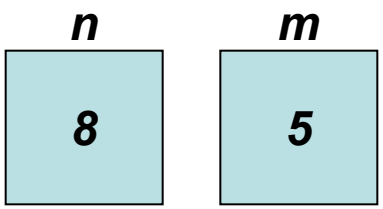
1. необходимые действия определяются путем разбиения всего процесса достижения поставленной цели на отдельные самостоятельные по смыслу достаточно крупные этапы, *не содержащие* условий и повторов;
2. количество этапов не должно быть особенно большим, обычно выделяется 5–7 этапов. Выделение большого количества этапов влечет за собой потерю наглядности, понятности разрабатываемого алгоритма;
3. выявляется и фиксируется порядок их следования в алгоритме;
4. при необходимости каждый этап уточняется, разбивается на *линейные* последовательности всё более и более мелких этапов до тех пор, пока алгоритм не окажется состоящим из простейших, *понятных* исполнителю действий;
5. если в процессе построения на любом его этапе выявляется наличие условий или необходимость в повторах, то алгоритм не относится к линейным, и его необходимо строить по другим правилам.

Пример. Обмен значениями двух переменных

Исходное состояние

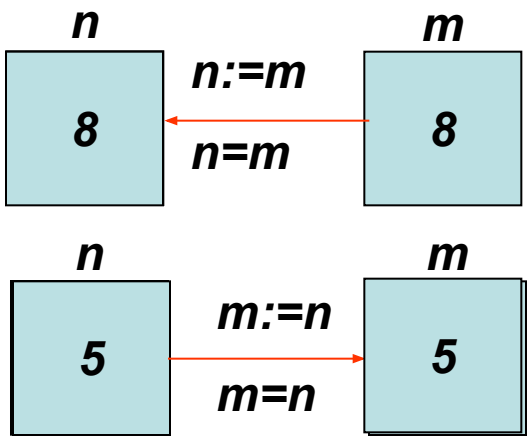


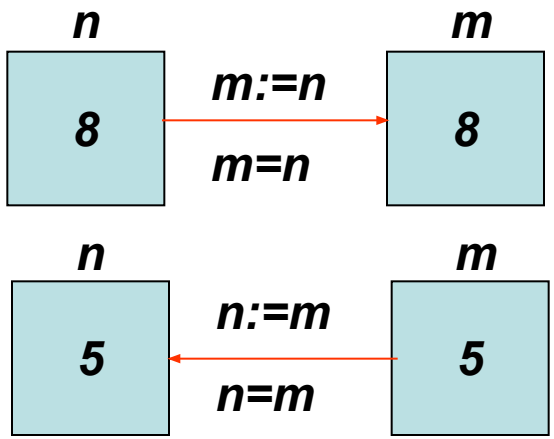
Требуемое результирующее состояние



Переменная ***n*** должна получить новое значение, равное старому значению переменной *m*, в свою очередь переменная ***m*** должна получить новое значение, равное старому значению переменной ***n***,

Буквальная реализация условия задачи





Решение задачи обмена значениями

Необходимо понять причину неудачи предыдущих вариантов решения.

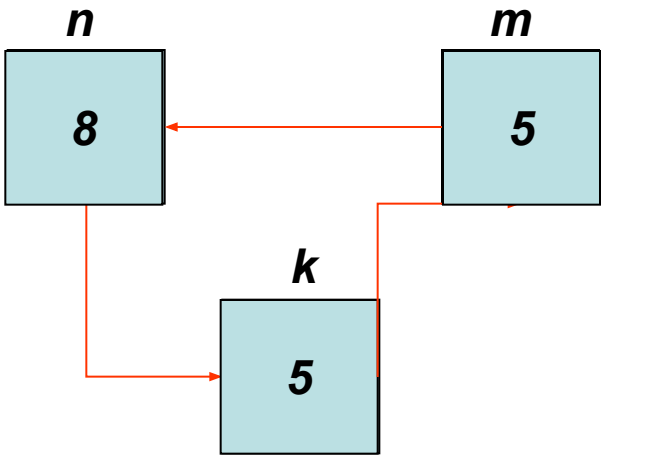
Дело в том, что при выполнении любого присваивания старое значение переменной безвозвратно теряется.

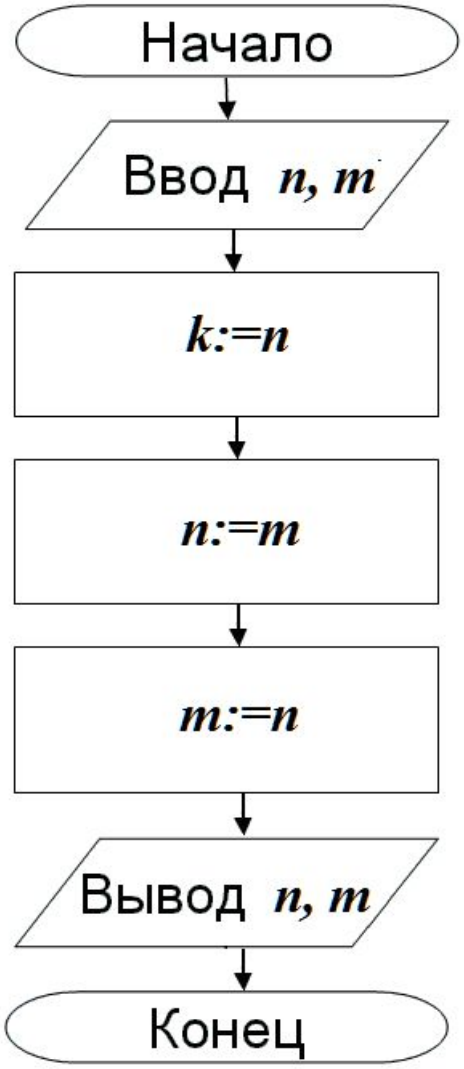
Вывод: чтобы решить задачу старое значение любой из переменных перед первым же присваиванием следует где-то сохранить. Такое сохранение в алгоритме может быть выполнено только с помощью присваивания этого значения какой-либо переменной.

Так как исходные переменные для этого не годятся, следует использовать вспомогательную, дополнительную переменную.

k=n; ***n:=m;*** ***m:=k;***

k=n; ***n=m;*** ***m=k;***






```
//Паскале-подобный алгоритм обмена,
var n, m, k: integer; //Описания используемых переменных
   begin
      read(n, m); //Ввод исходных данных
      k:=n; n:=m; m:=k; // Обмен
      write (n, m) //Вывод результатов
   end.
//Си-подобный алгоритм обмена
main ( ) //Обязательный заголовок
   {
     int n, m; //Описания основных переменных
     scan (n, m); //Ввод исходных данных
        int k=n; n=m; m=k; //Описание промежуточной
//переменной k и обмен
     print (n, m); //Вывод результатов
   }
```

Общий порядок построения алгоритма

- внимательно проанализировать условие задачи выявить, что задано, какие величины являются исходными;
- выяснить что именно необходимо получить, какие величины являются искомым результатом;
- выбрать для каждой величины какое-либо имя, определить и зафиксировать в соответствующем описании её тип;
- выбрать или разработать способ решения задачи;
- в соответствии с этим способом определить какие действия, и в каком порядке необходимо выполнить, чтобы получить требуемый результат;
- зафиксировать действия и порядок их выполнения выбранным средством записи алгоритма (блок-схема, алгоритмический язык);
- выполнить проверку алгоритма, проанализировать полученный алгоритм с точки зрения его правильности и эффективности.

$$\begin{cases} a_{11}x + a_{12}y = b_1 \\ a_{21}x + a_{22}y = b_2 \end{cases}$$

Шаги 1-3. Когда говорят «задана система алгебраических уравнений» это означает, что коэффициенты при неизвестных a_{11}, a_{12}, a_{21}, a_{22} и коэффициенты b_1, b_2 в правой части *известны*.

Исполнителю алгоритма должны быть переданы их *численные* значения, чтобы он имел возможность выполнить вычисления и найти искомые неизвестные.

Следовательно, все эти коэффициенты являются *исходными* величинами строящегося алгоритма — набор их значений является его *входом*.

По условию задачи также понятно, что *искомыми* результатами являются неизвестные *x* и *y* — их значения являются *выходом* алгоритма.

По общему смыслу задачи можно считать, что все коэффициенты системы, а также неизвестные относятся к *вещественному* типу, хотя во многих частных случаях они могут быть и целыми. Для определённости выберем вещественный тип, как более общий вариант.

В данной задаче символьный, логический, а также структурированные типы для обсуждаемых величин не допустимы, поскольку над ними требуется выполнять операции сложения и умножения

В качестве имён искомых неизвестных можно выбрать их названия x и y, использованные в математической постановке задачи.

Проблема возникает только с выбором имен и структуры данных для коэффициентов системы. Можно предложить два варианта такого выбора. В первом варианте каждый коэффициент считается отдельной *простой* переменной вещественного типа.

Тогда подходящим может быть, например, такой выбор имён: a11, a12, a21, a22, b1 и b2.

Второй вариант основан на использовании сложной структуры данных — массива. В этом варианте коэффициенты левой части системы образуют матрицу A и тогда они обозначаются: A[1,1]; A[1,2], A[2,1], A[2,2], а коэффициенты правой части рассматриваются как элементы вектора b и обозначаются b[1] и b[2] соответственно.

Ответ на вопрос, какой из рассмотренных вариантов предпочтительнее зависит от размерности системы уравнений. Для системы из двух уравнений оба варианта примерно одинаковы, но при возрастании размерности для организации вычислений придётся применять циклы и тогда единственным разумным вариантом является использование массивов.

Шаг 4. Решить систему — это значит найти *её корни*, то есть такие значения неизвестных величин x и y, при которых оба её уравнения превращаются в тождества. Каким способом можно решить эту систему?

Решение системы можно получить по правилу Крамера:

$$x = \frac{b_1 a_{22} - b_2 a_{12}}{a_{11} a_{22} - a_{21} a_{12}}, y = \frac{b_2 a_{11} - b_1 a_{21}}{a_{11} a_{22} - a_{21} a_{12}}$$

Шаги 5–6. Алгоритм определения корней складывается из ввода исходных данных, двух операторов присваивания, которые соответствуют приведённым соотношениям и вывода полученных результатов.

Вспомним сказанное раньше: обычно ввод организуют в самом начале алгоритма, а вывод — перед его конечной точкой.

А какие величины из использованных в рассуждениях подлежат вводу и выводу?

Для ответа на этот вопрос можно использовать следующие соображения:

- Следует вообразить себя исполнителем и попытаться мысленно или явно выполнить действия алгоритма, метода на уровне числовых значений. Тогда станет понятно, численные значения каких величин необходимо знать, чтобы фактически произвести вычисления.
- Важное отличие между исходными и промежуточными величинами состоит в том, что исходные величины могут принимать достаточно произвольные значения, в то время как значения промежуточных величин жестко привязаны к значениям исходных, вычисляются через них, и потому они произвольно менять свои значения не могут.

Теперь обсудим собственно вычисления.

Легко заметить, что дроби, значения которых необходимо вычислить, имеют совершенно одинаковые знаменатели, и, следовательно, не имеет смысла вычислять их два раза для каждой из дробей в отдельности. Очевидно, что используя промежуточную переменную можно вычислить знаменатель только один раз.

Чтобы не заставлять компьютер несколько раз вычислять значение одного и того же выражения во всех аналогичных случаях целесообразно вводить промежуточные переменные и запоминать вычисленное значение с их помощью. Этот приём носит название **экономии вычислений.**

Выберем для промежуточной переменной, имеющей смысл знаменателя рассматриваемых дробей, имя *D*. Поскольку для коэффициентов системы выбран вещественный тип, эта переменная также должна относиться к вещественному типу.

$$D = a_{11}a_{22} - a_{12}a_{21}$$

Промежуточные переменные полезно вводить не только для экономии выражений, часто они используются для упрощения чтения и понимания текста алгоритма. В этом смысле можно ввести промежуточные переменные, обозначающие числители каждой из дробей, например *Dx* и *Dy*.

$$Dx = b_1a_{22} - b_2a_{12}, Dy = b_2a_{11} - b_1a_{21}$$

После этого вычисление корней сведется к очень простым действиям:

$$x = Dx / D, y = Dy / D$$

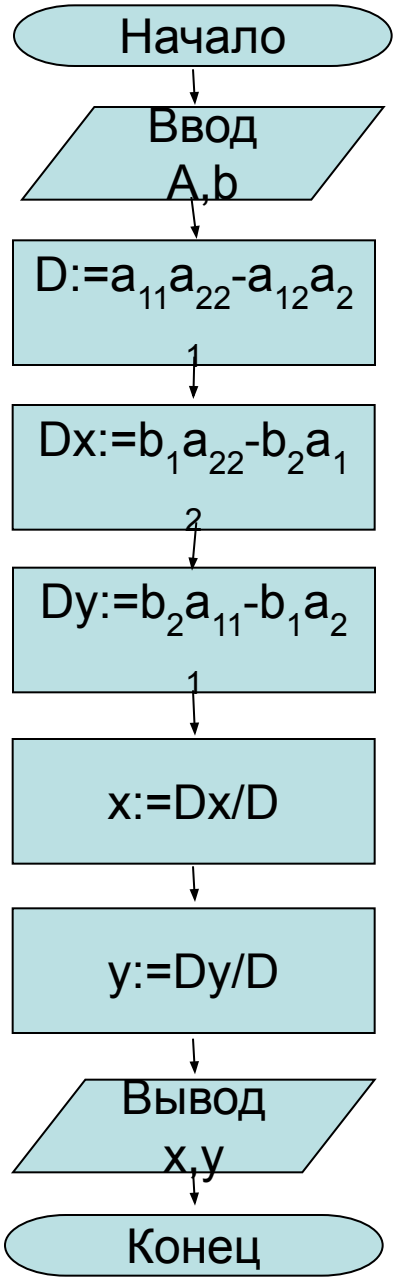
Осталось определиться с порядком выполнения вычислений. Поскольку для нахождения как *x*, так и *y* необходимо знать значение переменной *D*, его следует вычислить сразу же после ввода исходных данных.

Порядок остальных вычислений жёстко не зафиксирован. Например, после вычисления знаменателя можно найти числитель первой дроби, затем числитель второй дроби, после чего уже вычислить корни.

В рассматриваемом алгоритме порядок выполнения действий ***в определенных пределах*** можно изменять и это не повлияет на конечный результат.

Например, сначала можно найти числитель второй дроби, а затем первой, можно далее вычислить *y*, а затем уже *x* и т.д.

Но в любом случае значение переменных необходимо вычислять до их использования в каких-либо выражениях.




```
A : array [1..2,1..2] of real; b:array[1..2] of real;
x, y, D, Dx, Dy: real;
begin
      read (A,b);

      {вычисление главного детерминанта системы}
      D:=a[1,1]*a[2,2]-a[1,2]*a[2,1];

      {вычисление вспомогательного детерминанта Dx }
       Dx:=b[1]*a[2,2]-b[2]*a[1,2];

      {вычисление вспомогательного детерминанта Dy }
      Dy:=b[2]*a[1,1]-b[2]*a[2,1];


      x := Dx/D;


      y:= Dy/D;


      write (x,y)

end.
```

```
// Си-подобный
main ( )
{
    float A[2][2], b[2], x, y;
// Ввод всех элементов массивов A  и b
    scan(A, b);
    float  D=A[0][0]*A[1][1]–A[0][1]*A[1][0];//Знаменатель
    float  Dx=b[0]*A[1][1]–b[1]*A[0][1];//Числитель для x
    float  Dy=b[1]*A[0][0]–b[0]*A[1][0];//Числитель для y
     x = Dx/D; y = Dy/D;   //Корни системы
    print (x, y);
}
```

```
// Паскале-подобный.
var a11,a12,a21,a22,b1,b2,x,y,D,Dx,Dy:  real;
begin
    read(a11, a12, a21, a21, a22, b1, b2);
    D := a11*a22 – a12*a21; //Знаменатель
    Dx := b1*a22 – b2*a12; //Числитель для x
    Dy := b2*a11 – b1*a21; //Числитель для y
    x := Dx/D; y := Dy/D;   //Корни системы
    write (x, y);
end.
```

```
// Си-подобный.
main ( )
{
   float  a11, a12,  a21, a22, b1, b2, x, y;
   scan(a11, a12, a21, a21, a22, b1, b2);
   float  D = a11*a22 – a12*a21; // Знаменатель
   float  Dx = b1*a22 – b2*a12; //Числитель для x
   float  Dy = b2*a11 – b1*a21; //Числитель для y
   x = Dx/D; y = Dy/D;   //Корни системы
   print (x, y);
}
```

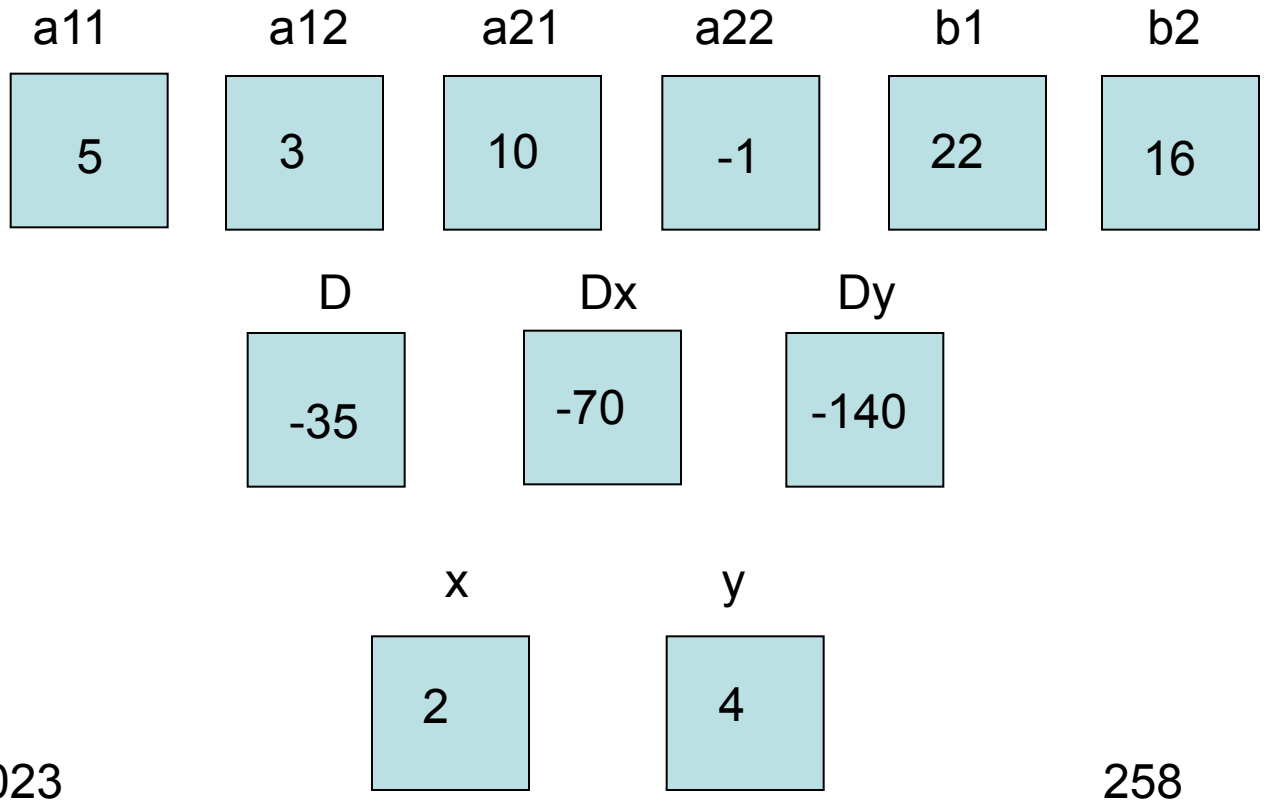
При записи текстов алгоритмов применяются простые графические приёмы: связанные по смыслу строки сдвигаются вправо на одинаковое количество позиций, операторы сдвигаются вправо относительно операторных скобок и записываются с вертикальным выравниванием, в одной строке можно записывать более одного оператора и т.д. Эти приёмы определённым образом структурируют текст и облегчают восприятие алгоритма при чтении. Любой алгоритм рекомендуется оформлять с использованием таких приёмов.

Шаг 7. Тестирование алгоритма следует проводить для системы уравнений, решение которой известно. Возьмем, например, в качестве тестовой такую систему

$$\begin{cases} 5x + 3y = 22 \\ 10x - y = 16, \end{cases}$$

Её корнями являются x=2 и y=4 , в чём можно убедиться непосредственной подстановкой в систему.

```
read(a11, a12, a21, a21, a22, b1, b2);
  D := a11*a22 – a12*a21; //Знаменатель
  Dx := b1*a22 – b2*a12; //Числитель для x
  Dy := b2*a11 – b1*a21; //Числитель для y
  x := Dx/D;
  y := Dy/D;   //Корни системы
write (x, y);
```



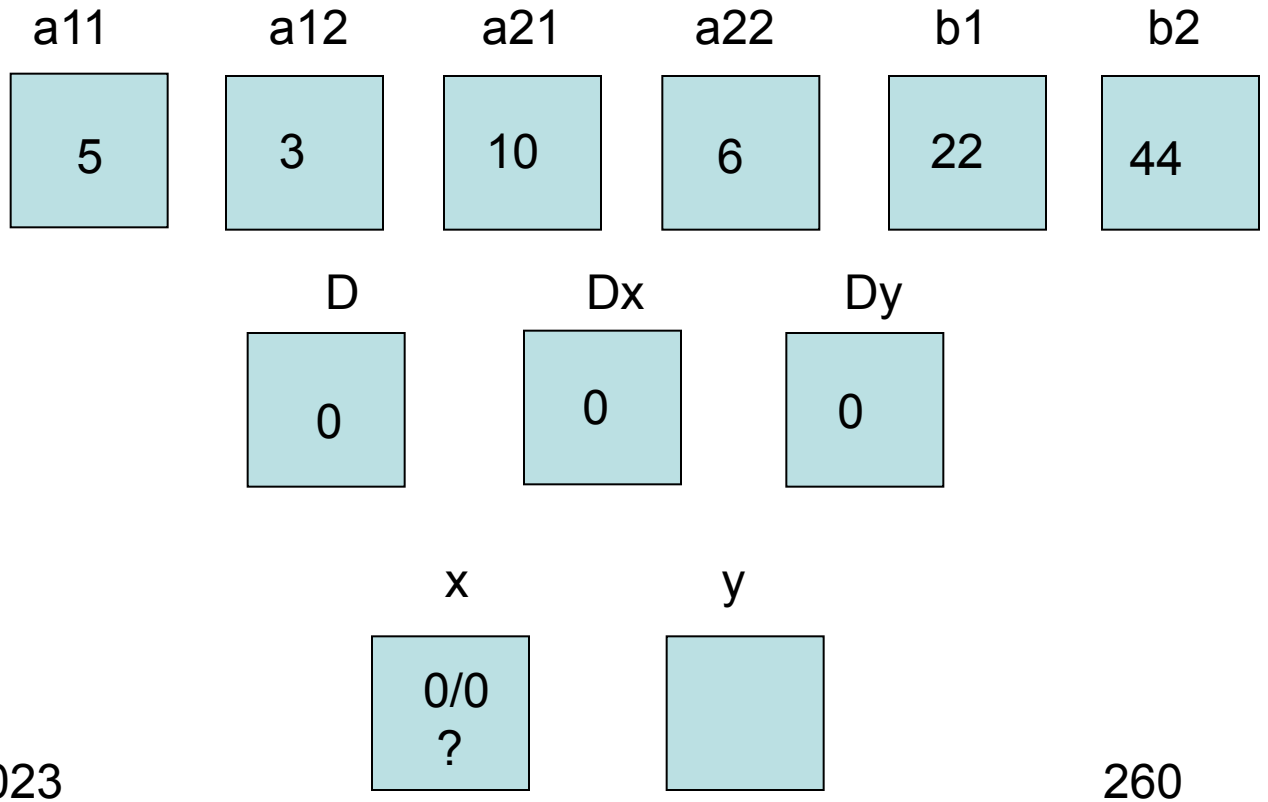
Найденные в результате трассировки значения величин x и y совпали с известным решением системы. Можно ли говорить о безусловной правильности алгоритма?

Проведём для очистки совести ещё одну проверку. Возьмём, например, такую систему:

$$\begin{cases} 5x + 3y = 22 \\ 10x + 6y = 44, \end{cases}$$

которая имеет точно такие же корни x=2 и y=4.

```
read(a11, a12, a21, a21, a22, b1, b2);
  D := a11*a22 – a12*a21; //Знаменатель
  Dx := b1*a22 – b2*a12; //Числитель для x
  Dy := b2*a11 – b1*a21; //Числитель для y
  x := Dx/D;
  y := Dy/D;   //Корни системы
write (x, y);
```



Этот тест завершился выявлением в алгоритме *принципиально невыполнимого действия.*

Несмотря на то, что результаты одной из тестовых проверок совпали с требуемыми, приходится делать вывод об *ошибочности* алгоритма, так как по условию задачи требуется, чтобы он давал ответ для *любых коэффициентов системы*.

Основной целью тестирования является *нахождение ошибок* в алгоритме. Если выбраны исходные данные, с помощью которых удалось обнаружить ошибку, то тест признаётся **полезным**, так как после исправления ошибки алгоритм станет более надёжным.

Теперь необходимо выяснить причину появления ошибки. Выполнение алгоритма оказалось прерванным из-за невозможности выполнить деление на нуль. Значит необходимо проверить правильность оператора, с помощью которого вычислется значение этого знаменателя.

Проверка правильности записи выражения для оказавшегося равным нулю знаменателя *D*, показывает, что ошибка отсутствует.

Получается, что обращение в нуль знаменателя является проявлением свойств самой системы уравнений, которые проявились на выбранном для теста наборе её коэффициентов.

Таким образом, проведённый анализ вернул нас к проверке метода решения задачи — правилу Крамера. Необходимо внимательнейшим образом проверить правильность понимания самого метода, а особенно ***условия*** его применения.

Проверка его точной математической формулировки показывает, что ошибка была допущена и заложена в самом начале построения алгоритма.

В формулировке правила Крамера указывается, что выбранные для вычисления корней системы соотношения можно применять *только в том случае*, когда главный определитель системы, он же знаменатель обеих дробей, *D не равен нулю*.

Но такая проверка в алгоритме не организована, именно поэтому во втором тестовом наборе исходных данных возникло деление на нуль.

Заметим, что в проведённых рассуждениях анализ проводился в обратном направлении — от точки появления ошибки к началу. Существует, ещё один вариант анализа когда проверка проводится в прямом направлении, с самого начала рассуждений. Заметим, что такой вариант может привести к обнаружению ошибки быстрее, чем обратный анализ.
Итак, тестирование показало, что при правильном построении алгоритма перед вычислением корней необходимо проверить, потенциально опасную операцию деления: отличен ли знаменатель от нуля.

Это означает, что в алгоритме появляются, по крайней мере, две ветви — одна должна описывать действия при не равном нулю знаменателе, а вторая — при его обращении в нуль. Следовательно, обсуждаемый алгоритм имеет ветвление.

Таким образом, дополнительный анализ привел к отказу от представления о линейности обсуждаемого алгоритма. Это довольно типичная ситуация.

Вопрос. А сколько именно ветвей необходимо зафиксировать в обсуждаемой алгоритме решения задачи?

И вновь лучше всего начать с математического аспекта решения рассматриваемой задачи.

Если вспомнить точные математические утверждения относительно существования решений системы линейных алгебраических уравнений, то получим следующие возможные ситуации:

- главный определитель системы отличен от нуля — решение существует и единственно;
- главный определитель равен нулю, а дополнительный — нет, тогда решений у системы нет;
- и главный и дополнительный определители равны нулю — система имеет бесконечно много различных решений.

Полный анализ позволяет утверждать, что в строящемся алгоритме должно быть предусмотрено три ветви

Этот пример наглядно демонстрирует необходимость внимательного отношения к математическим формулировкам, используемым при построении алгоритма.

Кроме того, всегда следует помнить об обязательности проведения тестирования построенного алгоритма, которое при правильном его проведении обеспечивает нахождение большинства ошибок.

Алгоритмы с ветвлением

Алгоритмы с ветвлениями более сложны по сравнению с линейными. В любом таком алгоритме имеется *несколько* вариантов действий и обязательно существует этап, во время выполнения которого требуется *выбрать* один из вариантов.

В качестве простого примера алгоритма, содержащего ветвление, рассмотрим порядок осуществления разговора по стационарному телефону.

Сначала производится подготовка к звонку, во время которой следует снять трубку телефона и набрать номер.

После чего в трубке можно услышать либо обычные гудки свободной линии, либо частые гудки занятости линии, и по их типу определяется возможность проведения разговора.
Таким образом, выявлено два варианта возможных действий, две ветви алгоритма:

1) при наличии обычных гудков необходимо выждать ответа абонента, затем поговорив с ним положить телефонную трубку на место;
2) услышав частые гудки (обнаружена занятость линии), следует положить трубку сразу.

Обсудим подробнее структуру полученного алгоритма. Какой этап является этапом выбора?

Этап определения характера гудков.

Теперь обратим внимание на то, что этапу выбора предшествует *линейный* участок подготовки к ветвлению, содержащий действия: а) снять трубку телефона и б) набрать номер. Такого рода начальные, подготовительные действия принято называть *инициализацией*.

Начальные действия, выполняемые с целью подготовки к каким-либо другим, более сложным действиям называются **инициализацией** (англ. initialization – инициирование): инициализация ветвления, инициализация цикла и т.д.

В общем случае *инициализация* представляет собой подготовку к работе, создание условий для работы или определение каких-либо параметров работы. Обычно это приведение объекта (устройства, алгоритма или программы, некоторого участка алгоритма или программы) в состояние готовности к использованию или выполнению действий.

Далее следует этап, содержащий *условие выбора* варианта дальнейших действий: оценка характера услышанного гудка. На этом этапе собственно происходит разделение на две ветви.

Затем размещаются точные описания каждого из вариантов действий (дождаться ответа и поговорить; не разговаривать).

После выполнения действий по любому из вариантов ветви вновь *объединяются* в одну. В обсуждаемом примере заключительный участок алгоритма содержит одно и то же действие для любой ветви: необходимо положить трубку на телефонный аппарат.

Выявленная структура характерна для любого алгоритма с ветвлением:

1. общий участок инициализации, то есть подготовки к ветвлению;
2. этап выбора, содержащий одно или несколько условий;
3. отдельное описание каждой ветви;
4. общий участок, завершающий ветвление.

Рассмотренный пример содержит два варианта действий лишь на первый взгляд. Более детальный анализ показывает, что на самом деле вариантов больше, а именно: если слышны обычные гудки то абонент может ответить, а может и не ответить. Чтобы определиться в возникшей ситуации, требуется некоторое время выждать.

Получается, что в результате первоначального анализа были выделены только две ветви, затем *одна из них разделилась ещё на две ветви*. Таким образом, в алгоритме должно быть *три* ветви.

А ещё более внимательный анализ показывает, что при обсуждении характера гудков, была упущена важная для построения корректного алгоритма возможность — гудки могут вообще отсутствовать, что означает неисправность телефонного аппарата или же отсутствие связи с телефонной станцией. Тогда получается, что в алгоритме должно быть предусмотрено ещё большее количество ветвей.

Проведённые рассуждения показывают, что анализ задачи необходимо проводить *полностью*, детально исследуя каждую возникающую ситуацию и обеспечивая охват *всей проблемной области*.

Отдельные ветви алгоритма чаще всего оказываются линейными. Однако, как видно из рассмотренного примера, любая ветвь может в свою очередь содержать ветвление. Такая ситуация, когда какая-либо ветвь алгоритма содержит ветвление, называется **вложением** ветвлений. На любом уровне вложения любая ветвь может быть ещё раз разделена, это создает возможность реализации алгоритмов, содержащих необходимое количество ветвей.

Общие рекомендации по построению алгоритмов с ветвлениями

Участок инициализации — подготовки к ветвлению — служит для формирования *возможности* проверить условие выбора одного из вариантов, а также *возможности* выполнения *любой* выбранной ветви, поэтому он ***обязательно*** должен присутствовать в алгоритме с ветвлением. Участок инициализации обычно бывает линейным.

В каждом конкретном случае исполнения алгоритма с ветвлением действия выполняются только по какому-то одному варианту, а действия всех остальных вариантов не затрагиваются. Указать заранее, до начала исполнения алгоритма, до появления конкретных условий, какой именно вариант придётся выбрать *невозможно*.

Поэтому в алгоритме с ветвлением должны быть предусмотрены и точно заданы абсолютно *все возможные* варианты, а также условия их *однозначного* выбора.

Неверное задание условий выбора ветвей, их пересечение, допускающее *произвольный* выбор разных ветвей, или неполный охват этими условиями всех возможных в решении задачи вариантов, приводит к неоднозначности алгоритма и к неправильному решению задачи.

Завершающий участок ветвления служит для объединения всех путей исполнения алгоритма. Он может быть не очень большим и содержать всего одно действие, но по современным представлениям о структуре алгоритма, такой участок должен быть предусмотрен после любого ветвления.

Чтобы обеспечить выполнение этих требований целесообразно придерживаться следующих рекомендаций:

1. исходя из формулировки задачи и анализа проблемной области, определить *точное количество* различных вариантов возможных действий, неполный охват проблемной области приведёт к некорректности алгоритма;
2. для *каждого* из вариантов выявить точные условия его выбора;
3. для обеспечения однозначного выбора *только одного* из возможных вариантов, количество условий должно быть равно количеству ветвей, кроме того они не должны попарно пересекаться;
4. для каждого варианта точно и полно описать все реализующие его действия;
5. перед этапом выбора необходимо сформировать участок инициализации, на котором подготавливается возможность осуществления выбора, а также возможность выполнения всех действий во всех ветвях.

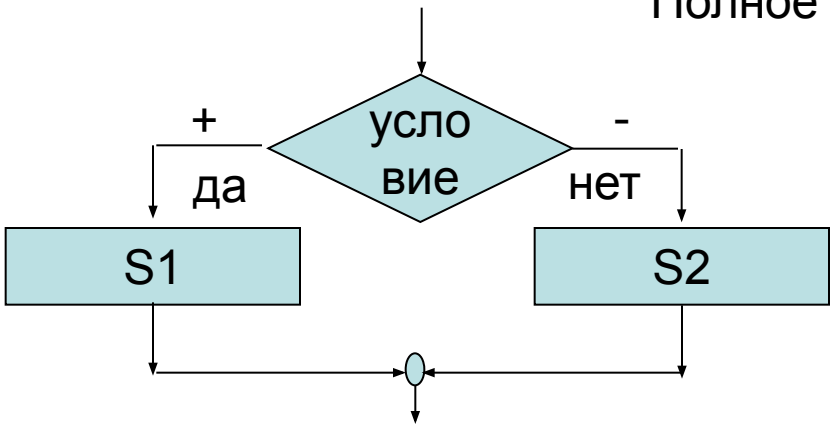
Несмотря на то, что участок инициализации ветвления должен находиться *перед* этапом выбора, его лучше формировать *после* того, как определятся все ветви и все условия их выбора, после того как станет полностью понятно, что именно потребуется для осуществления выбора и для исполнения каждой из ветвей.

Для реализации алгоритмов, содержащих ветвления, используется несколько разновидностей управляющих конструкций, которые соответствуют различным количествам ветвей в алгоритмах и разным условиям их выбора. Подробное их рассмотрение связано с особенностями конкретных алгоритмических языков. Поэтому мы обсудим только простейшие, но при этом *универсальные* варианты управляющих конструкций, с помощью которых можно построить алгоритмы с любыми ветвлениями.

Управляющие конструкции ветвлений

Заметим, что с организацией ветвлений мы уже фактически сталкивались, рассматривая программы для машины Поста. В системе команд этой машины предусмотрена отдельная команда, которая в зависимости от содержания текущей ячейки на ленте делит алгоритм на две ветви: одну, начинающуюся командой с верхним номером, а вторую — командой с нижним номером. Эта команда, следовательно, относится к управляющим конструкциям ветвления, которая приспособлена к особенностям машин Поста.

Полное ветвление

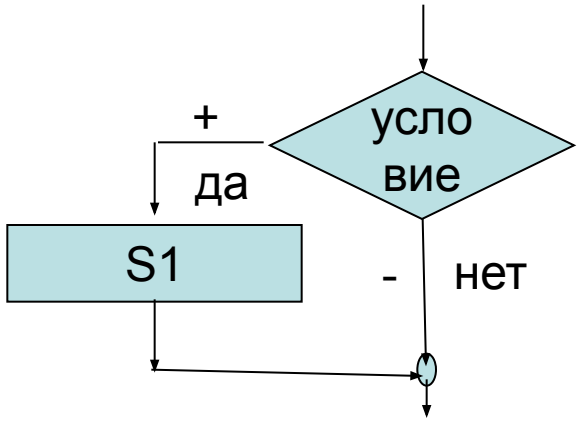


if <условие> then S1 else S2

if (<условие>) S1; else S2;

Полное ветвление используется когда в алгоритме присутствуют два непустых варианта действий

Сокращенное ветвление



if <условие> then S1

if (<условие>) S1;

Сокращенное ветвление используется, если в одном из вариантов никаких действий выполнять не нужно

Основной особенностью рассматриваемых управляющих конструкции ветвления является использование в ней *только одного условия*, хотя имеются две ветви и каждая из них выбирается с помощью своего условия.

Пусть *L1* — логическое выражение соответствующие условия выбора первой ветви, а *L2* — логическое выражение выбора второй ветви. В управляющих конструкциях считается, что одно из этих условий является *отрицанием* другого: *L1*= ¬ *L2*.

Поэтому для *корректного* построения ветвлений с помощью этих конструкций, сформулированные в конкретных алгоритмах условия выбора ветвей *L1* и *L2* не должны пересекаться $L1 \cap L2 = \varnothing$ и вместе они должны охватывать всю проблемную область задачи.

Как следствие в управляющих конструкциях ветвления можно использовать любое из условий *L1* или *L2*. Фактически включённое в управляющую конструкцию условие мы будем называть **условием разделения** ветвей.

Для машин Поста условие разделения связано с наличием или отсутствием отметки в текущей клетке ленты. В блок-схемах такое условие может быть совершенно произвольным и его можно формулировать на естественном языке. А в условных операторах условие разделения всегда должно быть представлено *выражением логического типа*.

По определению условного оператора каждая из ветвей может быть представлена *только одним оператором*. Если по смыслу выполняемых действий одного оператора недостаточно и требуется несколько операторов, то их можно «превратить» в один, заключив в операторные скобки.

Пример организации полного ветвления

Для любого заданного значения вещественного аргумента *x* вычислить значение кусочно-непрерывной функции *y*, заданной следующими соотношениями:

$$y = \begin{cases} 2x+1, x > 3 \\ \sin x, x \le 3 \end{cases}$$

Чтобы вычислить значение функции необходимо знать конкретное числовое значение её аргумента *x*, который, следовательно, является входной величиной. Выходная величина прямо указана в условии задачи — это искомое значение функции *y*. В данном случае целесообразно выбрать имена входной и выходной величин совпадающими с их математическими обозначениями: *x* и *y* соответственно. По условию задачи они относятся к вещественному типу.

По определению функции возможны всего два варианта вычисления её значения:

1. Если фактическое значение аргумента *x* окажется *больше* чем 3 (*x*>3), то для вычисления следует выбрать верхнее определение *y*=2*x*+1.
2. Если заданное значение аргумента *x* окажется *не больше* чем 3 (*x*≤3), необходимо выбрать нижнее определение *y*=sin *x*.

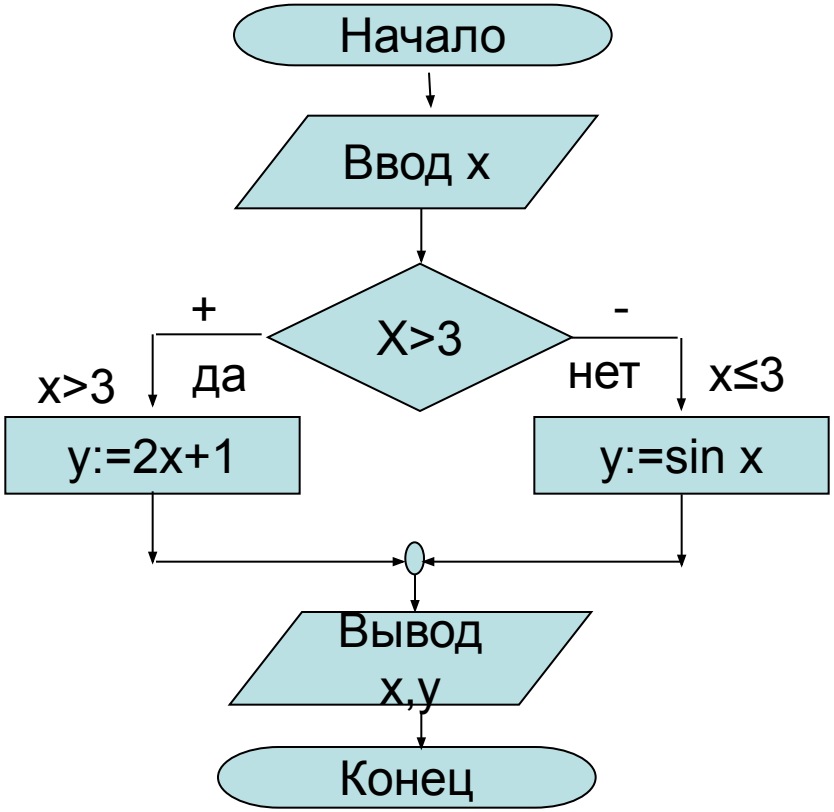
Условия выбора ветвей $x>3$ и $x\leq 3$ взаимно не пересекаются, каждое из них является отрицанием другого, и вместе они охватывают всю вещественную прямую, поэтому с их помощью можно построить корректное ветвление.

Для вычисления значения функции в каждом из вариантов, а также для выбора ветви достаточно знать только значение аргумента, поэтому инициализацией ветвления является ввод значения x.

Действия в каждой из ветвей задаются соответствующими операторами присваивания: $y:=2x+1$ для $x>3$ и $y:=\sin x$ для $x\leq 3$.

При построении ветвления в качестве условия разделения можно выбрать любое из двух имеющихся условий, требуется только внимательно следить за соответствием выбранного условия разделения и ветви.

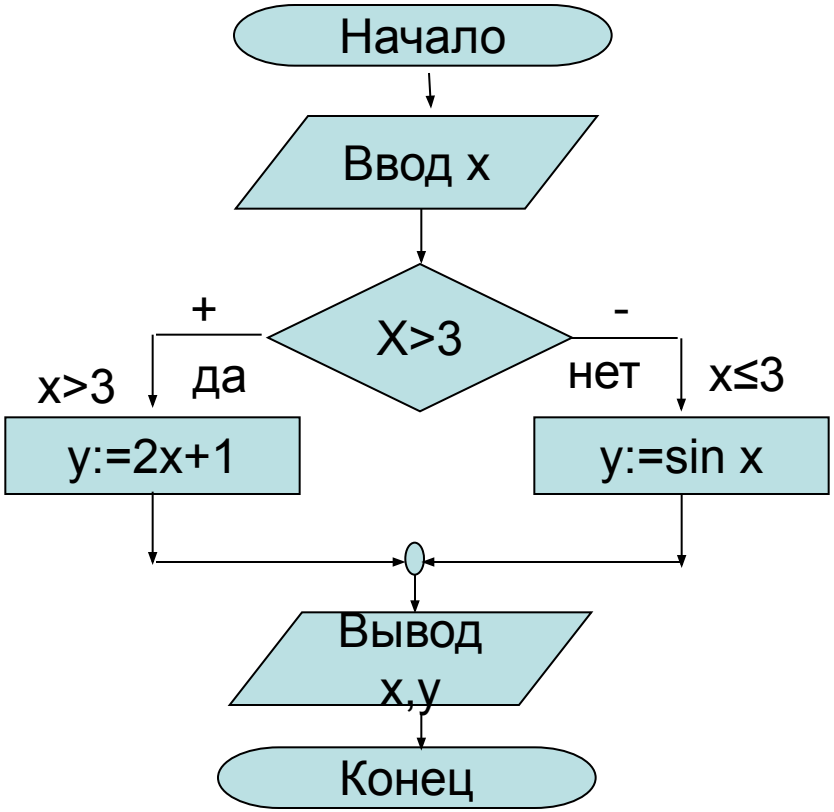
Заметим, что для наглядности ***полезно*** выводить не только непосредственный результат, но и значение аргумента, для которого он получен. Поэтому завершается алгоритм организацией вывода значений аргумента x и функции y.

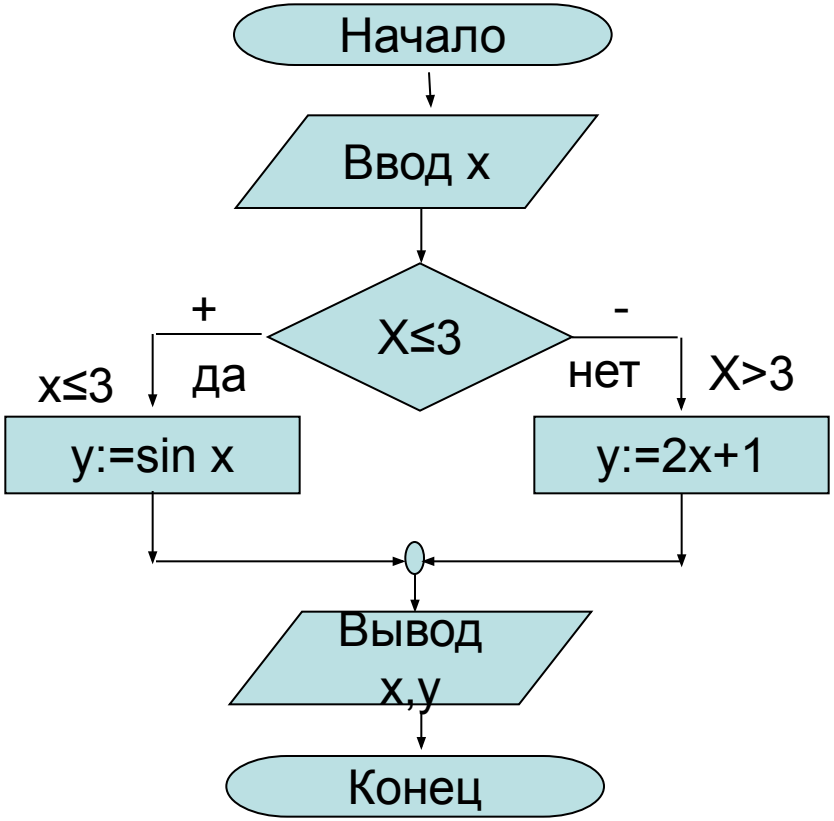



```
var x, y: real;
begin
    read(x);
    If x>3 then
        {x>3, берется  верхняя ветвь}
         y:= 2*x -1
       else
          {x≤3, берется нижняя ветвь}
          y:= sin(x);
    write (x,y)

end.
```

```
// Си-подобный.
main ( )
{
    float  x, y;
    scan(x);
   if (x>3.0)       //x>3, выбрана  верхняя ветвь
        y=2.0*x+1.0;
     else           //x<=3,  выбрана нижняя ветвь
        y=sin(x);
    print (x, y);
}
```

Если выбрать противоположное условие x≤3, то ветви следует поменять местами.





```
// Паскале-подобный.
var x, y: real;
begin
   read(x);
   if x<=3.0 then    //x<=3, выбрана нижняя ветвь
                y:=sin(x)
           else      //x>3, выбрана  верхняя ветвь
               y:=2.0*x+1.0;
   write (x, y);
end.
```

```
// Си-подобный
main ( )
{
    float  x, y;
    scan(x);
     if (x<=3.0)
        y=sin(x);    //x<=3,  выбрана нижняя ветвь
     else              //x>3, выбрана верхняя ветвь
         y=2.0*x+1.0;
    print (x, y);
}
```

Ещё раз обращаю внимание на графические приёмы оформления фрагментов программ, которые облегчают их чтение и понимание. В частности на способ записи условных операторов, когда начальная часть оператора, содержащая условие выбора ветви и ключевое слово else располагаются на отдельных строчках, а каждая ветвь сдвигается влево по отношению к ним. При этом ветви записываются с одинаковым отступом от начала строки

Отметим ещё одну особенность приведённых выше алгоритмов. Оператор присваивания записан в виде y:=2.0*x+1.0, хотя проще было бы записать его так: y:=2*x+1. Правильными являются оба варианта записи операторов.

Но в выражении 2*x+1 участвуют константы целого типа и вещественная переменная. Такая запись выражений *разрешена*, но во время выполнения вычислений целые константы придётся преобразовать к вещественному типу, а на это компьютеру потребуется дополнительное время. Поэтому лучше использовать вариант y:=2.0*x+1.0, в котором все типы совпадают.

Для получения более эффективной машинной программы рекомендуется выбирать форму записи непосредственных данных, которая соответствует типу выражения и тем самым минимизирует затраты на преобразования типов.

Особо обращаю внимание на отсутствие точки с запятой после первой ветви в записи на языке Паскаль и в её наличие в этом месте в записи на языке Си. Это отличие объясняется разным смыслом использования этого знака в этих языках.

Это смысловое различие как раз и проявляется в обсуждаемых фрагментах. Если на языке Паскаль поставить точку с запятой после завершения первой ветви:

```
if x>3.0 then y:=2.0*x+1.0;  else y:=sin(x)
```

то она *отделит* начальный участок условного оператора

```
if x>3.0 then y:=2.0*x+1.0
```

от его хвостовой части

```
else  y:=sin(x)
```

которая не может самостоятельно использоваться. Такая запись *полной* формы условного оператора в языке Паскаль является грубой ошибкой.

Если на языке Си записать условный оператор без заканчивающей оператор точки с запятой:

```
if (x<=3.0)  y=sin(x)  else  y=2.0*x+1.0;
```

то это означает, что в оператор, задающий первую ветвь, войдут все символы текста до ближайшей точки с запятой, которой он должен заканчиваться

```
y=sin(x) else y=2.0*x+1.0;
```

Это запись также является грубой ошибкой.

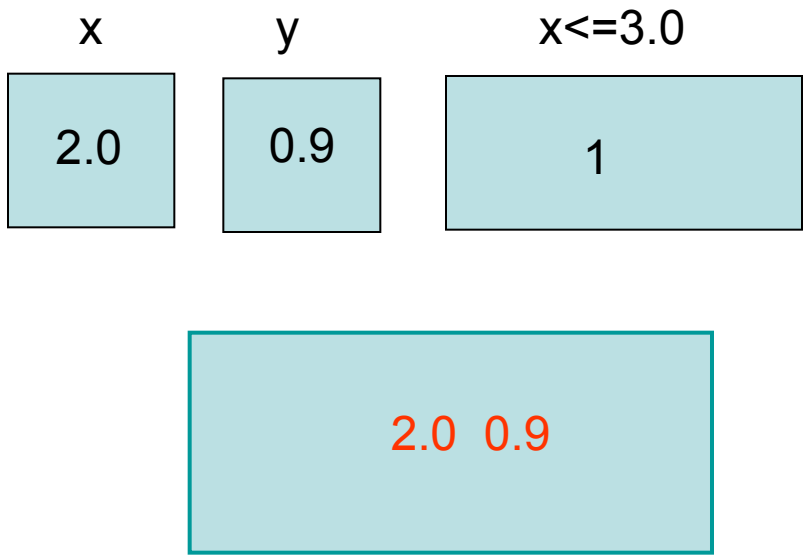
Для алгоритмов с ветвлением тестовые наборы *должны проверить каждую из ветвей*. Кроме того обязательно должны проверяться граничные точки, отделяющие ветви друг от друга. Дело в том, что такие точки часто бывают *точками разрыва* функций, и от того в какую ветвь попадает граничная точка может зависеть правильность построенного ветвления.

В рассматриваемой задаче можно выбрать любое значение, меньшее чем 3, равное 3 и большее, чем 3.

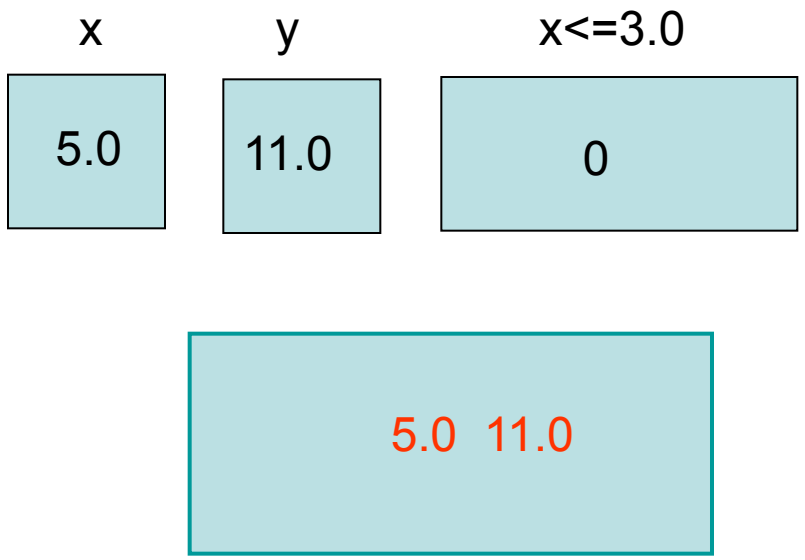
Для конкретности возьмём в качестве тестовых значения x=2.0, 3.0 и 5.0. Для проведения теста необходимо заранее, до исполнения алгоритма знать каким должен быть ответ. В соответствии с определением функции, должно получиться:

- y(2.0)=sin(2.0)≈0.9
- y(3.0)=sin(3.0) ≈0.1
- y(5.0)=11.0

```
main( )
  {
    float x, y;
    scan (x);
    if (x<=3.0)
        y=sin (x); //выбрана нижняя ветвь
      else  //выбрана верхняя ветвь
        y=2.0*x+1.0;
    print (x, y);
  }
```



```
main( )
  {
    float x, y;
    scan (x);
    if (x<=3.0)
        y=sin (x); //выбрана нижняя ветвь
      else  //выбрана верхняя ветвь
        y=2.0*x+1.0;
    print (x, y);
  }
```



На основании этих трассировок можно сделать вывод: проведённое тестирование *не выявило ошибок* в проверяемом алгоритме.

Пример организации неполного ветвления

Для ознакомления с особенностями использования сокращенной формы ветвления рассмотрим задачу *упорядочивания* последовательности, состоящей из двух элементов.

Задана последовательность из двух элементов a_1, a_2 . Требуется переставить её элементы так, чтобы они образовали неубывающую последовательность.

Элементы заданной последовательности a_1, a_2 являются исходными величинами алгоритма. Как и ранее, можно считать их простыми переменными или же элементами массива.

Поскольку эта задача является частным случаем общей задачи упорядочивания последовательности, состоящей из *n* из элементов, целесообразно рассматривать последовательность как вектор **a**={a_1,a_2}, состоящий из двух (n=2) элементов.

В задаче требуется переставить элементы последовательности. Это значит, что требуется *изменить* входной массив, следовательно, элементы этого же, *но измененного* массива являются выходными данными.

В качестве имени массива целесообразно использовать его математическое обозначение a.

К какому типу могут относиться элементы рассматриваемого массива?

Над элементами массива в процессе преобразования потребуется выполнять только операции сравнения и присваивания.

Эти операции, как правило, не определены для полных значений структурированных типов (например, для массива, взятого целиком), но допустимы для любых простых типов.

Поэтому элементы массива могут относиться к целому, вещественному, логическому или символьному типу.

Для определённости будем считать элементы массива целыми. Как выглядят описания этого массива на Паскале и на Си?

var a: array[1..2] of integer; // Па, int a[2]; // Си.

Чтобы понять в чём именно состоит задача и наметить путь её решения настоятельно рекомендуется выбрать ***несколько конкретных и простых числовых примеров*** и вручную получить искомый результат.

Пусть ***a={4,8}***. Компоненты вектора расположены в нужном порядке и делать ничего не нужно.

Пусть теперь ***a={5,2}***. В данном случае компоненты вектора следует переставить. В результате получится вектор ***a={2,5}*** с требуемым порядком следования компонент.

Далее можно рассуждать опираясь на математические соображения: между введёнными элементами массива a_1 и a_2 могут быть только два варианта отношений: либо $a_1 \le a_2$, либо $a_1 > a_2$.

Вспоминая математическое определение неубывающей последовательности, получаем, что в *результирующей* последовательности элементы *должны* удовлетворять условию $a_1 \le a_2$.

Таким образом, получаем два варианта, две ветви алгоритма:

1. при удовлетворении для введённых значений неравенства $a_1 \le a_2$ никаких действий выполнять не требуется — последовательность уже упорядочена;
2. при удовлетворении неравенства $a_1 > a_2$ элементы последовательности придётся выполнить ***дополнительное*** действие: поменять местами элементы последовательности (что равносильно обмену значениями между компонентами вектора).

Условия выбора ветвей $a_1 \le a_2$ и $a_1 > a_2$ взаимно не пересекаются, каждое из них является отрицанием другого, и вместе они охватывают все возможные варианты соотношений между элементами массива a_1 и a_2, поэтому с их помощью можно построить корректное ветвление.

Итак, в решении задачи один из вариантов пустой, то есть не содержит никаких действий. Эта ситуация соответствует ветвлению в сокращенной форме.

Для построения ветвления с использованием этой формы в качестве условия разделения следует взять неравенство $a_1 > a_2$, при удовлетворении которого выбирается *непустая* ветвь

Непустая ветвь представляет собой линейный алгоритм обмена значениями, который был ранее построен.

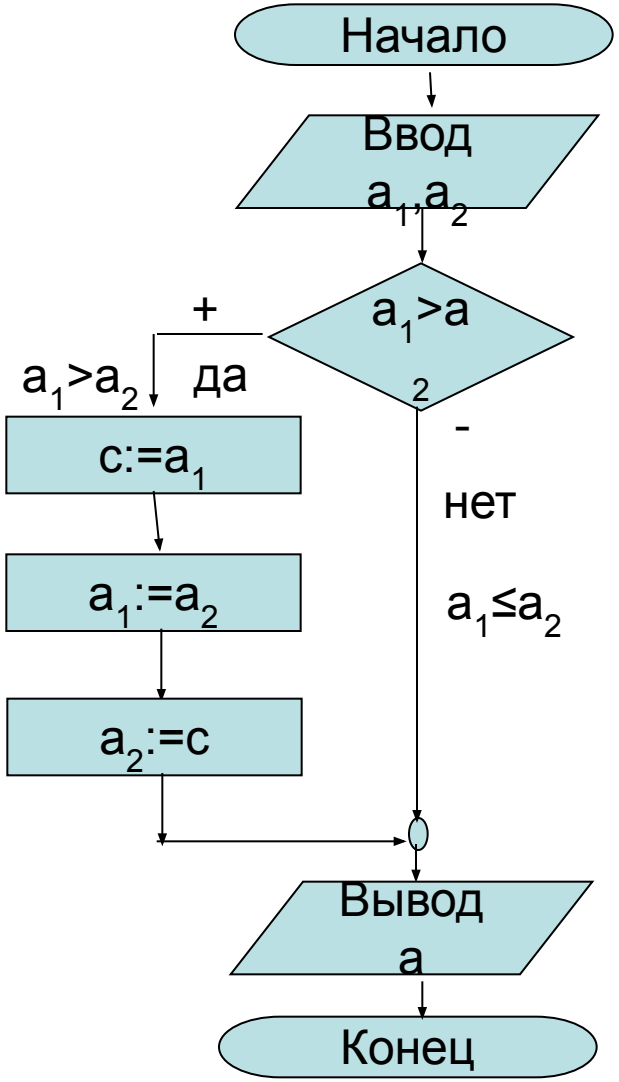
Только применить его требуется не к переменным *n* и *m*, а к элементам массива a_1 и a_2. Записать этот фрагмент алгоритма, используя для обмена вспомогательную переменную c.

c:=a[1]; a[1]:=a[2]; a[2]:=c

Стандартным способом построения алгоритмов, является использование *ранее построенных алгоритмов* в качестве «кирпичиков», составных частей нового алгоритма. Поэтому программисту необходимо знать как можно больше способов решения задач и соответствующих им алгоритмов. Чтобы стать профессионалом необходимо постоянно наращивать свой личный «багаж» знаний в области алгоритмов обработки данных.

При записи на алгоритмическом языке не следует забывать о необходимости описания всех появившихся по ходу рассуждений промежуточных величин.

Следует также обратить внимание на то, что действия в непустой ветви задаются *более чем одним оператором*, и в соответствии с правилом их необходимо заключить в операторные скобки, то есть применить *составной оператор*.




```
var a: array of real; c: real;
begin
  read(a);
  if a[1]>a[2] then
      begin {обмен значениями}
        c:=a[1];
        a[1]:=a[2];
        a[2]:=c
      end;
  write(a)
end.
```

Записать алгоритм на СИ-подобном языке

```
// Си-подобный.
main ( )
{
    int a[2];
    scan(a); // Ввод всех элементов массива a
     if (a[0]>a[1])
         {// Обмен значениями элементов массива
          Int c=a[0]; a[0]:=a[1]; a[1]:=c;
          }
    print (a); // Вывод всех элементов массива a
}
```

При обсуждении вариантов построения полного ветвления было выяснено, что изменив условие на противоположное и поменяв местами ветви, можно получить полностью эквивалентный алгоритм.

Однако для сокращенной формы такие изменения делать *не следует*, так как получаются плохо воспринимаемые при чтении конструкции. Для сокращенной формы ветвления настоятельно рекомендуется выбирать условие разделения, которое соответствует *непустой* ветви.

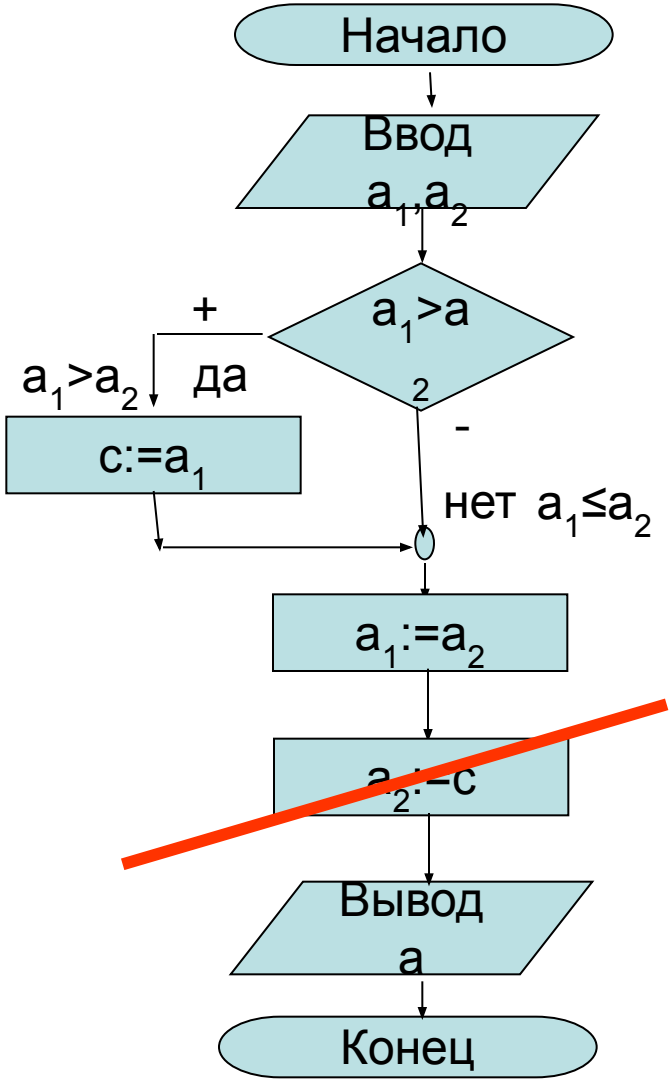
Как и в случае полной формы для тестирования построенного алгоритма необходимы, по крайней мере, три тестовых набора — по одной на каждую ветвь и один на точку перехода между ветвями. Можно взять, например, такие наборы: (5, 2), (4, 8) и (3, 3).

Довольно часто встречаются ошибки, связанные с пропуском операторных скобок при построении ветвлений. По уже упоминавшемуся правилу любая ветвь условного оператора может состоять только из одного оператора.

Вместе с тем, по логике решения задачи может потребоваться (как это имеет место в рассматриваемой задаче) чтобы ветвь содержала более одного оператора.

В таких случаях применяется составной оператор, поскольку любая группа операторов, заключенная в операторные скобки, по правилам алгоритмических языков *считается одним оператором*. Именно поэтому при выполнении условия a[1]<a[2] выполнялись все три оператора, входящие в составной.

Теперь рассмотрим, к чему приведёт ошибка, если по невнимательности, или по незнанию в алгоритме не использовать составной оператор.



1. Невозможность выполнить присваивание для некоторых вариантов исходных данных.
2. Несоответствие смысла действий условиям задачи и разработанному способу её решения.

Алгоритмы с большим количеством ветвей

В задачах часто возникают ситуации, при которых в алгоритме приходится строить ветвление с тремя, четырьмя или ещё большим количеством ветвей. Для построения таких алгоритмов в алгоритмических языках предусмотрены специальные операторы, но они имеют некоторые ограничения на применение.

Построение алгоритмов с большим количеством ветвей базируется на том, что по правилам языка в состав условного оператора могут входить любые операторы, в том числе и опять условные. Такая подобная матрёшке конструкция называется **вложенным ветвлением**. Оператор, внутри которого производится вложение, называется **внешним**, а вкладываемый оператор — **внутренним**.

При вложении условного оператора в условный происходит *расщепление* одной из ветвей на две. В новую ветвь можно на общих основаниях вложить ещё одно ветвление и т.д. Глубина вложений обычно не ограничивается, что и позволяет строить алгоритмы с любым количеством ветвей.

Алгоритм, содержащий *n* ветвей, строится за *n*-1 шаг. На каждом шаге производится отделение одной ветви, для чего применяется одна управляющая конструкция ветвления.

Конструкция первого шага образует внешнее ветвление, а каждая следующая вкладывается в ветвление предыдущего шага.

На первом шаге выбирается любая ветвь, желательно с простым условием выбора, а все оставшиеся ветви объединяются в ***группу***. Эта группа считается второй ветвью *обычного* ветвления, которое реализуется условным оператором в полной форме. В результате такого разделения в «групповой» ветви окажется *n*-1 исходная ветвь.

Далее точно таким же способом «групповая» ветвь вновь разделяется на две. После чего в новой «групповой» ветви останется *n*-2 исходные ветви.

Процесс продолжается до тех пор, пока в последней «групповой» ветви не останется только две исходные ветви. Последнее разделение приведет к тому, что все исходные ветви окажутся отделёнными друг от друга.

Пример: алгоритм с тремя ветвями

Построить алгоритм вычисления значения функции

$$y = \begin{cases} 0, x \le 0 \\ x^2 - x, 0 < x \le 1 \\ x^2 + \sin \pi x, x > 1 \end{cases}$$

для любого вещественного значения аргумента x.

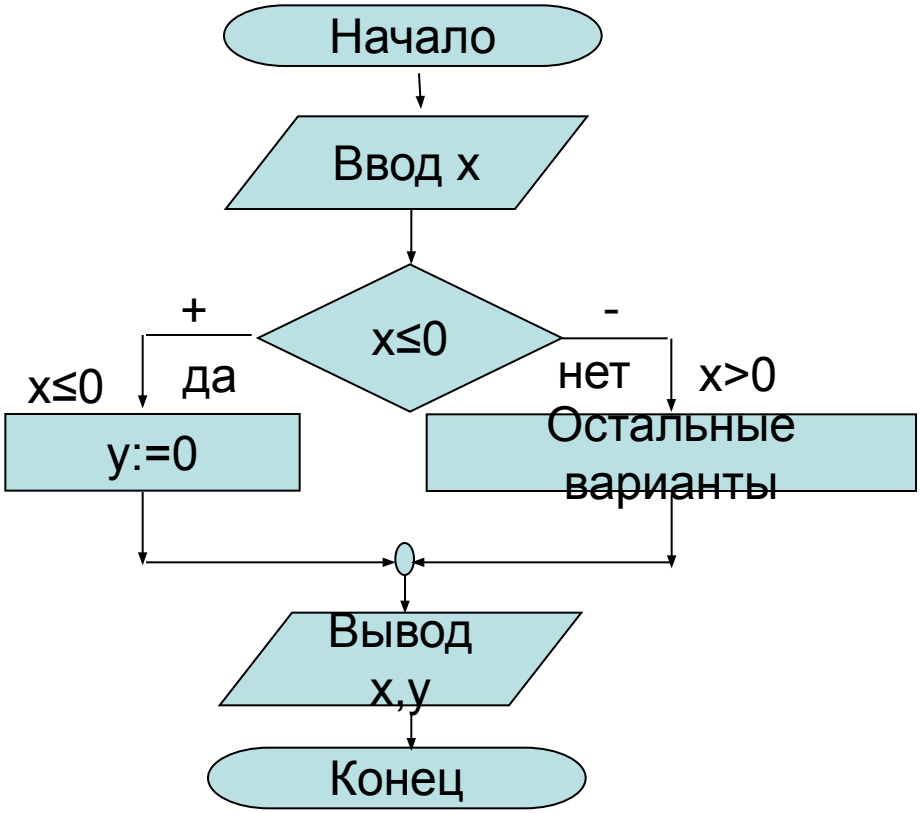
Рассуждая точно также как ранее. приходим к выводу о том, что входной величиной алгоритма является вещественная переменная *x*, а выходной — вещественная переменная y. И описания переменных и действия, связанные с вводом/выводом, записываются точно такие же, как в решении задачи о полном ветвлении.

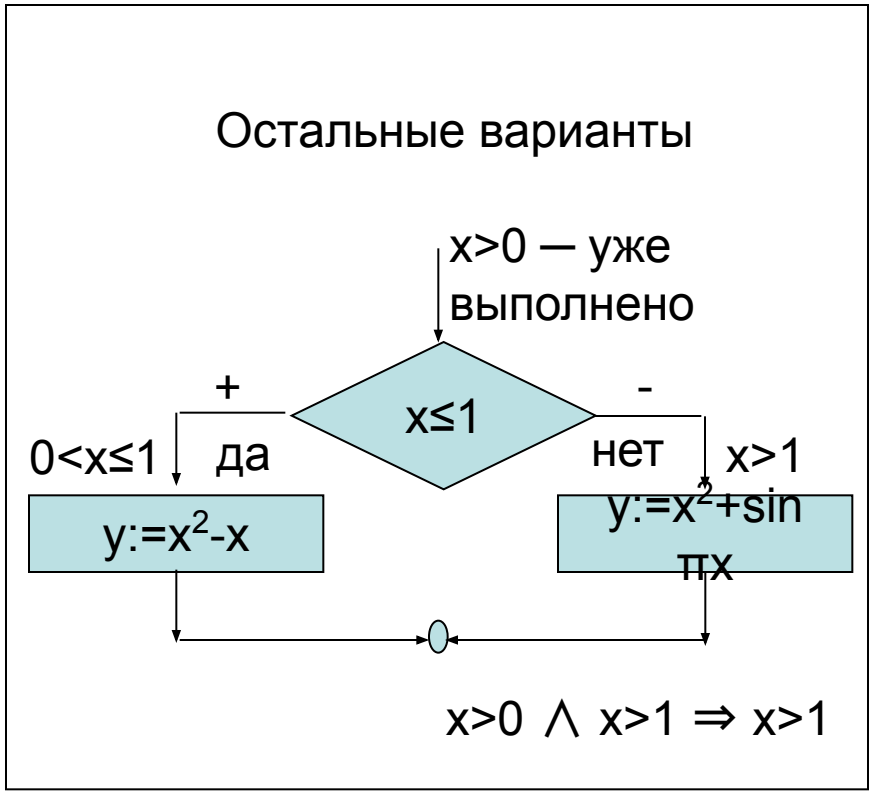
По определению функции, видно, что в зависимости от заданного значения аргумента x возможны три варианта вычисления её значения:

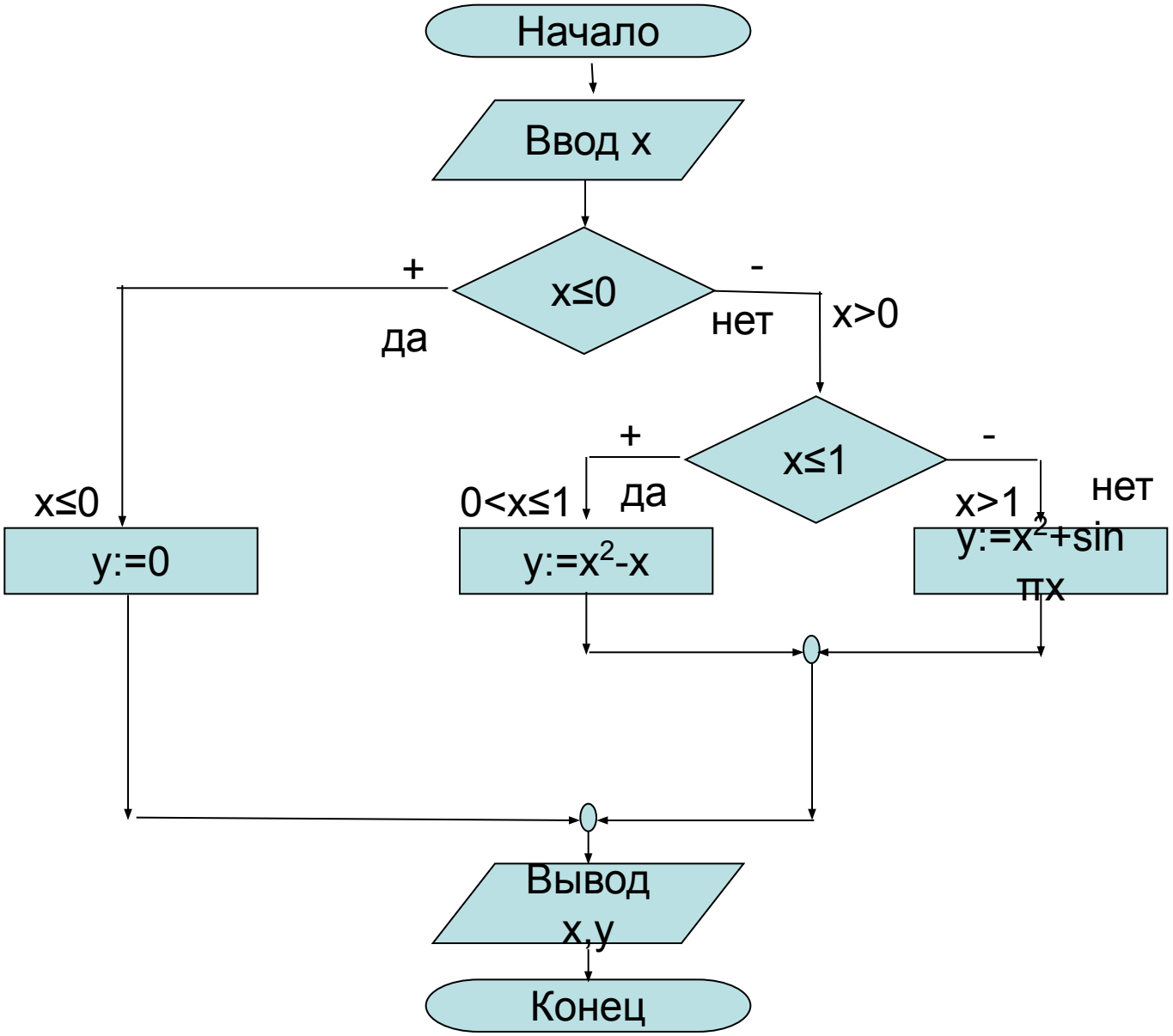
1. при $x \le 0$: $y(x)=0$;
2. при $0 < x \le 1$: $y(x)=x^2-x$;
3. при $x > 1$: $y(x)=x^2+\sin \pi x$.

Анализ условий выбора ветвей показывает, что они попарно не пересекаются, а их объединение охватывает всю вещественную прямую. Следовательно, эти три условия могут быть использованы для *корректной* организации ветвления.

По общей схеме организации ветвления вначале с помощью условия $x \le 0$ выделим две ветви — одну обычную, содержащую первый вариант вычисления y:=0, а вторую — объединяющую все оставшиеся варианты.


Начало
Ввод x
x≤0
+
-
x≤0
да
нет
x>0
y:=0
Остальные варианты
Вывод x,y
Конец


Остальные варианты
x>0 — уже выполнено
x≤1
+
-
0<x≤1
да
нет
x>1
y:=x2-x
y:=x2+sin πx
x>0 ∧ x>1 ⇒ x>1


Начало
Ввод x
x≤0
+
да
-
нет
x>0
x≤1
+
да
-
нет
x≤0
y:=0
0<x≤1
y:=x2-x
x>1
y:=x2+sin πx
Вывод x,y
Конец

Обратите внимание на оформление в этой блок-схеме вложенного и внешнего ветвлений.

Блоки действий в ветвях рекомендуется располагать на одном и том же уровне.

Каждое деление на две ветви завершается объединение с помощью собственного соединителя, и каждый соединитель располагается строго под своим блоком ветвления.

Такое оформление помогает лучше понять содержание ветвлений, и кроме того, соответствует *технологии структурного программирования*, о которой речь пойдёт в дальнейшем.

```
// Паскале-подобный
    var x, y: real;
 begin
    read(x);
    if x<= 0.0 then //x<=0.0, выбрана верхняя ветвь
                    y:=0.0
                else  //x>0,  выбрана «групповая» ветвь
                    {Остальные варианты}
   write (x, y);
end.
```

Теперь, учитывая, что в точке алгоритма, обозначенной комментарием «Остальные варианты», неравенство x>0 удовлетворяется, разделим две оставшиеся ветви. Выбрав неравенство x<=1 в качестве условия *вложенного* ветвления, получим такое уточнение:

```
                {Остальные варианты. x  уже больше нуля}
if x<=1.0 then {0<x<=1 — средняя ветвь}
                  y:=x*x–x
               else {x>1 — нижняя ветвь}
                  y:=x*x+ sin(π*x);
```

После вложения отдельно проработанного участка в заготовку получим окончательный вариант алгоритма:

```
// Паскале-подобный
var x, y: real;
begin
  read(x);
  if x<= 0.0 then //x<=0.0, выбрана верхняя ветвь
               y:=0.0
          else  //x>0,  выбрана «групповая» ветвь
             {Остальные варианты}
              if x<=1.0 then
                          {0<x<=1 — средняя ветвь}
                           y:=x*x – x
                       else {x>1 — нижняя ветвь}
                           y:=x*x +  sin(π*x);
    write (x, y);
end.
```

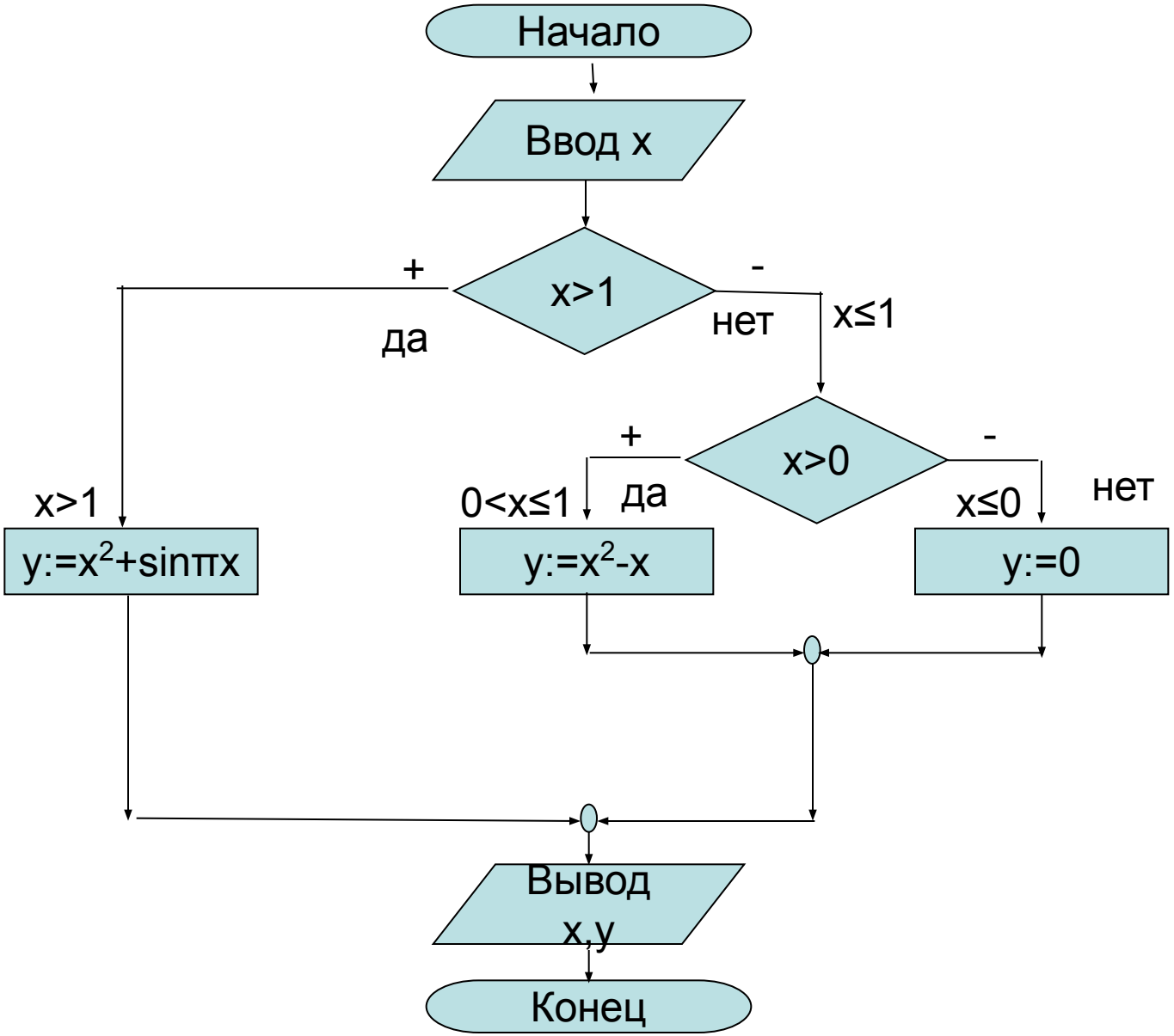
```
// Си-подобный
main ( )
 {
   float x, y;
   scan (x);
   if (x<=0.0)
          y=0.0; //x<=0,   выбрана верхняя ветвь
     else
       // x>0,  выбрана  «групповая» ветвь, остальные варианты
        if (x<=1.0)
             y=x*x – x; /* 0<x<=1 —  средняя ветвь */
          else
            y=x*x + sin(π*x); /* x>1 — нижняя ветвь */
   print (x, y);
 }
```

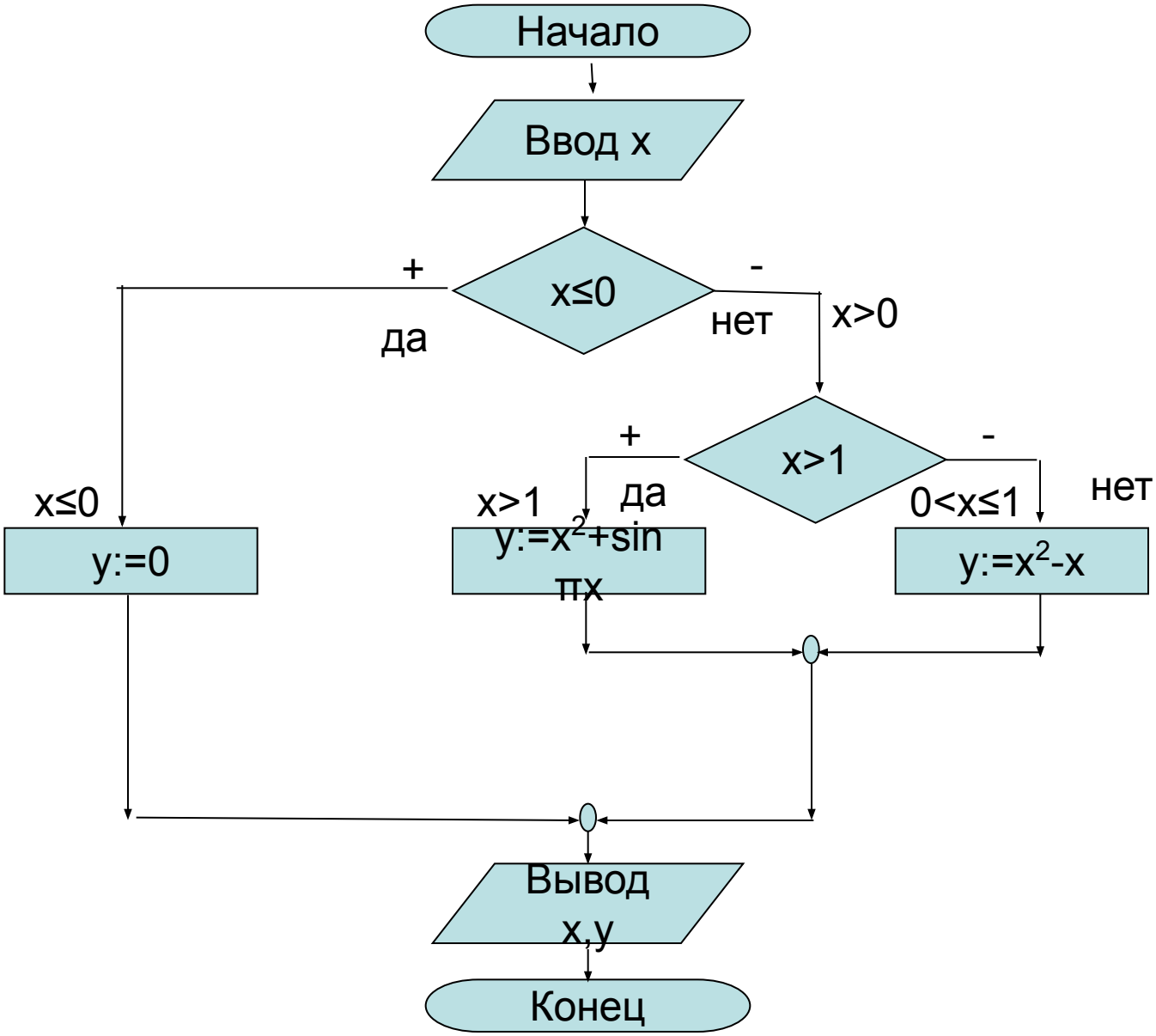
Для тестирования алгоритма, содержащего n ветвей необходимо подобрать $2n$-1 тестовый набор, по одному набору на каждую ветвь и на каждую точку разделения ветвей.

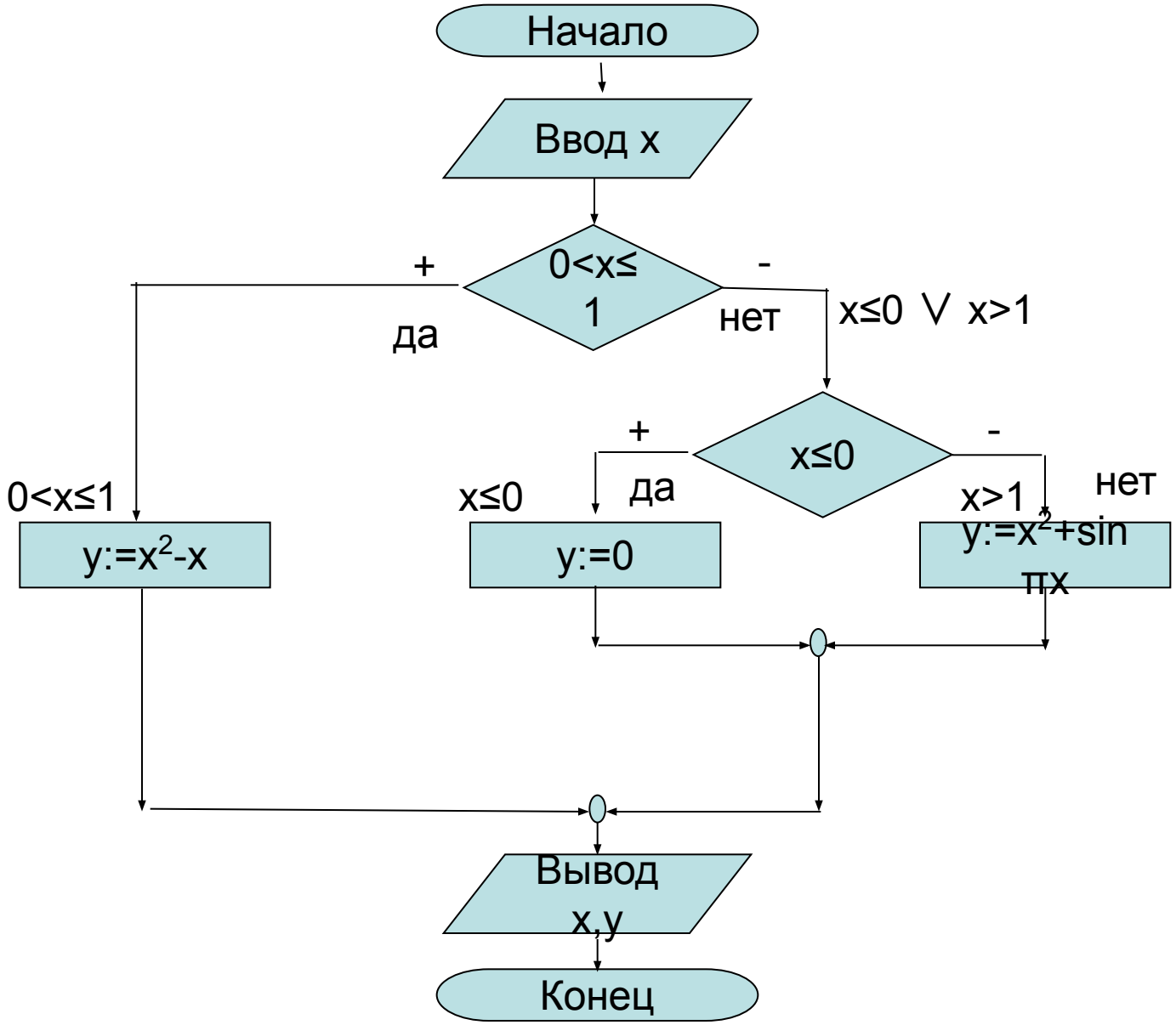
Особое внимание необходимо уделять точкам ветвления, так как именно в этих точках чаще всего по невнимательности допускаются ошибки, связанные с тем, что вместо строго неравенства используется нестрогое (либо наоборот), в результате граничная точка попадает не в ту ветвь, в которой она должна быть.

Исходя из приведённых соображений для тестирования алгоритма необходимо, по крайней мере, пять тестовых наборов.

В построенном алгоритме для внешнего ветвления было выбрано условие разделения $x \leq 0$. Вообще говоря, для разделения ветвей можно выбрать любое другое условие. Исходя из того, что в рассматриваемом алгоритме три ветви и каждое вложенное ветвление можно записать двумя способами, получаем, что алгоритм можно записать шестью различными способами. Несложно доказать, что для алгоритма, содержащего n ветвей при использовании условного оператора существует $n!$ вариантов его записи.


Начало
Ввод x
x>1
+
да
-
нет
x≤1
x>0
+
0<x≤1
да
-
нет
x≤0
x>1
y:=x²+sinπx
y:=x²-x
y:=0
Вывод
x,y
Конец


Начало
Ввод x
x≤0
+
да
-
нет
x>0
x>1
+
да
-
нет
0<x≤1
x≤0
y:=0
x>1
y:=x2+sin πx
y:=x2-x
Вывод x,y
Конец


Начало
Ввод x
0<x≤1
+
да
-
нет
x≤0 ∨ x>1
0<x≤1
y:=x2-x
x≤0
+
да
x≤0
-
x>1
нет
y:=0
y:=x2+sin πx
Вывод x,y
Конец

Все построенные таким путём варианты со смысловой точки зрения одинаковы и при отсутствии ошибок все они должны давать правильные результаты. Однако в качестве логических выражений для условных операторов *рекомендуется* выбирать как можно более простые условия.

Например, условие выбора средней ветви использовать не следует, так как по правилам алгоритмических языков ему соответствует выражение вида $(x>0.0) \wedge (x\leq 1.0)$, содержащее три операции, следовательно, соответствующий фрагмент алгоритма будет менее эффективным, чем при использовании простых выражений вида $x>1.0$.

Построение циклических алгоритмов

Решения подавляющего большинства задач включают в себя многократно выполняющиеся последовательности действий. Поэтому очень сложно найти практически важную программу или алгоритм, в котором отсутствует циклический участок.

Общую структуру циклических алгоритмов удобно проследить на примере, рассмотренном при обсуждении свойства конечности алгоритма. В этом алгоритме циклический участок образуется из трех действий: 1) приём лекарства утром, 2) приём лекарства в обед и 3) приём лекарства вечером.

Такую группу *многократно* выполняемых действий принято называть **телом цикла**. *Однократное* выполнение тела цикла называется **итерацией** (англ. iteration — повторение). В рассматриваемом примере итерация представляет собой совокупность действий по приёму дневной нормы лекарства.

С телом цикла связано **условие повторения**, которое определяет *необходимость* выполнения действий, входящих в тело. В примере условие повторения имеет вид «пока больной не выздоровеет».

Фрагмент конструкции цикла, в котором находится условие повторения, обычно называют **заголовком цикла**. Таким образом, в *стандартном* варианте цикл состоит *из заголовка и тела*.

Условие повторения может иметь самые разные формы. При его удовлетворении итерации *продолжаются*, а как только это условие нарушится, итерации *прекращаются*.

Поэтому с циклом можно связать ещё одно условие — **условие завершения**, которое является *отрицанием* условия повторения. Если удовлетворено условие завершения, то условие повторения не удовлетворяется, поэтому итерации прекращаются, а значит, завершается выполнение цикла в целом. В обсуждаемом алгоритме условие завершения — «больной выздоровел».

При *некорректном* построении цикла условие повторения *может* удовлетворяться *всегда*, а условие завершения — не удовлетворяться никогда. Такая ситуация носит название **бесконечного** цикла. Наличие непрекращающихся итераций, очевидно, противоречит требованию *конечности алгоритма*.

Перед циклом обязательно находится подготовительный (обычно линейный) участок, без которого невозможно организовать цикл. В обсуждаемом примере подготовительные действия состоят в предварительной покупке лекарства. Такого рода действия принято называть **инициализацией цикла**.

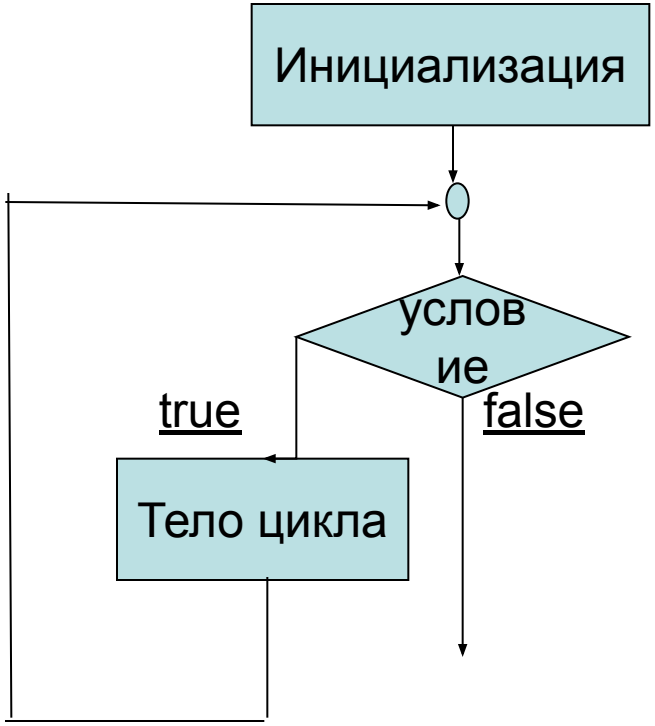
Итак, в структуре алгоритма с циклом имеется три основных, связанных с его организацией элемента:

- инициализация цикла;
- условие повторения/условие завершения;
- тело цикла.

Любой цикл обязательно содержит эти составные части, и от их точного построения и ***взаимодействия*** существенно зависит правильность результата исполнения цикла, да и алгоритма в целом.

Фактически мы уже сталкивались с циклами, когда обсуждали программу для машины Поста. Если проанализировать, например, ход выполнения программы увеличения заданного на ленте числа на единицу, легко заметить, что команды 1 и 2 выполняются *два раза подряд*, следовательно, они образуют *тело цикла*, в котором исполняются *две итерации*.

Для задания циклов в алгоритмах используются несколько управляющих конструкций. В частности, для формирования любых циклических участков алгоритмов используется универсальный ***цикл с предусловием***.



Действия, которые требуются выполнить многократно, образуют ***тело цикла***. Часть цикла, содержащая условие называется ***заголовком цикла***. Действия, которые выполняются до цикла и связаны с ним, называются ***инициализацией (подготовкой) цикла***.

В соответствии с этой схемой вначале выполняются все связанные с инициализацией цикла действия, которые, собственно говоря, не входят в цикл, а лишь подготавливают его выполнение. За исключением очень редко встречающихся случаев без такой подготовки выполнить цикл *невозможно*.

Сама управляющая конструкция цикла с предусловием предписывает следующую последовательность шагов его выполнения:

1. проверяется условие повторения, и если оно удовлетворяется, то тело цикла выполняется, то есть осуществляется *ровно одна* итерация тела;
2. затем происходит возврат к проверке условия повторения и вновь при его удовлетворении выполняется одна итерация тела цикла;
3. эти действия — проверка условия повторения и очередная итерация — *повторяются* до тех пор, пока условие повторения удовлетворяется;
4. как только условие повторения *в первый раз* окажется нарушенным, (то есть как только удовлетворится условие завершения) выполнение цикла заканчивается.

Особо подчеркнём, что если условие повторения окажется нарушенным при самой первой проверке, то тело цикла не выполнится ни разу.

Эта управляющая конструкция имеет практически точные аналоги почти во всех алгоритмических языках.

Так, в языке Паскаль цикл с предусловием определяется так:

while <условие > do <тело цикла>

В Си подобных языках:

while (<условие >) <тело цикла>;

Начальный фрагмент оператора while *<логическое выражение>* do // Па, while (*<логическое выражение>*) // Си представляет собой *заголовок цикла*, а входящее в заголовок логическое выражение — это записанное по правилам соответствующего языка *условие повторения* тела цикла.

Завершающий фрагмент конструкции *<оператор>* — это *тело цикла*, то есть многократно повторяющееся действие. Из определений следует, что тело цикла в этой форме оператора может быть задано *только одним оператором*.

Если по логике решения задачи в этом месте должно находиться более одного оператора, то они должны быть заключены *в операторные скобки*.

Значение входящего в заголовок логического выражения вычисляется *перед* каждым выполнением тела цикла.

Если получается логическое значение true, то оператор, задающий тело цикла, выполняется.

Как только логическое выражение получит значение false — выполнение оператора цикла завершается и наступает очередь выполнения оператора, расположенного следом за оператором цикла

Общие рекомендации по построению циклических алгоритмов

Чтобы обеспечить корректность построения циклов можно придерживаться следующих рекомендаций:

1. исходя из формулировки задачи и анализа проблемной области, зафиксировать действия, которые должны выполняться многократно, то есть определить *тело цикла*;
2. точно сформулировать *условие повторения* тела цикла;
3. для контроля правильности выбранного условия повторения рекомендуется явно определить, что именно в терминах предметной области означает невыполнение этого условия, то есть явно сформулировать *условие завершения*, которое должно быть *отрицанием* условия повторения;
4. и тело цикла, и условие его повторения, как правило, зависят от некоторых начальных установок, которые должны быть заданы на участке инициализации;
5. действия, которые должны быть выполнены во время инициализации цикла, следует прорабатывать в *последнюю* очередь, когда полностью определится всё, что должно быть подготовлено как для проверки условия повторения, так и для итераций тела цикла;
6. полученные в результате этого анализа элементы циклической структуры фиксируются с помощью блок-схемы или выбранного алгоритмического языка.

Эти рекомендации имеют весьма общий характер, и их следует тщательно соблюдать, особенно на первых порах, когда приобретаются первичные навыки алгоритмизации. Несоблюдение этих рекомендаций, как правило, приводит к нарушению требований конечности, определённости или выполнимости алгоритма.

Одна из самых характерных ошибок в построении циклов состоит в *некорректной* формулировке условия повторения.

Главное в проверке корректности условий повторения и завершения: убедится в том, что итерации (выполнения) тела цикла рано или поздно прекратятся. То есть необходимо следить за тем, чтобы на какой-либо итерации было удовлетворено условие завершения, что и обеспечивает *завершаемость* цикла.

Завершаемость *за конченое время* всего алгоритма в целом гарантируется завершаемостью *каждого* присутствующего в алгоритме цикла.

Построенный циклический алгоритм необходимо особенно тщательно тестировать, поскольку чаще всего ошибки возникают именно в циклах.

Проверка правильности работы цикла требует анализа трех периодов в его выполнении, которые можно условно назвать **входом в цикл**, **рабочим периодом** и **выходом из цикла.**

- Вход в цикл представляет собой выполнение всех действий участка инициализации и первый анализ условия повторения. Цель тестирования этого периода — убедиться в том, что все участвующие в цикле переменные получили исходные значения. Кроме того, чтобы выполнение цикла началось, чтобы вход в цикл был фактически выполнен, условие повторения при первой проверке обязательно должно *удовлетворяться*.
- Рабочий период включает в себя проверку условия повторения и выполнения тела цикла. Цель тестирования этого периода — убедиться в наличии *действий* или *изменений*, которые после нескольких итераций приводят к удовлетворению *условия завершения*.
- Выход из цикла включает в себя *последнюю* итерацию, после выполнения которой условие повторения нарушается, и работа цикла завершается. Цель тестирования выхода — убедится в том, что тело цикла выполнено ровно столько раз, сколько требуется в алгоритме.

Для проверки корректности цикла необходимо составить тестовые наборы, с помощью которых контролируется каждый из периодов его выполнения. Кроме того, необходимо проверить реакцию алгоритма или программы на случай задания *бессмысленных* с точки зрения решаемой задачи значений переменных, связанных с циклом.

Задача о рекуррентной последовательности

Последовательности у которых явно заданы ***k*** подряд расположенных элемента, а любые другие элементы связаны друг с другом одним и тем же соотношением $a_n=f(a_{n-1}, a_{n-2}, \ldots, a_{n-k})$ называются ***рекуррентными***.

Пример.

Пусть элементы последовательности связаны друг с другом соотношением:

$$a_k = ka_{k-1} + \frac{1}{k+1}, k = 1,2,\ldots; a_0 = 0.5$$

Требуется найти а) ***5***, б) ***100***, в) ***n***-й элемент последовательности.

Найти пятый элемент последовательности можно следующим способом:

Подготовительные действия: положим k=0 и a_0=0.5

Положим k=1, тогда, зная a_0 можно найти a_1: $a_1=1*a_0+1/2=1$

Положим k=2, тогда, зная a_1 можно найти a_2: $a_2=2*a_1+1/3\approx2,33.$

Положим k=3, тогда, зная a_2 можно найти a_3: $a_3=3*a_2+1/4 \approx 7,25$

Положим k=4, тогда, зная a_3 можно найти a_4: $a_4=4*a_3+1/5\approx29,2$

Положим k=5, тогда, зная a_4 можно найти a_5 $a_5=5*a_4+1/6 \approx 146,17$

Зафиксируем вытекающие из рекуррентного соотношения важные для дальнейшего моменты в проведённых рассуждениях: для определения *любого* элемента последовательности a_k (кроме нулевого) необходимо знать: 1) его номер k ; 2) значение предыдущего элемента последовательности a_{k-1}.

Обратим также внимание на то, что алгоритм определения *пятого* элемента последовательности состоит *ровно из шести* шагов.

На начальном, нулевом шаге фиксируются явно заданные значения, а на оставшихся *пяти* определяются действия, необходимые для вычисления каждого очередного элемента. Эта связь между номером элемента и количеством шагов в линейном алгоритме *сохраняется* для любого другого элемента последовательности.

Отсюда следует, что аналогичный построенному линейный вариант алгоритма практически может быть использован только для тех случаев, когда требуется определить значение элемента последовательности с относительно *маленьким* номером.

Конечно, построить подобный алгоритм для вычисления сотого элемента (пункта *б* задачи) также можно, но его запись, содержащая сто один шаг, окажется очень громоздкой и неудобочитаемой.

С теоретической точки зрения линейный алгоритм, несмотря на то, что он будет содержать очень много шагов, можно записать для любого *конечного* номера элемента *n*, но такой номер должен быть задан *заранее*, *до построения* алгоритма.

Если номер элемента заранее не задан, и его значение необходимо вводить по ходу исполнения алгоритма или же вычислять с помощью какого-либо другого алгоритма, то записать *линейный* алгоритм для решения этой задачи *принципиально невозможно*, поскольку количество шагов такого алгоритма *на этапе его построения* оказывается неизвестным.

Заметим также, что у линейного варианта построения для различных значений *n* получаются разные алгоритмы, содержащие различное количество шагов в записи алгоритма.

А это в свою очередь означает, что использовать один и тот же *линейный* алгоритм для определения *различных* элементов последовательности *невозможно* — для каждого элемента потребуется свой собственный алгоритм.

Из этих рассуждений следует, что для получения решения пункта ***в***) задачи требуется искать принципиально другой способ построения.

При анализе проведённых расчётов возникает ощущение, что при вычислении каждого очередного элемента последовательности приходится выполнять одни и те же действия:

$$a_k = ka_{k-1} + \frac{1}{k+1}, k = 1,2,...$$

Чтобы это увидеть следует ввести в рассуждения переменную, которая имеет смысл текущего элемента последовательности. Назовем такую переменную, например **a.**

Особенность переменной a в том, что её значение совпадает с значением элемента последовательности, который рассматривается на текущем шаге:

0) **a**=a_0

1) **a**=a_1

2)**a**=a_2

…

k)**a**=a_k

Тогда, опираясь на свойства действия присваивания, вычисление очередного элемента можно записать в виде:

a:=k*a+1/(k+1)

По условиям задачи в начальный момент исполнения действий *к=0*, *а=0.5.*

```
k := 0: a := 0.5; {Задание начальных значений}
k := 1; a:=k*a+1/(k+1);  {Вычисление первого элемента}
k := 2; a:=k*a+1/(k+1);  {Вычисление второго элемента}
k := 3; a:=k*a+1/(k+1);  {Вычисление третьего элемента}
k := 4; a:=k*a+1/(k+1);  {Вычисление четвертого элемента}
k := 5; a:=k*a+1/(k+1);  {Вычисление пятого элемента}
```

Анализируя полученный фрагмент алгоритма, можно отметить, что в нем имеется группа почти одинаковых действий. Обращая внимание на то, что в каждой следующей группе значение ***k*** увеличивается ровно на единицу, приходим к выводу о том, что эту группу можно привести к совершенно одинаковой форме если использовать присваивание вида

k:=k+1

```
k := 0: a := 0.5: {Инициализация, задание начальных значений}
k := k+1; a:=k*a+1/(k+1);  {Вычисление первого элемента}
k := k+1 ; a:=k*a+1/(k+1);  {Вычисление второго элемента}
k := k+1;  a:=k*a+1/(k+1);  {Вычисление третьего элемента}
k := k+1; a:=k*a+1/(k+1);  {Вычисление четвертого элемента}
k := k+1; a:=k*a+1/(k+1);  {Вычисление пятого элемента}
```

```
// Си-подобный
k = 0; a = 0.5; // Задание начальных значений
k ++; a = k*a+1.0/(k+1);  // Вычисление первого элемента
k ++; a = k*a+1.0/(k+1);  //Вычисление второго элемента
k ++; a = k*a+1.0/(k+1);  //Вычисление третьего элемента
k ++; a = k*a+1.0/(k+1);  //Вычисление четвертого элемента
k ++; a = k*a+1.0/(k+1);  //Вычисление пятого элемента
```

Увидели, что имеется группа действий, которые выполняются неоднократно, то есть имеем дело с циклическим алгоритмом. Далее рассуждаем в соответствии с рекомендациями по построению циклов:

1. Устанавливаем, из каких действий состоит тело цикла:

увеличение номера элемента k на единицу и вычисление значения очередного элемента последовательности (текущего значения a).

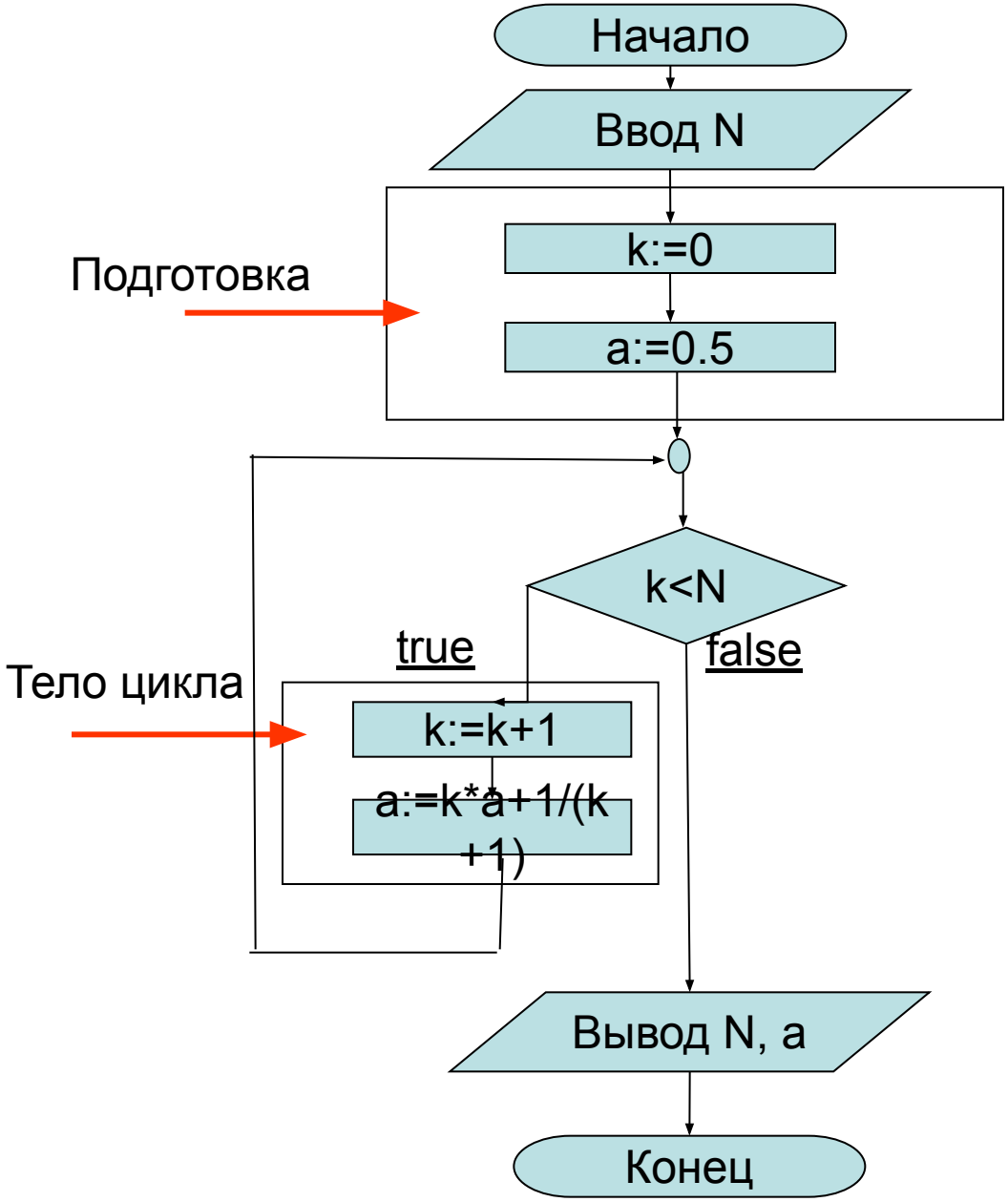
2. Определяем условие повторения:

повторяем вычисления, пока требуемый элемент не окажется уже вычисленным, то есть, его номер k меньше номера n искомого.

Определяем действия инициализации:

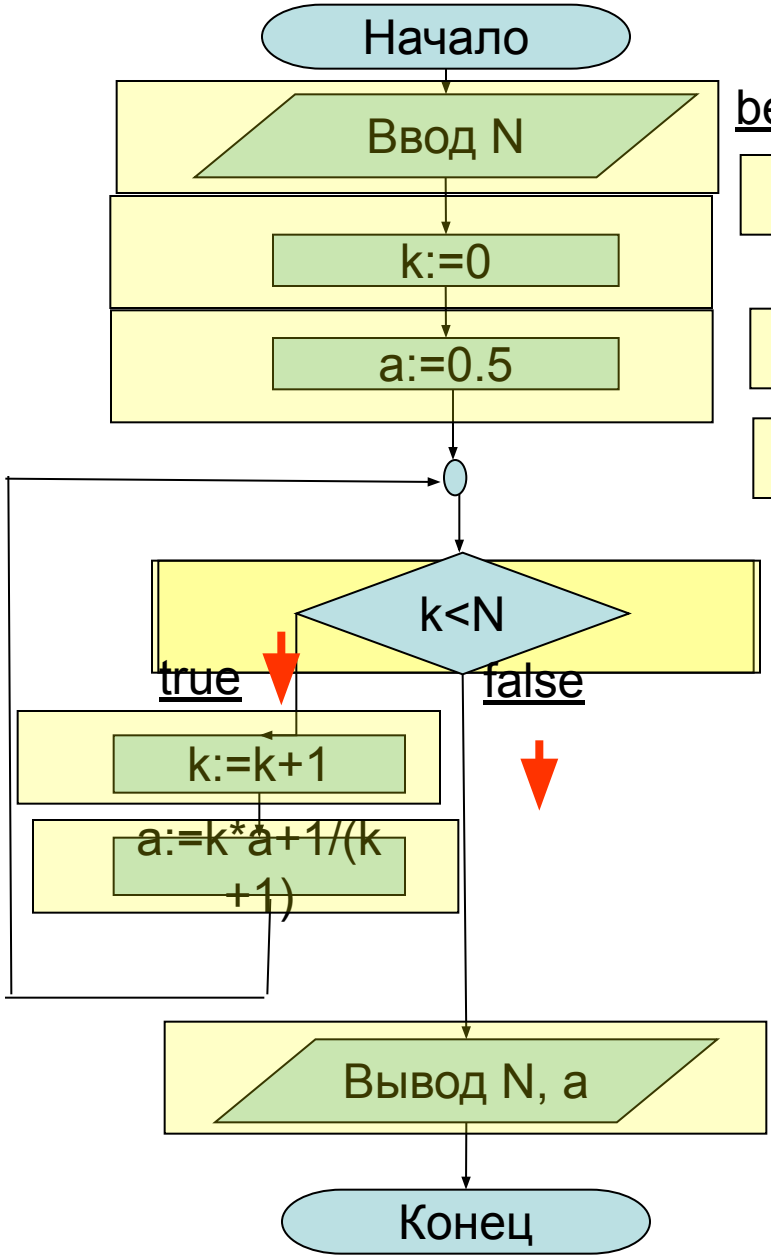
k=0 и a=0.5

Теперь по общей схеме построения алгоритмов определяемся с входными, выходными и промежуточными величинами и их типами.




```
     N,k: integer; a:real;
begin
     read(n);
     {Подготовка к циклу}
     k:=0;
     a:=0.5;

     while k<N do
          begin
               {Номер очередного
               элемента}
               k:=k+1;
               {вычисление  очередног
               элемента}
               a:=k*a+1/(k+1);
          end;
     write(N,a)
end.
```

```
// Си-подобный
main ( )
{
    int  n;
   scan(n); //Ввод заданного номера n
    // Инициализация цикла
   int  k=0; //Начальный номер
   float a=0.5; //Начальный элемент
    while (k<n)  //Пока номер уже вычисленного элемента меньше номера искомого
       {
         //Получение номера элемента
          k++;
           //Получение его значения
           a=k*a+1.0/(k+1);
       }
// Вывод номера и значения искомого элемента
    print (n, a[n]);
}
```




```
N,k: integer; a:real;
begin
    read(n);
    {Подготовка к циклу}
    k:=0;
    a:=0.5;
    while k≤N do
    begin
        {Номер очередного элемента}
        k:=k+1;
        a:=k*a+1/(k+1);{вычисление очередного элемента}
    end;
    write(N,a)
end.
```

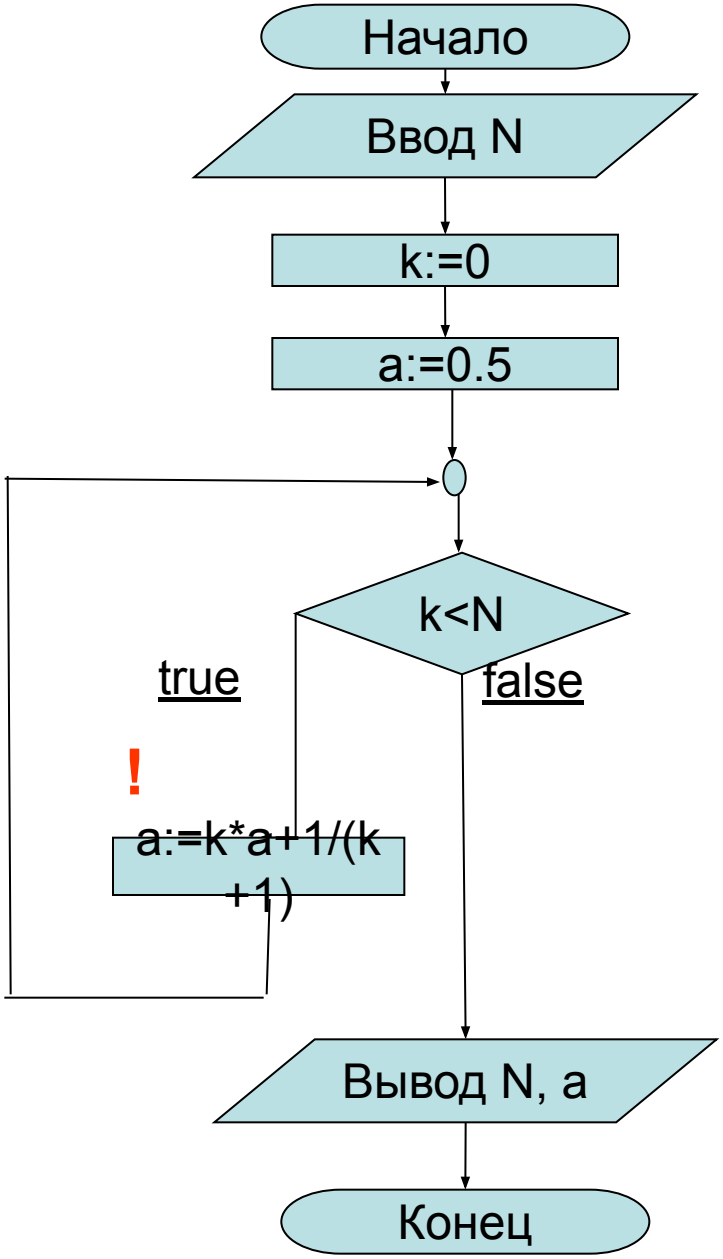
N	k	a
5	5	146, 17

Условие k<n

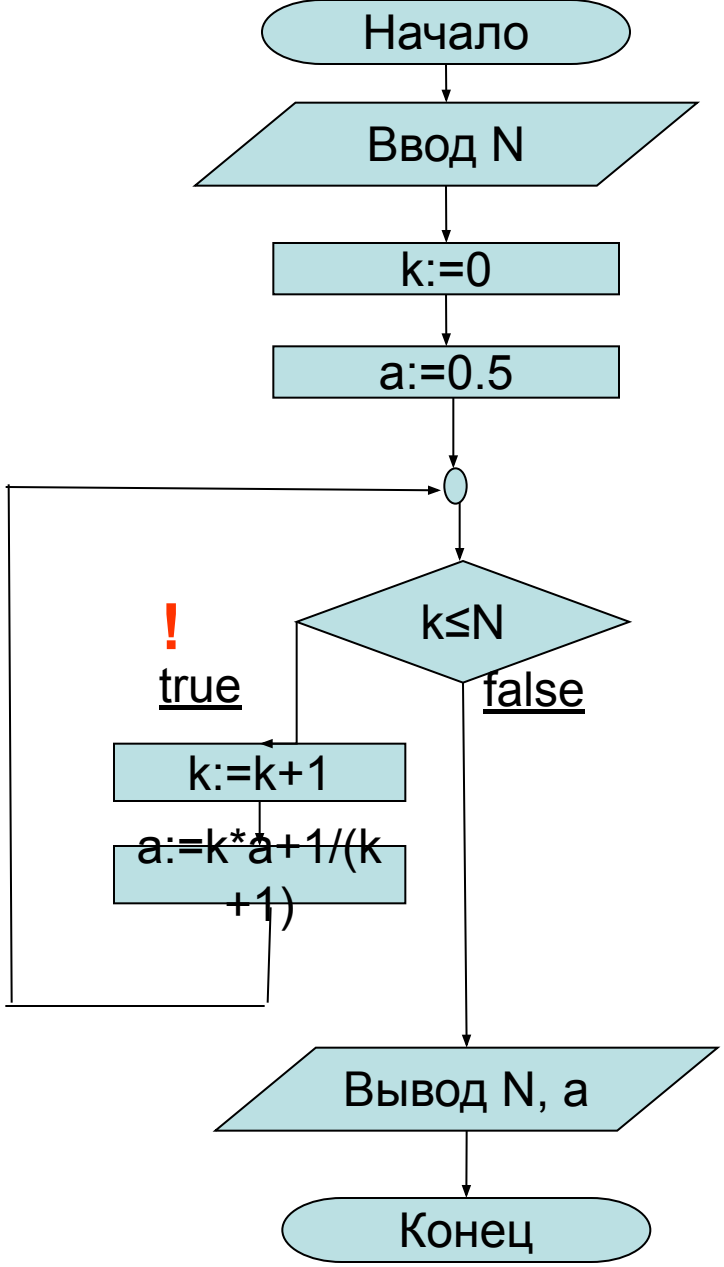
(k=5)< (N=5) ⇒ false

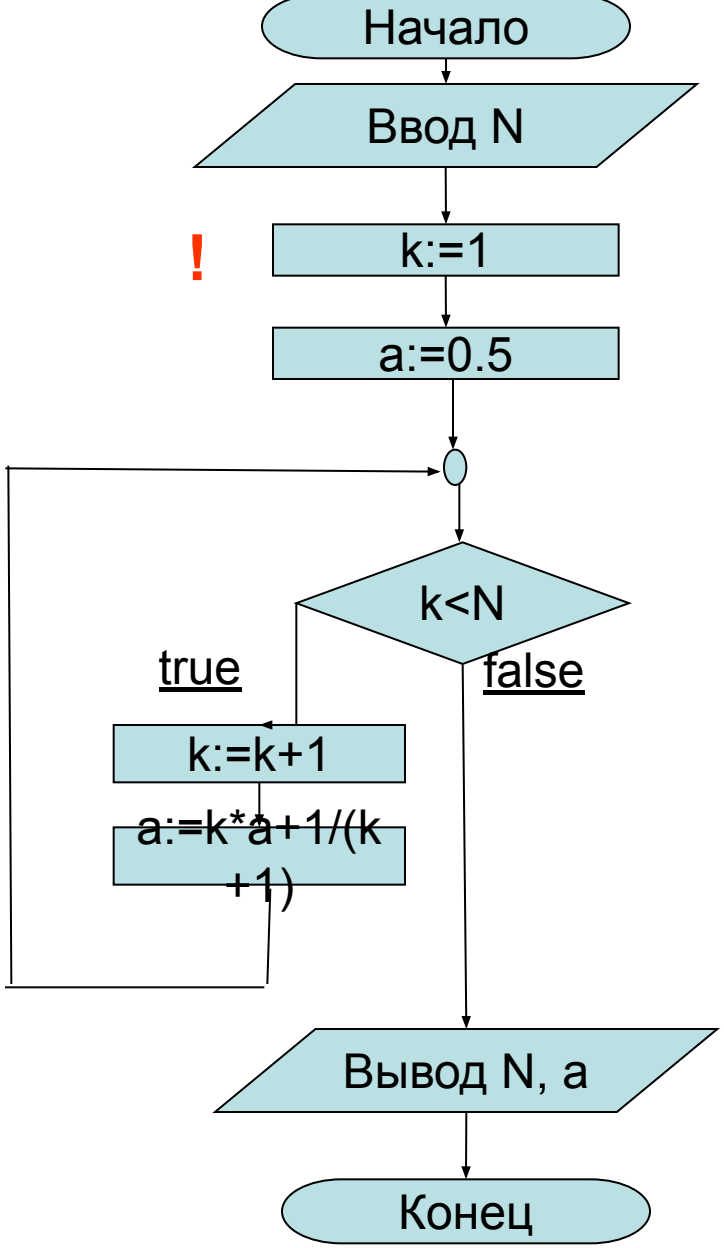
Результат:

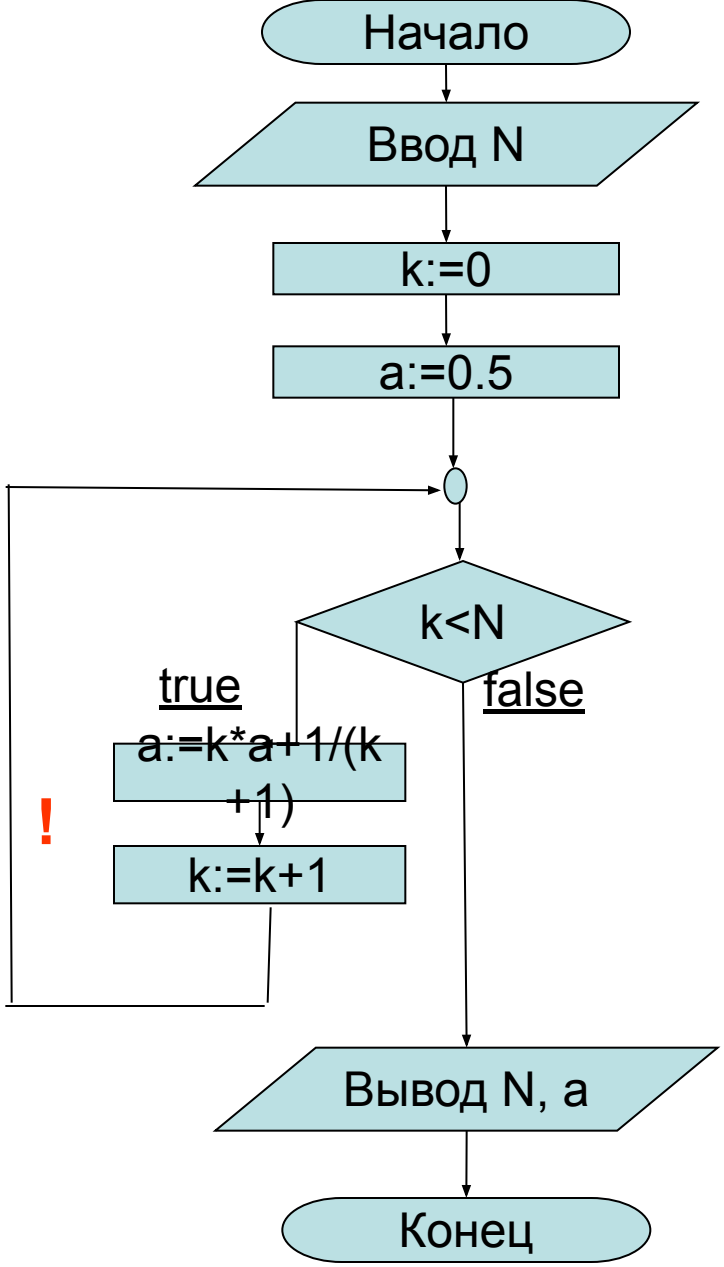
N	a
5	146, 17

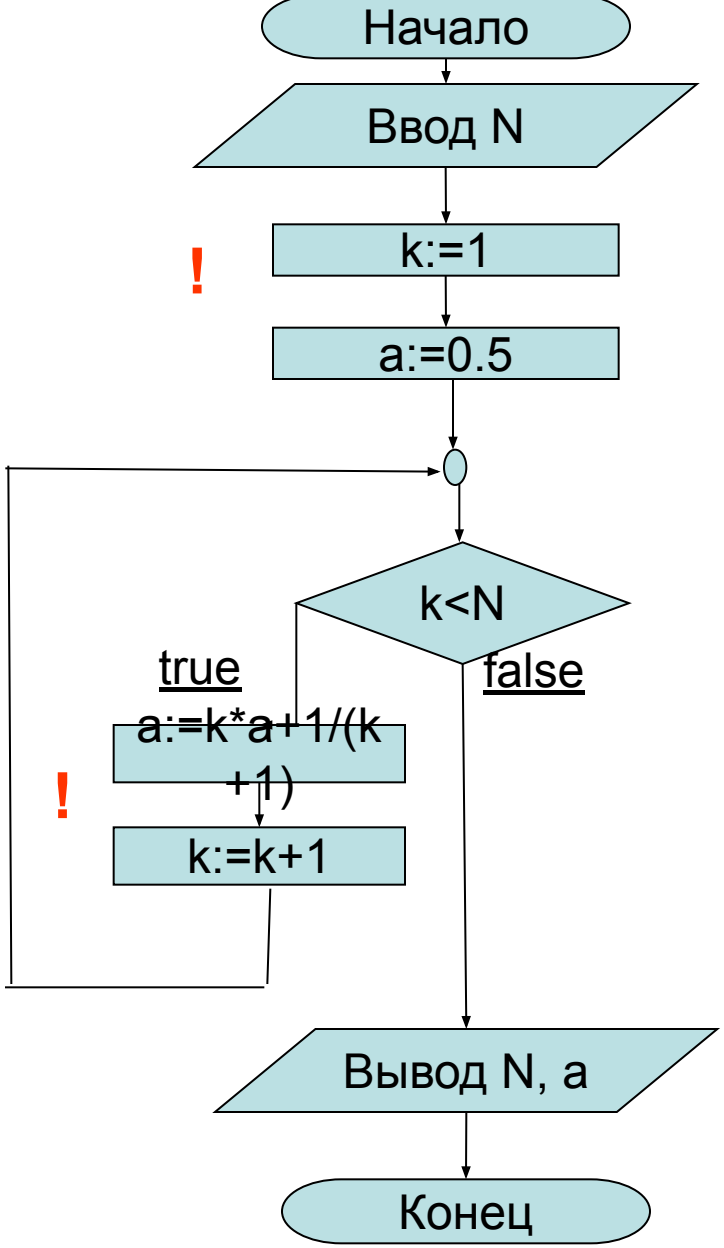


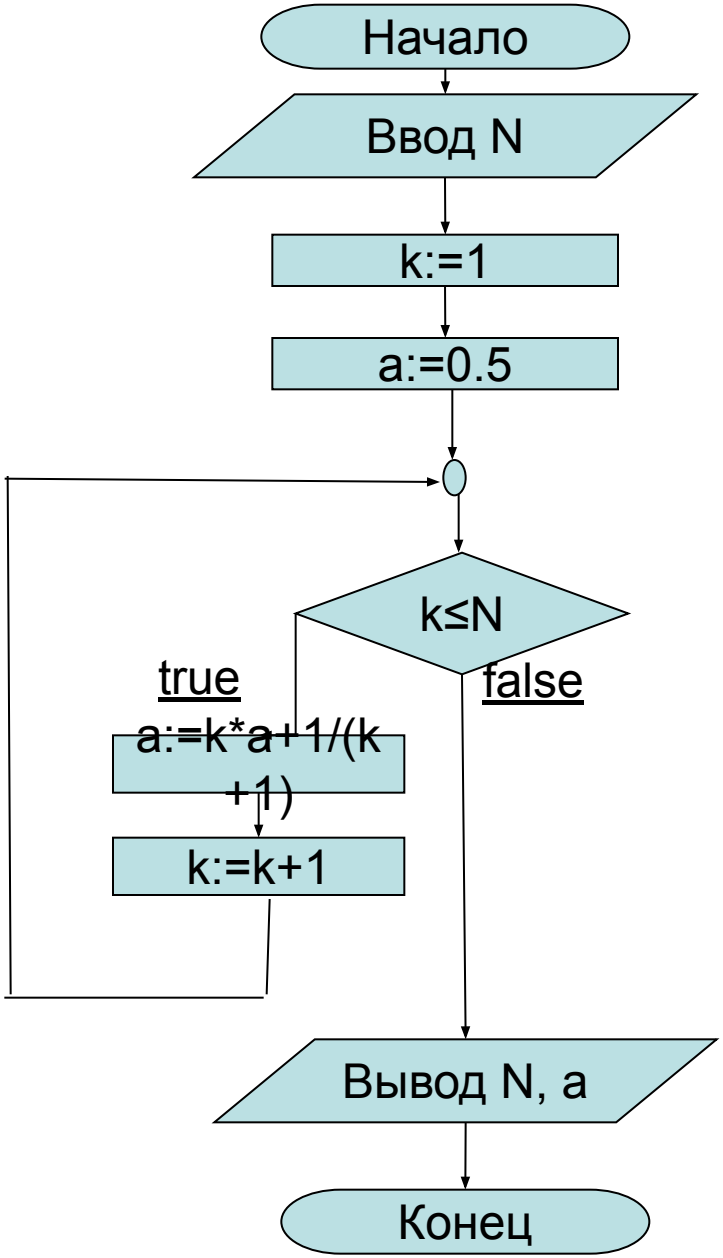

```
     N,k: integer; a:real;
begin
     read(n);
     {Подготовка к циклу}
     k:=0;
     a:=0.5;

     while k<N do
          begin

               a:=k*a+1/(k+1);{вычислени
               очередного элемента}
          end;
     write(N,a)
end.
```


Начало
Ввод N
k:=0
a:=0.5
!
k≤N
true
false
k:=k+1
a:=k*a+1/(k+1)
Вывод N, a
Конец


Начало
Ввод N
!
k:=1
a:=0.5
k<N
true
false
k:=k+1
a:=k*a+1/(k+1)
Вывод N, a
Конец


Начало
Ввод N
k:=0
a:=0.5
k<N
true
false
a:=k*a+1/(k+1)
k:=k+1
!
Вывод N, a
Конец


Начало
Ввод N
k:=1
a:=0.5
k<N
true
false
a:=k*a+1/(k+1)
k:=k+1
!
!
Вывод N, a
Конец




```
    N,k: integer; a:real;
begin
    read(n);
    {Подготовка к циклу}
    k:=1;
    a:=0.5;

    while k≤N do
        begin
        a:=k*a+1/(k+1);{вычисление
        очередного элемента}
        {Номер следующего
        элемента}
        k:=k+1;
        end;

    write(N,a)
end.
```

Ещё раз, кратко:

1. Выявить группу повторяющихся действий ─ сформировать тело цикла.
2. Установить условия повторения этих действий.
3. Установить начальные значения величин, участвующих в условии и в теле цикла ─ выполнить инициализацию цикла.

Задача вычисления суммы

$$S = \sum_{i=1}^{N} \frac{1}{i^2+1} = \frac{1}{2} + \frac{1}{5} + \frac{1}{10} + \ldots + \frac{1}{N^2+1}$$

Исходной величиной задачи является переменная ***N*** целого типа — количество слагаемых, величина ***i*** — номер текущего слагаемого — является промежуточной также целого типа, а величина ***S*** — искомая сумма вещественного типа.

Мы можем рассматривать переменную ***S*** как текущее значение накапливаемой суммы, вычислять и по очереди добавлять к ней каждое слагаемое. Тогда после добавления последнего слагаемого значение ***S*** будет представлять собой искомый результат

Итак, тело цикла образуют следующие действия: вычисление очередного слагаемого, добавление его к текущему значению ***S*** и переход к следующему слагаемому.

Условие, при котором выполняются эти действия — наличие еще не вычисленных и не добавленных слагаемых.

До начала вычислений уже накопленное значение суммы равно нулю, а накапливать сумму можно начать с первого слагаемого.

```
var S,a: real; N,I :integer;
begin
    read (N);   {Ввод количества слагаемых}
     {Инициализация цикла}
    S:=0; {Начальное значение накапливаемой суммы}
    i:=1; {накопление суммы с первого слагаемого }
    while  i≤N do  {пока есть еще не рассмотренные слагаемые}
        begin
            a:=1/(i*i+1);   {вычисление очередного слагаемого }
            S:=S+a;       {добавление очередного слагаемого}
            i:=i+1 {переход к следующему слагаемому}
        end;
    write (N,S) {Вывод результатов}
end.
```

```
//Си-подобный
main ( )
{
   int  n; float a;
   scan (n);  // Ввод количества слагаемых n
   // Инициализация цикла
  float S=0.0; //Начальное значение суммы
  int  i=1; //Номер начального слагаемого
  while (i<=n)  //Перебор слагаемых
      {
         a=1.0/(i*i+1.0); //Очередное слагаемое
         S+=a;           //Добавление слагаемого
          i++; //Номер  следующего слагаемого
       }
// Вывод количества слагаемых и их суммы
   print (n, S);
}
```

В данном случае можно не использовать отдельную переменную для очередного слагаемого.

```
    S: real; N,i:integer;
begin
    read (N);   {Ввод количества слагаемых}
     {Инициализация цикла}
    S:=0; {Начальное значение накапливаемой суммы}
    i:=1; {накопление суммы с первого слагаемого }
    while  i≤N do {пока есть еще не рассмотренные слагаемые}
        begin
            S:=S+1/(i*i+1);  {добавление очередного слагаемого}
            i:=i+1 {переход к следующему слагаемому}
        end;
    write (N,S) {Вывод результатов}
end.
```

В данном случае в качестве начального значения ***S*** можно выбрать первое слагаемое. Тогда циклический процесс накопления суммы нужно начинать со второго слагаемого.

```
    S: real; N,i:integer;
begin
    read (N);   {Ввод количества слагаемых}
     {Инициализация цикла}
!   S:=0.5; {Начальное значение накапливаемой суммы}
!   i:=2; {накопление суммы со второго слагаемого }
    while  i≤N do {пока есть еще не рассмотренные слагаемые}
        begin
            S:=S+1/(i*i+1);  {добавление очередного слагаемого}
            i:=i+1 {переход к следующему слагаемому}
        end;
    write (N,S) {Вывод результатов}
end.
```

В качестве начального значения ***i*** можно использовать нуль, а не единицу. Это повлечет за собой необходимость изменить строгое неравенство в заголовке цикла на нестрогое, а также изменить порядок выполнения действий в теле цикла.

```
    S: real; N,i:integer;
begin
    read (N);   {Ввод количества слагаемых}
     {Инициализация цикла}
    S:=0; {Начальное значение накапливаемой суммы}
!   i:=0; {начальное значение номера}
!   while  i<N do {пока есть еще не рассмотренные слагаемые}
        begin
!         i:=i+1 {получение номера текущего слагаемого}
          S:=S+1/(i*i+1);  {добавление очередного слагаемого}
        end;
    write (N,S) {Вывод результатов}
end.
```

В данном случае можно начать накопление суммы с последнего слагаемого и двигаться к первому, переходя от текущего к предыдущему

```
        S: real; N,i:integer;
    begin
        read (N);   {Ввод количества слагаемых}
         {Инициализация цикла}
        S:=0; {Начальное значение накапливаемой суммы}
!       i:=N; {накопление суммы с последнего слагаемого }
!       while  i≥1 do {пока есть еще не рассмотренные слагаемые}
            begin
                S:=S+1/(i*i+1);  {добавление очередного слагаемого}
!               i:=i-1 {переход к предыдущему слагаемому}
            end;
        write (N,S) {Вывод результатов}
    end.
```

Суммирование с рекуррентным соотношением между слагаемыми

$$S = \sum_{i=0}^{N} \frac{1}{i!} = 1 + \frac{1}{1} + \frac{1}{1\times 2} + \frac{1}{1\times 2\times 3} + \frac{1}{1\times 2\times 3\times 4} \ldots + \frac{1}{1\times 2\times 3\times \ldots \times N}$$

Пусть a_i — очередное i слагаемое $a_i = 1/i!$. Тогда

$$\frac{a_i}{a_{i-1}} = \frac{1\times (i-1)!}{i!\times 1} = \frac{1}{i},$$

Таким образом, для слагаемых получено следующее рекуррентное соотношение:

$$a_i = a_{i-1} / i, i = 1,2,\ldots; a_0 = 1$$

Для накопления суммы переменной ***S*** присвоим значение нулевого слагаемого, а циклическое накопление начнем с первого. Кроме того, учтем, что в данном случае потребуется явное использование переменной ***a***, служащей для хранения значения очередного слагаемого

```
    S,a: real; N,i:integer;
begin
    read (N);   {Ввод количества слагаемых}
    {Инициализация цикла}
    S:=1. {Начальное значение накапливаемой суммы}
    a:=1; {Нулевое слагаемое}
    i:=1; {накопление суммы с первого слагаемого }
    while  i≤N do {пока есть еще не рассмотренные слагаемые}
        begin
            a:=a/i; {Рекуррентное определение очередного слагаемого}
            S:=S+a;  {добавление очередного слагаемого}
            i:=i+1 {переход к следующему слагаемому}
        end;
    write (N,S) {Вывод результатов}
end.
```

```
//Си-подобный
main ( )
{
   int  n;
   scan (n);  // Ввод количества слагаемых n
   // Инициализация цикла
  float S=1.0; //Начальное значение суммы, включено нулевое слагаемое
  float a=1.0; //Первое слагаемое
  int i=1; //Номер начального слагаемого
  while (i<=n)  //Перебор слагаемых
      {
         a/=i;         //Очередное слагаемое a=a/i;
         S=+a;         //Добавление слагаемого S=S+a;
          i++;         //Номер  следующего слагаемого i=i+1;
       }
// Вывод количества слагаемых и их суммы
   print (n, S);
}
```

Суммирование произвольных слагаемых

$$S = \sum_{i=1}^{N} x_i = x_1 + x_2 + x_3 + \ldots + x_N$$

```
    x: array of real; S: real; N,i: integer;
begin
    read (N, x); {Ввод количества слагаемых и всех элементов массива}
     {Инициализация цикла}
    S:=0; {Начальное значение накапливаемой суммы}
    i:=1; {накопление суммы с первого слагаемого }
    while  i≤N do  {пока есть еще не рассмотренные слагаемые}
          begin
              S:=S + x[i];       {добавление очередного слагаемого}
              i:=i+1 {переход к следующему слагаемому}
          end;
    write (N,S) {Вывод результатов}
end.
```

Нахождение экстремального элемента массива

Пусть задан массив $x = \{x_1, x_2, \ldots, x_n\}$. В простейшем случае необходимо найти его наибольший (наименьший) элемент.

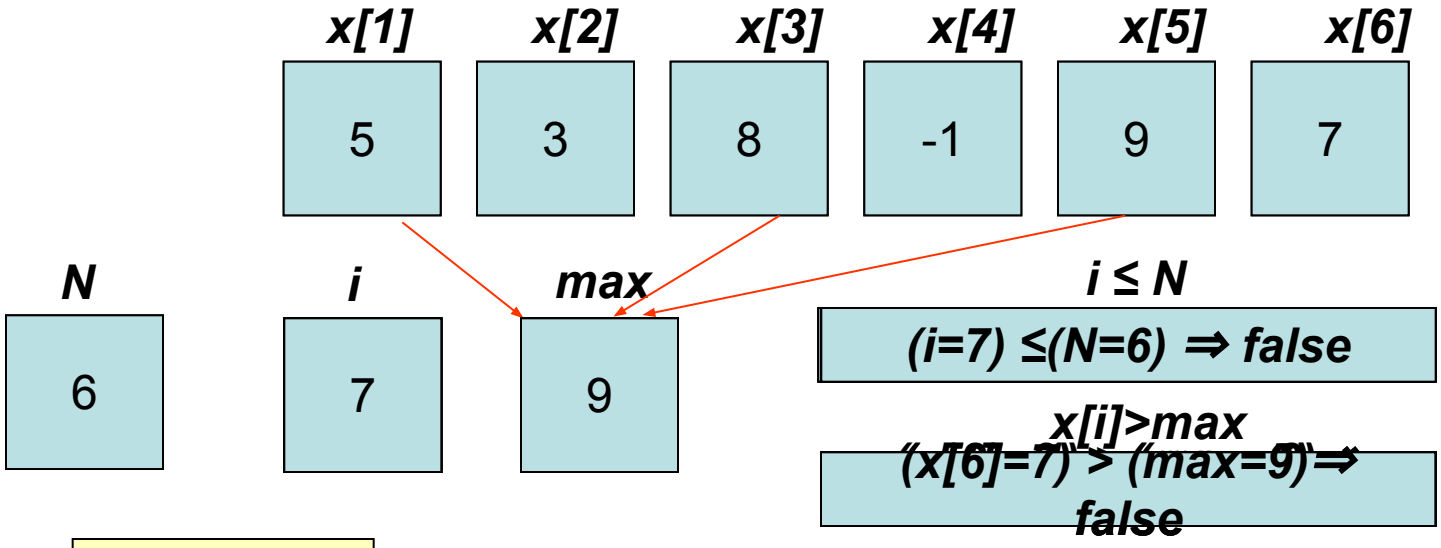
Пусть ***max*** — переменная, которая играет роль «кандидата на должность» наибольшего элемента, ***i*** — номер очередного элемента массива. Если элементы массива, относятся к целому типу, то и ***max*** тоже величина целого типа.

```
    x: array of integer; i, N, max: integer;
begin
    read (N, x); {ввод размерности и значений элементов массива}
    max:=x[1]; {назначение кандидатом первого элемента массива}
    i:=2: {перебор остальных элементов массива, начиная со второго}
     while i ≤ N do {пока есть ещё не рассмотренные элементы}
        begin
           if x[i]>max then max:=x[i]; {сравнение очередного с кандидатом}
           i:=i+1 {переход к следующему элементу массива}
        end;
    write (max)
end.
```

```
// Си-подобный
main ( )
{

    int  n, a[ ];
// Ввод размерности
n      и      элементов
векторов
   scan (n, a);
     int  MaxElem=a[1];
//Назначение
«кандидата»
                int      i=2;
//Начальный номер
перебора
   while (i<=n)
       {
        if (a[i]>MaxElem)
//Очередной лучше

MaxElem=a[i];
         i++;
        }
    print (MaxElem);
}
```

Проведем тестирование алгоритма на примере ***x={5, 3, 8, -1, 9, 7}***

x: array of integer; N, i, max: integer;




```
read (N, x); {ввод размерности и значений элементов массива}
max:=x[1]; {назначение кандидатом первого элемента массива}
i:=2: {перебор остальных элементов массива, начиная со второго}
  while i ≤ N do {пока есть ещё не рассмотренные элементы}
    begin
      if x[i]>max then max:=x[i]; {сравнение очередного с кандидатом}
      i:=i+1 {переход к следующему элементу массива}
    end;
write (max)
```

```
main ()
 {
        int  n, x[ ];
        scan (n, x); //ввод размерности и значений элементов массива
        Int max=x[1]; //назначение кандидатом первого элемента массива
         int i=2; //перебор остальных элементов массива, начиная со второго
         while (i <=n) //пока есть ещё не рассмотренные элементы
             {
              if (x[i]<max) max=x[i]; //очередной элемент меньше кандидата - замена
                 i++; //переход к следующему элементу массива
             }
        print (max);
 }
```

Определение номера экстремального элемента

```
main ()
 {
        int n,  x[ ];
        scan (n, x); //ввод размерности и значений элементов массива
        int Nmax=1; //назначение кандидатом первого элемента массива
        int i=2; //перебор остальных элементов массива, начиная со второго

         while (i <= n) //пока есть ещё не рассмотренные элементы
              {
                 if (x[i]>x[Nmax]) Nmax=i; //очередной лучше (больше) - заменяем
                  i++; //переход к следующему элементу массива
              }
        print (x[Nmax]);
 }
```

В последнем варианте построения алгоритма результат его выполнения *при наличии одинаковых элементов* в массиве оказывается зависящим от *строгости знака неравенства* в условном операторе. В то время как в первоначальном варианте, когда требовалось найти сам экстремальный элемент, результат от строгости этого знака не зависит.

Чтобы убедиться в этом можно взять, например, такой набор входных данных n=6 и x=(4, 9, 2, 9, 0, 3). Непосредственно видно, что в этом массиве имеется два одинаковых максимальных: второй и четвёртый элементы имеет значение 9.

Если искать только значение максимального, то ответ в любом случае один и тот же — 9. Если же требуется не само значение, а его номер, то ответом может быть и 2 и 4.

Обсудим теперь зависимость результата от строгости знака. Пусть сравнение в цикле осуществляется оператором if (x[i]>x[Nmax]) Nmax=i, условие которого содержит *строгое* неравенство x[i]>x[Nmax].

Тогда в результате выполнения алгоритма ответом будет номер *первого* среди максимальных Nmax=2. В самом деле, при i=4 неравенство x[i]>x[Nmax], в котором сравниваться x[4]=9 и a[Nmax]=9, не выполняется, поэтому замены значения Nmax не произойдёт.

Если же в условном операторе использовать *нестрогое* неравенство x[i]>=x[Nmax], которое при i=4 выполняется, замена значения Nmax произойдёт и алгоритм выдаст в качестве ответа номер *последнего* среди максимальных Nmax=4.

Определение экстремального элемента и его номера

```
main ()
        int i, n, max, Nmax , x[ ];
         scan (n, x); //ввод размерности и значений элементов массива
        max=x[1]; Nmax=1; //назначение кандидатов

        i=2; //перебор остальных элементов массива, начиная со второго
         while (i <= n) //пока есть ещё не рассмотренные элементы
              {
                 if (x[i]<x[Nmax])
                    {
                       max=x[i]; Nmax=i; //очередной лучше – заменяем
                    }
                 i++; //переход к следующему элементу массива
              }
        print (max, Nmax);
}
```

Во всех рассмотренных выше вариантах алгоритма в качестве «кандидата» во время инициализации цикла выбирался первый элемент массива, а перебор начинался со второго элемента. Вообще говоря, организация цикла может быть и другой.

В частности, начинать перебор можно как обычно с первого элемента массива. Отличие этого варианта только в том, что во время выполнения цикла произойдет одно *лишнее* сравнение «кандидата» с самим собой. На конечный результат это дополнительное сравнение никак не повлияет.

Кроме первого элемента на этапе инициализации «кандидатом» может быть назначен *любой* элемент массива. Это также приведёт к одному *лишнему* сравнению и не повлияет на конечный результат.

Следует понимать, что выбор начального значения «кандидата» среди элементов массива возможен только том случае, если эти элементы *уже известны*.

То есть, чтобы назначить «кандидата» во время инициализации цикла хотя бы один элемент массива должен быть уже известен.

Если до начала вычислений элементы массива не известны можно предложить другой способ выбора «кандидата». А именно «кандидатом» назначается *любое значение*, которое окажется заведомо «хуже» *любого* возможного значения элемента массива. Такой заведомо худший «кандидат» сразу же заменится первым же вычисленным элементом массива.

Например, при вычислении *наибольшего* среди положительных чисел, в качестве «кандидата» можно взять нуль или любое отрицательное число. А при вычислении наименьшего среди отрицательных «кандидатом» может быть любое положительное число.

Задача поиска

Общая постановка. В заданной совокупности элементов требуется найти элемент, обладающий заданным набором свойств.

Важнейший частный случай. Задан массив ***x***, нужно определить имеет ли в нем элемент, совпадающий с заданным числом ***a.*** Если такой элемент обнаружен, то в качестве дополнительной информации следует получить номер совпадающего с заданным элемента массива

x[1]	***x[2]***	***x[3]***	***x[4]***	***x[5]***	***x[6]***	***x[7]***	***x[8]***
43	55	82	42	94	18	6	63

a

25

Ответ: В рассматриваемом массиве ***x*** элемента, совпадающего с заданным ***a =25*** нет.

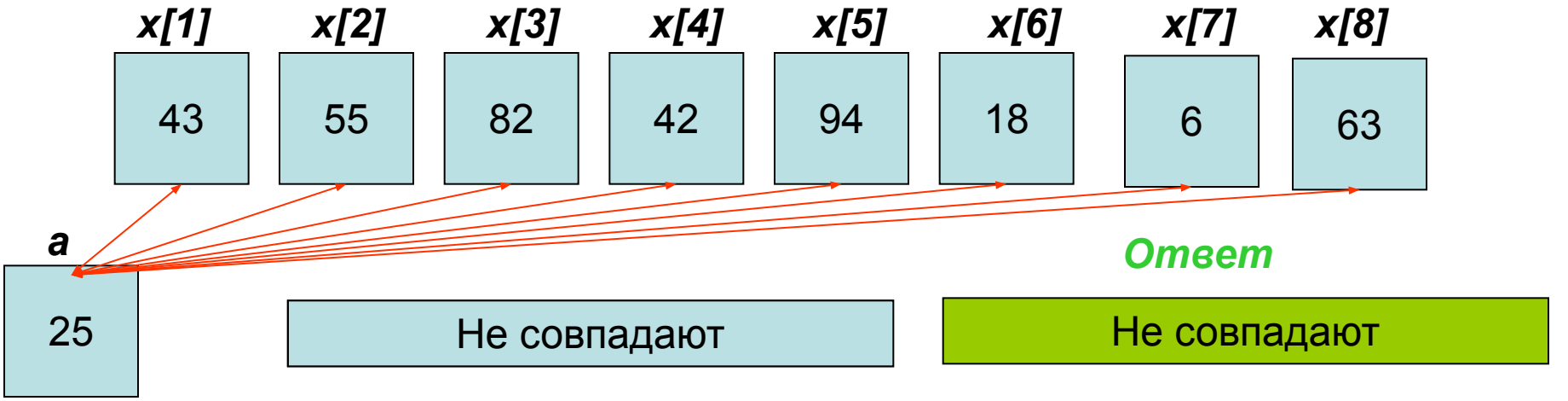
a

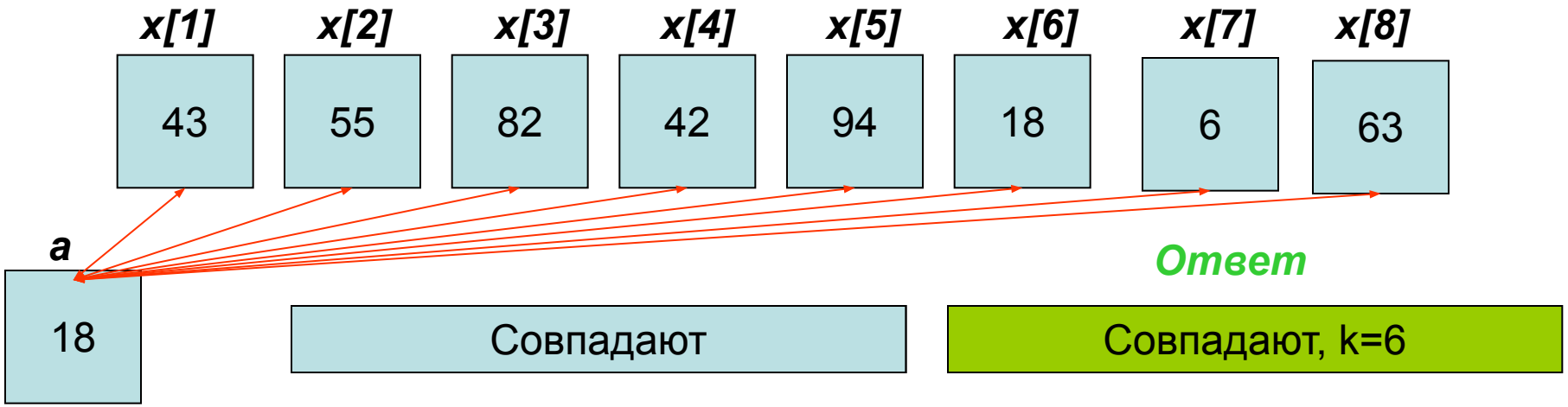
18

Ответ: В рассматриваемом массиве ***x*** есть элемент, совпадающий с заданным ***a=18***. Его номер ***k=6***.

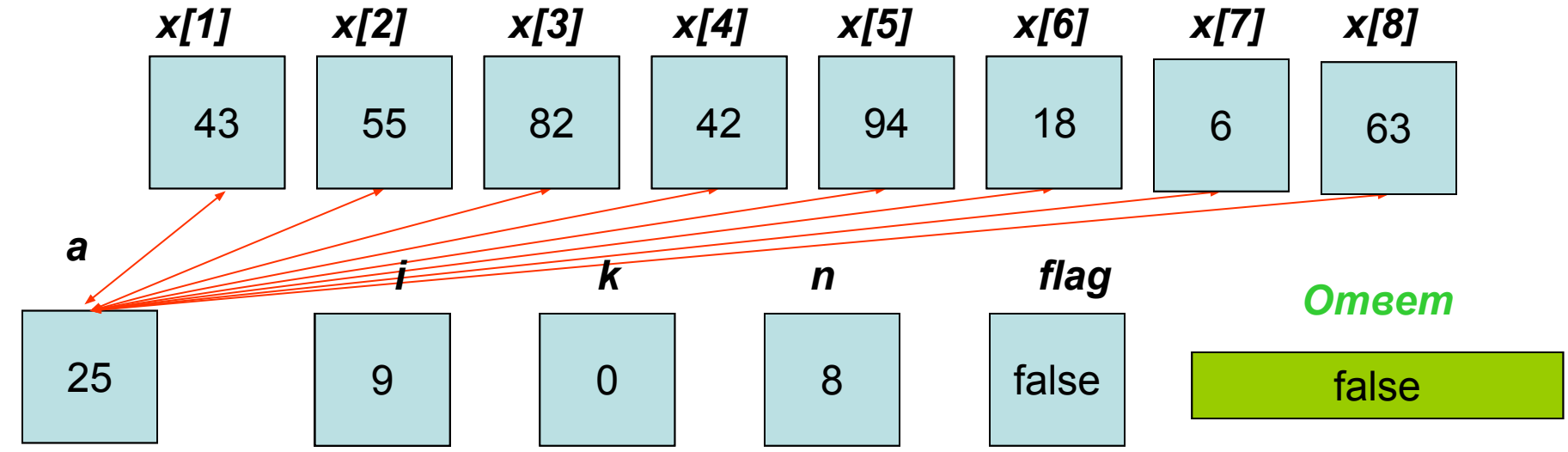
а)Классический алгоритм

Предлагается очевидный подход к решению задачи: последовательное сравнение каждого очередного элемента массива с заданным. Такой способ называется ***линейным*** поиском.





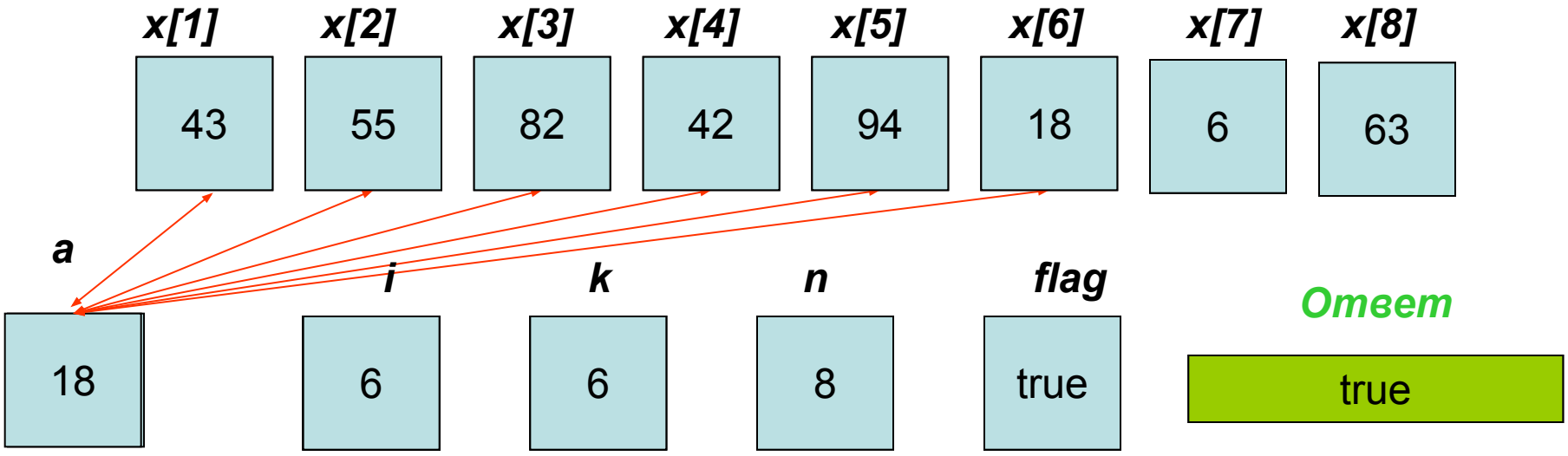
```
x :array of integer; a, k, i,n :integer; flag: boolean;
begin
 read(n,x,a); {ввод количества элементов в массиве, самих элементов массива  и искомого элемента }
 i:=1; {поиск начинается с первого элемента}
 flag:=false; k:=0; {элемент не найден}
 while (i<=N) and not flag do {Поиск продолжается}
    if  x[i]=a then   begin {элемент найден} k:=i; flag:=true end
      else {переход к следующему элементу} i:=i+1;
 {вывод результатов flag и k}
```



((i=9) ≤ (n=8)) ∧ not flag ⇒ false

(x[i]=63) = (a=25) ⇒ false

```
 x :array of integer; a, k, i,n :integer; flag: boolean;
begin
 read(n,x,a); {ввод количества элементов в массиве, самих элементов
массива  и искомого элемента }
 i:=1; {поиск начинается с первого элемента}
 flag:=false; k:=0; {элемент не найден}
 while (i<=N) and not flag do {Поиск продолжается}
    if  x[i]=a then   begin {элемент найден} k:=i; flag:=true end
      else {переход к следующему элементу} i:=i+1;
 {вывод результатов flag и k}
```



Ответ

true

((i=6) ≤ (n=8)) ∧ not flag ⇒ false

(x[i]=18) = (a=18) ⇒ true

б)Быстрый алгоритм

Идея быстрого алгоритма состоит в том, что выражения ***flag*** и ***x[i]=a*** эквивалентны. Следовательно, в условии в заголовке цикла вместо ***not flag*** можно использовать ***x[i]<>a***, и отказаться от использования индикатора ***flag***.

```
i:=1;
while (i<=N) and (x[i]<>a) do i:=i+1; {пока номер не превосходит последний и искомый элемент не найден перейти к следующему элементу}
```

в)Алгоритм с барьером

Следующее улучшение состоит в упрощении условия в заголовке цикла. Целевым является условие x[i]<>a, которое убрать или упростить невозможно. Следовательно, можно рассматривать только условие i<=N, обеспечивающие прекращение просмотра после исчерпания всех элементов цикла. Идея состоит в выставлении в конце массива «барьера». Для этого в массив добавляется N+1 элемент, значение которого приравнивается к искомому.

```
i:=1; x[N+1]:=a; while x[i]<>a do i:=i+1
```

В алгоритмах линейного поиска максимальное количество операций, которое приходится проделать для определения результата поиска пропорционально ***N***.

г)Бинарный поиск.

Дальнейшее улучшение алгоритма поиска, который можно было бы использовать для работы с *произвольными* массивами, *принципиально невозможно*. Это связано с тем, что в любом из массивов, которые могут поступить на вход алгоритма, искомый элемент может отсутствовать. А для произвольного массива это означает, что в процессе поиска потребуется просмотреть все его элементов

Областью поиска называется группа элементов массива, среди которых производится поиск. Для *произвольного* массива область поиска совпадает со всем массивом.

Существуют алгоритмы, в которых область поиска на каждом шаге выполнения *уменьшается*. Но для их использования массив, в котором производится поиск, должен быть *упорядоченным*.

Массив x называется **отсортированным** или **упорядоченным**, если для *всех его* элементов выполняется какое-либо одно из неравенств: $x[i]<x[i+1]$, $x[i]\le x[i+1]$, $x[i]>x[i+1]$ или $x[i]\ge x[i+1]$, $\forall$ $i=1,2,\ldots,n-1$, где n — количество элементов в массиве,

Если выполняется неравенство $x[i]\le x[i+1]$, $\forall$ $i=1,2,\ldots,n-1$, то массив называется *упорядоченным* (отсортированным) *по возрастанию*.

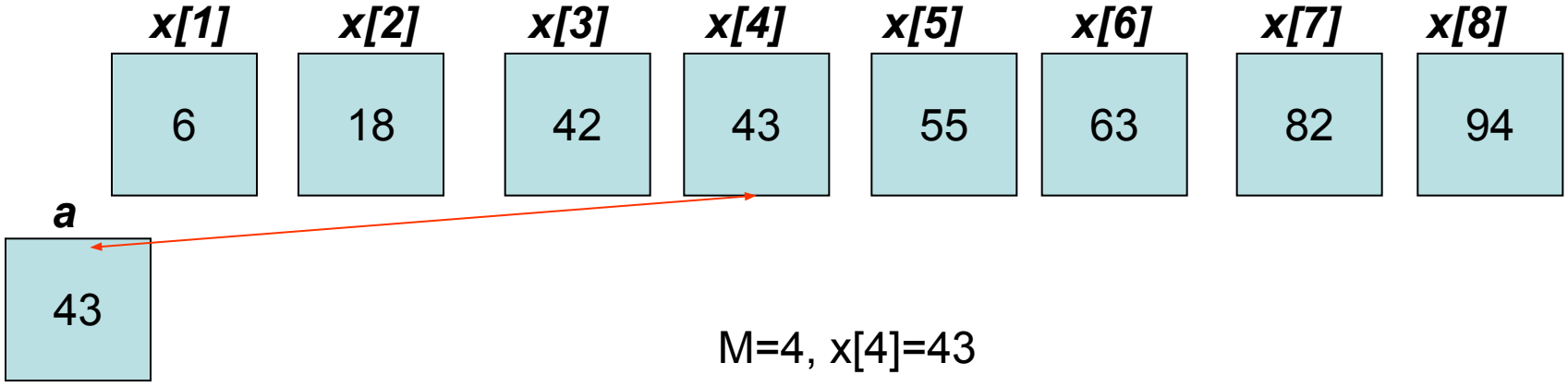
Если выполняется неравенство $x[i]\ge x[i+1]$, $\forall$ $i=1,2,\ldots,n-1$, то массив называется *упорядоченным* (отсортированным) *по убыванию*.

Свойство упорядоченности у элементов массива обеспечивает возможность использования общематематического приёма *деления пополам*, с помощью которого область поиска на каждом шаге *уменьшается в два раза*. В соответствии с этим приёмом поиск происходит по следующей схеме.

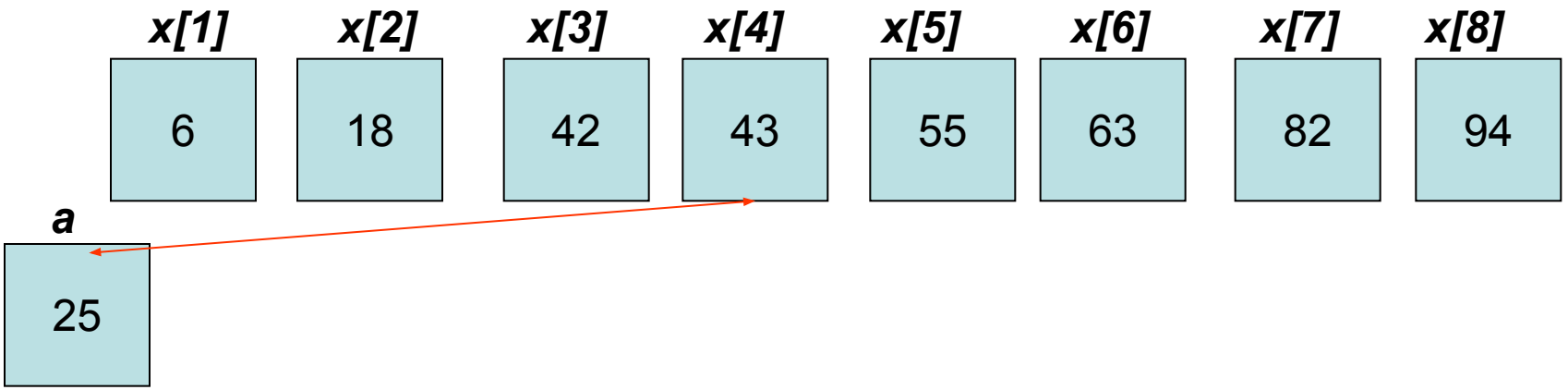
1. Выбирается один из элементов массива, который объявляется «средним» (то есть, находящимся примерно посередине списка элементов массива).

Существуют различные способы выбора такого элемента. В простейшем варианте «средний» элемент выбирается точно так же как вычисляется координата *c* середины отрезка [*a*, *b*]: *c*=(*a*+*b*)/2. Так как номера элементов могут быть только целыми, то номер M выбранного «средним» элемента массива, состоящего из n элементов, можно найти с помощью оператора M:=(n+1) div 2.

2. Выбранный элемент x[M] сравнивается с искомым a и если x[M]=a, то задача поиска считается решённой с положительным результатом и M есть номер найденного элемента.



M=4, x[4]=43



3. При несовпадении элементов, опираясь на упорядоченность массива, в дальнейшем поиске можно не рассматривать одну из половин массива.

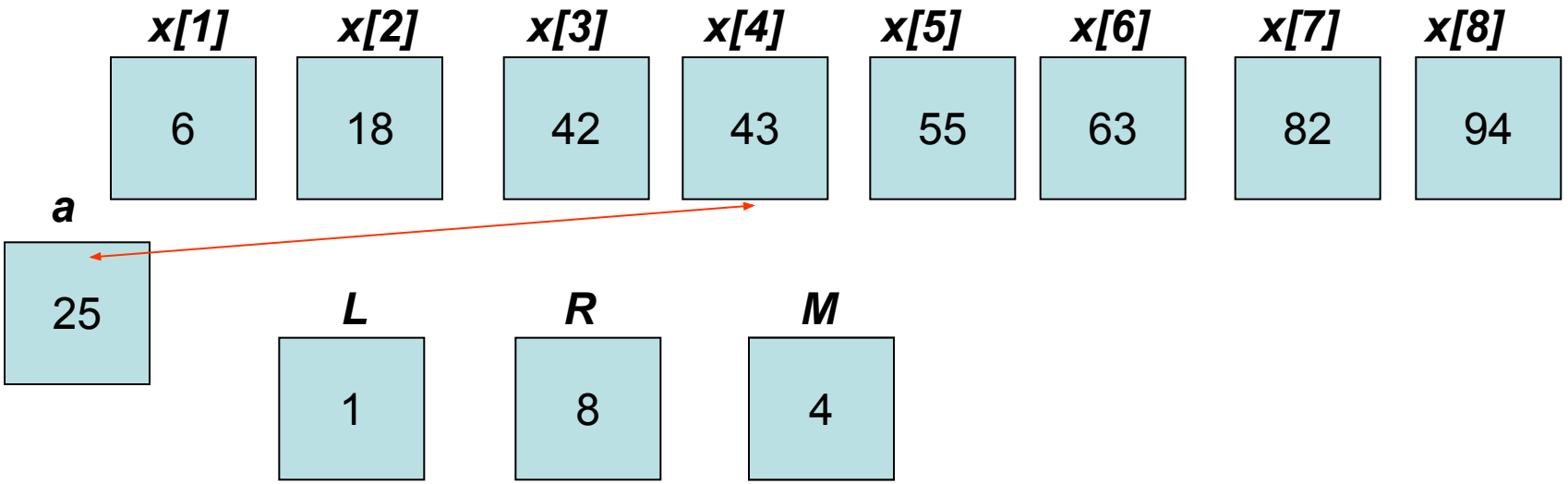
А именно, если средний элемент массива больше искомого, то все элементы, расположенные правее его, также будут больше искомого и дальнейший поиск следует осуществлять в левой половине массива.

В противном случае, если средний элемент массива меньше искомого, то все элементы, расположенные левее, будут также меньше его и дальнейший поиск следует выполнять в правой половине массива.

Действия пунктов 1–3 применяются к выбранной для продолжения поиска части массива, затем к половине этой части и т.д., до тех пор, пока искомый элемент не будет найден, или же область поиска не окажется пустой. В последнем случае задача поиска считается решённой с отрицательным результатом.

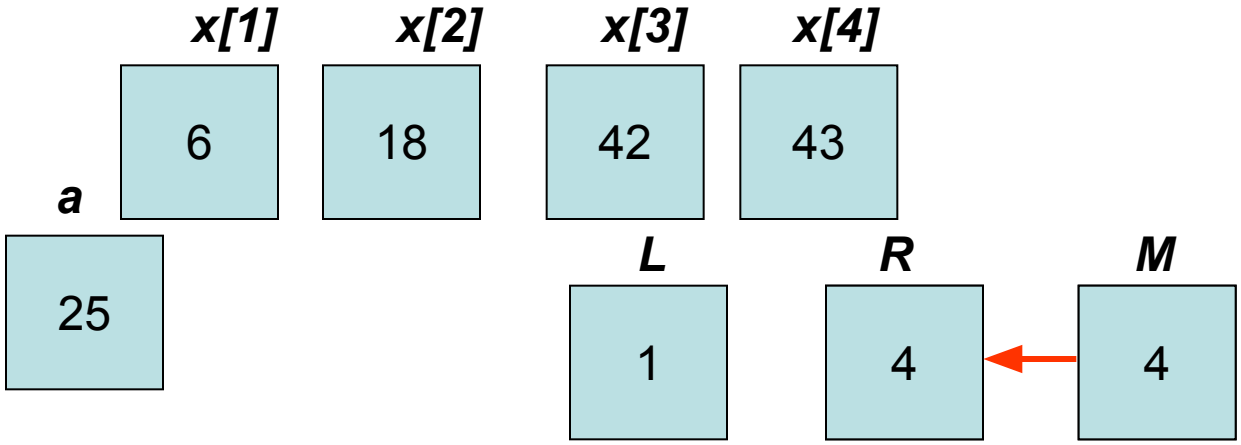
Итак, во время поиска приходится повторять следующие действия: 1) выбирать средний элемент массива; 2) сравнивать его с искомым; 3) при необходимости уменьшать рассматриваемую часть массива в два раза выбором его правой или левой половины.

Так как начальный и конечный элементы рассматриваемой части массива при отбрасывании одной из половин массива будут изменяться, то для записи алгоритма следует использовать переменные, имеющие смысл номеров начального (левого) ***L*** и конечного (правого) ***R*** элементов массива. Поскольку в начале рассматривается весь массив, то ***L=1***, а ***R=N***. Номер среднего элемента ***M*** можно найти стандартным способом: ***M=(R+L) div 2***.

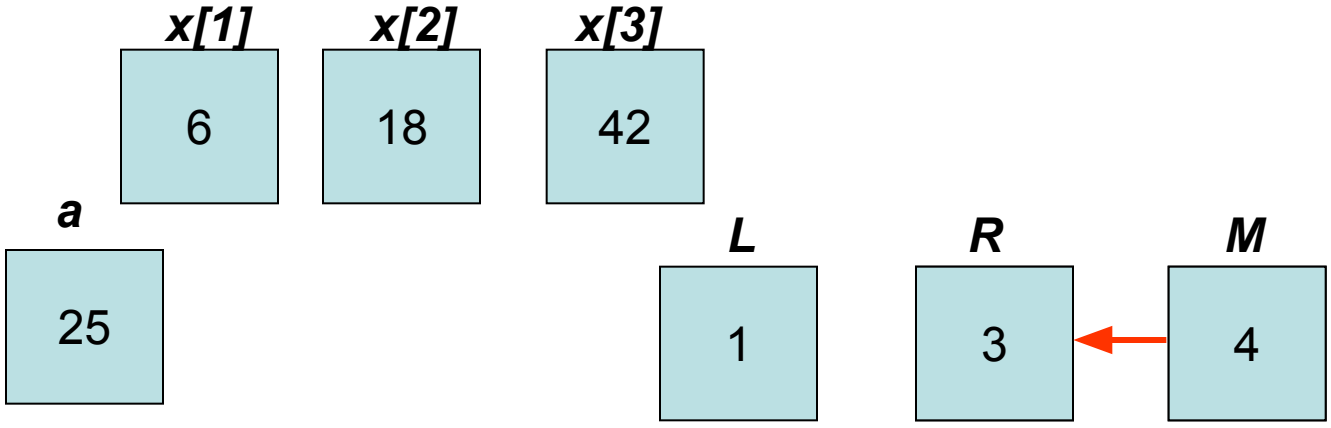


В данном случае выбранный элемент массива не совпадает с искомым. Он больше его. Следовательно для поиска нужно выбрать левую половину таблицы.

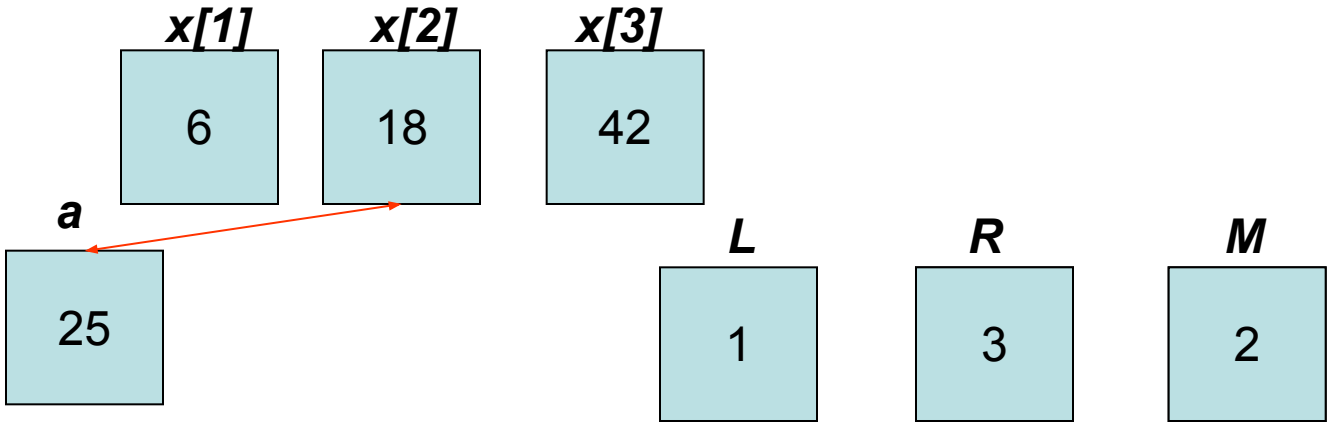
Выбор левой половины массива означает, что его правая граница перемещается к среднему элементу: ***R:=M*** .



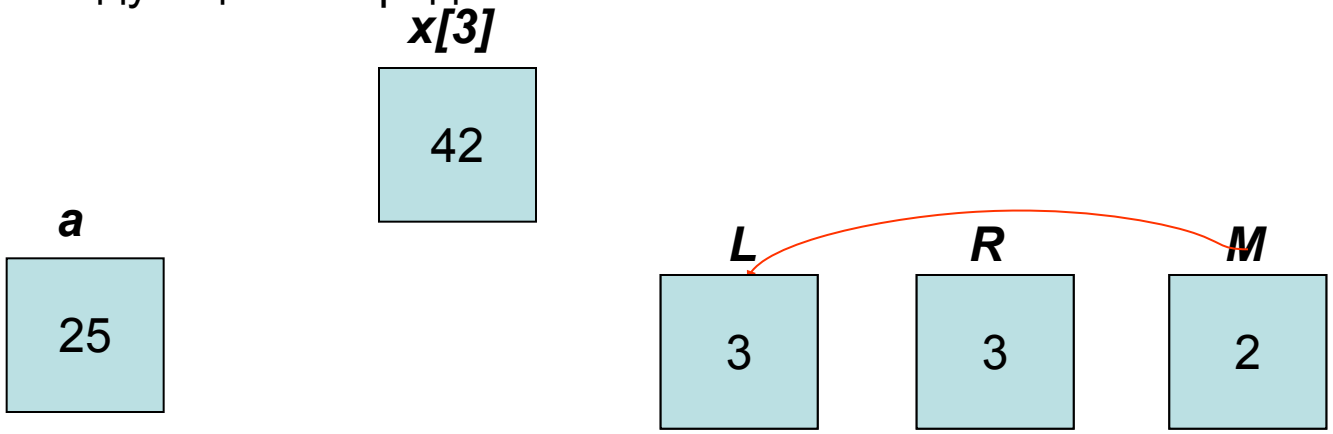
Но уже известно, что средний элемент не совпадает с искомым, следовательно, область поиска можно ёще больше сузить и считать правой границей элемент, стоящий перед средним: ***R:=M-1***



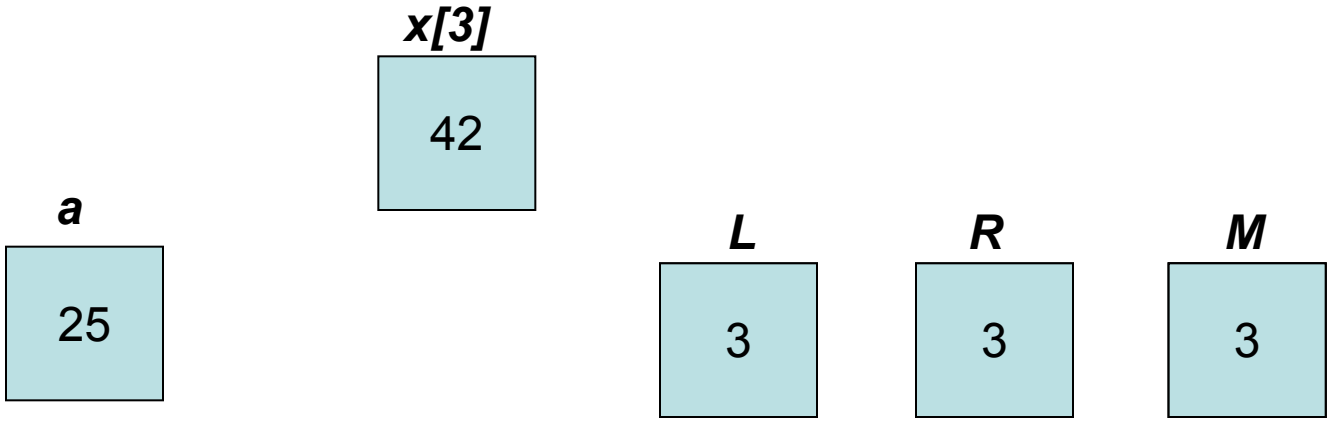
Вновь находим средний элемент рассматриваемой части массива: ***M:=(L+R) div 2***, ***M=2***.



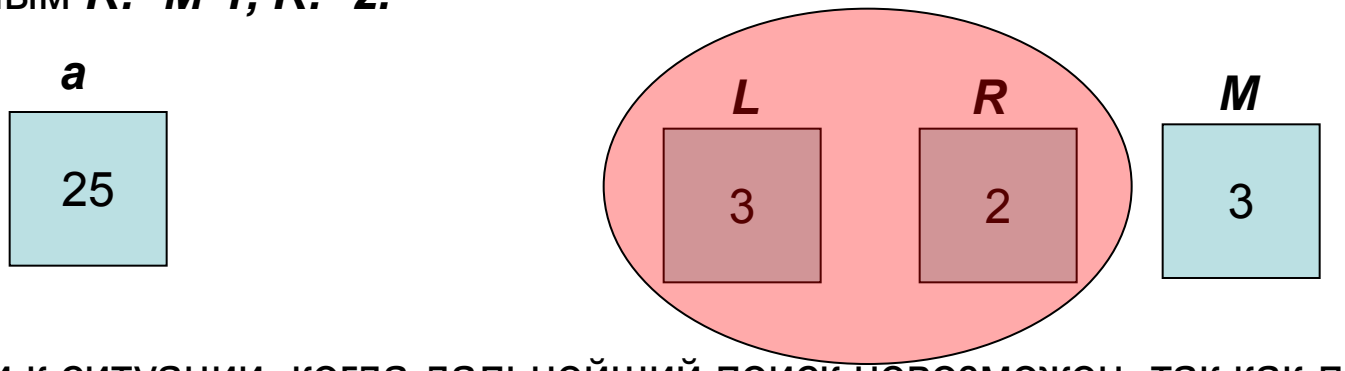
Опять не совпадение среднего элемента с искомым. Так как он меньше искомого (***x[2]=18,a=25)***, то для дальнейшего поиска следует выбрать правую половину массива, сместив левую границу ***L*** к среднему. Учитывая, предыдущее соображение относительно выбранного элемента, получим, что нужно взять следующий за средним ***L:=M+1***



Вновь находим средний элемент рассматриваемой части массива: ***M:=(L+R) div 2***, ***M=3***.



И вновь не совпадение среднего элемента с искомым. Так как выбранный элемент больше искомого (***x[3]=42,a=25)***, то для дальнейшего поиска следует выбрать левую половину массива, сместив правую границу ***R*** к следующему за выбранным ***R:=M-1, R:=2.***



Пришли к ситуации, когда дальнейший поиск невозможен, так как левая граница массива стала больше правой ***L>R*** – массив исчерпан.

Таким образом, повторяющиеся действия состоят в выборе среднего элемента массива ***M:=(L+R) div 2*** и организации соответствующего рассмотренной ситуации ветвления. Повторять эти действия следует пока искомый элемент ещё не найден и при этом левая граница рассматриваемого участка массива не превосходит правую. Начальные значения ***L*** и ***R*** уже обсуждались.

...

```
L:=1; {левая граница отрезка поиска совпадает с 1}
R:=N; {правая граница отрезка поиска совпадает с N}
flag:=false;{элемент еще не найден}
while (L<=R) and not flag do
    begin
        M:=(L+R) div 2; {определение номера среднего элемента}
        if x[M]=a then {элемент найден}
              flag:=true
           else  {элемент не найден}
              if  x[M]<a then {перенос левой границы отрезка поиска}
                    L:=M+1
                 else  {перенос правой границы отрезка поиска}
                    R:=M-1
    end;
{flag — показывает результат поиска.}
```

Эффективность алгоритмов

Нами были построены различные алгоритмы решения одной и той же задачи поиска. При этом, вообще говоря, *бездоказательно*, только на основе внешнего вида алгоритмов утверждалось, что, например, быстрый алгоритм и алгоритм с барьером лучше, эффективнее, чем классический алгоритм.

В связи с этим возникает ряд вопросов. Что понимать под эффективностью алгоритма? Как можно выяснить, какой алгоритм лучше, эффективнее, а какой хуже? Вообще, как можно сравнивать алгоритмы между собой?

В настоящее время алгортмы принято сравнивать друг с другом по двум критериям — затраченному на выполнение алгоритма времени и потребовавшийся для этого выполнения объём памяти. Первый критерий связан с **временной эффективностью** алгоритма, а второй — с **объёмной эффективностью** или **эффективностью по памяти.**

При сравнении нескольких алгоритмов решения одной и той же задачи по критериию *временной эффективности* лучшим считается тот алгоритм, на выполнение которого одним и тем же исполнителем на одном и том же наборе исходных данных требуется *меньшее время*. Аналогичным образом по критерию *объёмной эффективности* лучшим является тот алгоритм, на выполнение которого одним и тем же исполнителем на одном и том же наборе исходных данных потребовался *меньший объём памяти*.

Временная и объёмная эффективности довольно сложным образом связаны и с самим алгоритмом и друг с другом.
Приведённое выше определение и рассмотренные ранее примеры трассировок алгоритмов показывают, что важнейшим фактором, влияющим на время выполнения алгоритма, а также на требуемую память является *набор исходных данных*, для которого осуществляется исполнение алгоритма. Такой набор состоит из значений переменных, имена которых указаны в процедурах ввода алгоритма.

Пусть D_A— множество допустимых для данного алгоритма A наборов исходных данных. Конкретный набор исходных данных $d \in D_A$, для которого осуществляется фактическое исполнение алгоритма, принято для краткости называть **входом алгоритма**. Например, входом алгоритма поиска является набор входных данных d= {n → 6 ,x → (4, 9, 2, 9, 0, 3)}.

Фактически затраченное на выполнение алгоритма время, которое определяет его *временную эффективность*, существенно зависит от конкретного исполнителя, например, от модели компьютера, от его возможностей, скоростей выполнения различных действий и т.д.

Чтобы при анализе алгоритма отделить его собственные свойства от неодинаковых свойств различных исполнителей для сравнения между собой алгоритмов используется *не время* как таковое в секундах, минутах или часах, а *общее количество действий*, которые должны быть осуществлены при исполнении алгоритма.

Это даёт возможность не учитывать различную скорость работы исполнителей. Однако остается зависимость от набора действий, доступных исполнителю.

Пусть имеется несколько алгоритмов решения одной и той же задачи и для их выполнения используется один и тот же вход *d*. Тогда более эффективным по времени считается тот алгоритм, при исполнении которого для получения результата требуется выполнить *меньше действий*.

К отдельным действиям алгоритма при таком подсчёте относятся: арифметические операции (+, –, ×, /), операции сравнения (=, ≠, >, < , ≤, ≥), логические операции (∧, ∨, ¬), а также присваивание и некоторые другие действия. А вот действия, связанные с организацией обмена (ввода или вывода данных), как правило, *не учитываются*.

Таким образом, для *улучшения* алгоритма, для повышения его *временной* эффективности следует каким-либо образом *уменьшить* общее количество действий, связанных с его исполнением, но при этом *правильность результата не должна пострадать.* Описанный подход позволяет в основном исключить влияние индивидуальных особенностей исполнителя на сравнение алгоритмов и принимать во внимание только важнейшие особенности самих алгоритмов.

В связи с тем, что временная эффективность алгоритма и количество действий в нём связаны друг с другом *противоположной зависимостью*, пользоваться характеристикой эффективность не совсем удобно. Поэтому на практике для анализа алгоритмов применяют другую характеристику, которая называется *трудоёмкостью* алгоритма.

Трудоёмкостью алгоритма *A* называется характеристика, численно равная количеству действий, которые потребуется выполнить для получения результа во время исполнения алгоритма *A* на некотором его входе $d \in D_A$.

Трудоёмкость противоположна эффективности: чем больше операций приходится выполнять для получения результата, тем выше трудоёмкость алгоритма, и тем менее он эффективен по времени.

Поскольку на различных входах $d \in D_A$ трудоёмкость одного и того же алгоритма *A* различна, естественно ввести в рассмотрение **функцию трудоёмкости**, отражающую эту зависимость.

Функция трудоёмкости $T_A(d)$ представляет собой отображение множества допустимых наборов данных D_A на множество целых положительных чисел N, которое для каждого конкретного входа $d \in D_A$ определяет трудоёмкость алгоритма .

Таким образом, можно утверждать, что вход алгоритма *d* является аргументом, а сам алгоритм *A* определяет *вид* функциональной зависимости для функции трудоёмкости $T_A(d)$.

В качестве характерного примера определения вида функции трудоёмкости, рассмотрим алгоритм накопления суммы. В соответствии с приведёнными выше пояснениями для определения трудоёмкости требуется подсчитать только количество действий из фрагмента алгоритма, в который не входят описания и операции обмена:

```
S=0.0;  i=1; while (i<=n) {  a=1.0/(i*i+1.0); S=+a;  i++; }
```

Этот фрагмент содержит:

1. два действия присваивания на этапе инициализации (S=0.0 и i=1);
2. одну операцию сравнения, входящую в условие повторения i<=n;
3. три арифметические операции и присваивание в операторе a=1.0/(i*i+1.0);
4. одну арифметическую операцию и присваивание в операторе S=S+a;
5. одну арифметическую операцию и присваивание в операторе i=i+1 (i++).

Суммарное количество перечисленных в пунктах 1–5 действий равно одиннадцати.

Но девять действий пунктов 2–5 (определение значения условия повторения и действия, входящие в тело цикла) выполняются на *каждой* итерации цикла, а количество итераций равно *n*. Получаем, что функция трудоёмкости алгоритма равна $T_A(d)=9n+2$.

В рассматриваемом случае любой вход алгоритма состоит из одной переменной n, значение которой определяет количество слагаемых в сумме. Естественно, что она же и оказалась аргументом функции трудоёмкости $T_A(d)=T_A(n)=9n+2$.

Объёмная эффективность

При анализе эффективности алгоритма определение необходимого для его выполнения объёма памяти не менее важно, чем определение затрат времени. Это связано с тем, что в большинстве случаев *уменьшение затрат времени* на исполнения алгоритма приводит к необходимости соответствующим образом *увеличить объём памяти*.

Можно, например, хранить таблицу ответов для некоторой задачи, и почти не затрагивать времени на получение результата. Тогда времени потебуется мало, но зато необходимо много памяти на хранение всех ответов.

Другой вариант возникает если отказаться от хранения ответов, а сохранить только небольшой текст программы, которая по сути дела представляет собой набор правил определения ответа. Но при этом каждый раз необходимо тратить время на получение результата, поэтому времени потребуется больше, а памяти — меньше.

Следовательно, совершенно некорректно говорить об *общей* эффективности алгоритма, учитывая при этом только временные требования.

Для описания эффективности алгоритма по памяти, то есть требований алгоритма к необходимому для его выполнения объёму памяти, целесообразно ввести **функцию объёма памяти**, аналогичную функции трудоёмкости.

Функция объёма памяти $V_A(d)$ представляет собой отображение множества допустимых наборов данных D_A на множество целых положительных чисел N, которое для каждого конкретного входа $d \in D_A$ определяет объём памяти (оперативной и внешней), необходимой для выполнения алгоритма *A*.

Заметим, что при таком подсчёте необходимо учитывать ***все*** без исключения константы и переменные, участвующие в вычислениях.

Для определения вида функции объёма памяти следует использовать *описания* алгоритма, поскольку практически в любом алгоритмическом языке существует требование, в соответствии с которым должным быть описаны все объекты, в том числе все константы и переменные, используемые в алгоритме. В этих описаниях тем или иным образом указывается тип объекта, используемого в алгоритме, а тип в свою очередь определяет необходый для хранения объём памяти.

Так, например, из описаний алгоритма накопления суммы

```
int i,n; float a, S;
```

следует, что в нём используется четыре переменные скалярного типа.

Получается, что в этом алгоритме требуется память для хранения только четырёх чисел — значений используемых переменных. Если считать, что для хранения значения каждой из переменных алгоритма выделяются поля длиной четыре байта, то получим, что общий требуемый объём равен 16 байтам, и, следовательно, функция объёма памяти имеет вид: $V_A(n)=16$.

Характерный размер входа

Из общих соображений понятно, что отдельно взятый набор входных данных некоторого алгоритма может состоять из любого конечного количества значений. При этом не все входящие в набор исходных данных величины оказывают одинаково существенное влияние на трудоёмкость алгоритма и объём требуемой памяти.

Например, для линейного алгоритма

```
main ( )
 {
   int n, m; //Описания основных переменных
   scan (n, m); //Ввод исходных данных
   int k=n; n=m; m=k; //Описание промежуточной переменной k и обмен
   print (n, m) //Вывод результатов
}
```

Функции трудоёмкости объёма памяти являются константами: $T_A(d)=3$ и $V_A(d)=6$.

То есть они *вообще не зависят* от исходных данных.

Отсюда следует, что набор *фактических аргументов* функции трудоемкости и функции объёма памяти не обязательно совпадает с полным набором исходных данных алгоритма.

Обычно как для функции трудоёмкости, так и для функции объёма всё множество входов алгоритма удаётся охарактеризовать *единственным* числовым параметром, который играет определяющую роль в получении их значений.

Возьмем, например, задачу скалярного уножения векторов a и b, состоящих из n компонент.

```
main ( )
{ int n; float a[ ], b[ ];
// Ввод размерности n и элементов векторов
   scan (n, a. b);
     // Инициализация цикла
   float p=0.0; //Начальное значение суммы
   int k=1; //Номер начальной координаты
   while (k<=n)  //Перебор координат векторов
       {
        p=+a[k]*b[k];   //Добавление слагаемого
         k++; //Номер  следующей координаты
        }
   print (n, a, b, p);
}
```

Любой вход *d*= (n, a, b) у этого алгоритма состоит из 2*n*+1-го числа: значения переменной n, которая определяет размерность векторов, а также компонентов этих векторов — элементов массивов a и b.

Найти функцию трудоёмкости и функцию объёма памяти для этого алгоритма.

Его фунция трудоёмкости имеет вид $T_A(d)=T_A(n)=6n+2$, зависит только от одной входной величины n, которая определяет размерность векторов. При этом сами компоненты векторов на значение функции трудоёмкости никакого влияния не оказывают.

Определяя функцию объёма памяти, следует учесть, что если в алгоритме используются данные структурированных типов, например, массивы, то необходимо выделять поле соответствующе длины для хранения *каждого из элементов* структуры.

В рассматриваемой задаче используются два массива, водержащие по n элементов каждый. Таким образом, на хранение элементов этих массивов требуется $8n$ байтов памяти. Вся функция объёма памяти, следовательно, имеет вид:

$$V_A(d)=V_A(n)=8n+16.$$

Вход алгоритма обычно может быть охарактеризован некоторым *числовым* параметром. Чаще всего в качестве такой характеристики используется *целое число*, которое определённым образом связано со всем набором исходных данных алгоритма . Такую характеристику называют **характерным размером входа** и обозначают n . Размер входа является *аргументом* функций эффективности.

Так, в алгоритме вычисления суммы размер входа определяет количество слагаемых, а в алгоритме скалярного умножения векторов — размерность этих векторов. Характеристика входа чаще всего бывает связана с размерностью рассматриваемых в задаче массивов — векторов, матриц и т.д.

Заметим, что вход любого алгорима можно представить в виде двоичного кода *и задавать его длину в битах*. В этом случае характерный размер входа принято называть **длиной входа**. Этот вариант задания входа характерен для теоретических исследований.

Наилучший и наихудший случаи

Не для всех алгоритмов функция трудоёмкости является непрерывной функцией своих аргументов. Так в алгоритме сортировки двух элементов последовательности, основная часть которого представляет собой сокращённое ветвление

```
if (a[1]>a[2]) { c=a[1]; a[1]=a[2]; a[2]=c; }
```

функция трудоёмкости $T_A(d)=1$ в случае выполнения условия a[1]>a[2], а при его невыполнении $T_A(d)=4$.

Конечно, можно записать такую функцию в традиционном для кусочно-непрерывных функций виде:

$$T_A(d) = \begin{cases} 1, & a_1 \le a_2 \\ 4, & a_1 > a_2 \end{cases}$$

Но это более или менее приемлемый способ для двух-трёх ветвей, а при наличии большого их количества работать с функцией в таком виде становится очень неудобно.

Важно понимать, что в подавляющем большинстве случаев точное распределение значений функции по ветвям не представляет никакого практического интереса.

Как правило, для практических целей важно знать наименьшее и наибольшее количество операций, которые придётся выполнить для получения результата. При этом наименьшее значение считается **лучшим случаем** и обозначается $T_A^{\vee}(d)$, а наибольшее считается **худшим случаем** и обозначается $T_A^{\wedge}(d)$.

С использованием введённых обозначений для алгоритма сортировки получим: $T_A^{\vee}(d) = 1$ и $T_A^{\wedge}(d) = 4$.

для содержащего три ветви алгоритма вместо уже достаточно неудобного вида обычной функции трудоёмкости

$$T_A(d) = \begin{cases} 2, & x < 0 \\ 5, & 0 \le x < 1 \\ 7, & x > 1 \end{cases}$$

получается короткая, информативная запись: $T_A^{\vee}(d) = 2$ и $T_A^{\wedge}(d) = 7$, которая имеет очень прозрачный смысл: в лучшем случае при исполнении алгоритма потребуется выполнить две операции, а в худшем — семь

Используем теперь введенные понятия для анализа алгоритма определения экстремального элемента массива, основная часть которого имеет вид:

{ … max=x[1]; i=2; while (i <= n) {if (x[i]>max) max=x[i]; i++;} …}

В целом функция трудоёмкости как будто может быть определена по описанной методике. Но при подсчёте количества действий, выполняемых на одной итерации цикла, возникает осложнение, связанное с тем, что внутри цикла имеется ветвление if (x[i]>max) max=x[i];

В этом ветвлении неравенство x[i]>max удовлетворяется для одних элементов массива и не удовлетворяется для других. Следовательно, сколько именно раз окажется выполненным оператор max=x[i] существенно зависит от исходного массива x.

Получается, что в этом алгоритме невозможно даже определить вид *обычной* функции трудоёмкости. Но зато очень просто определить лучший и худший случаи. А именно: в лучшем случае оператор max=x[i] не выполнится ни разу, а в худшем случае этот оператор будет выполняться каждый раз, для каждого очередного элемента массива.

Лучший случай возникает, когда первый элемент массива оказывается наибольшим. Тогда в результате первого присваивания max=x[1] значение переменной max оказывается равным значению этого первого элемента массива и неравенство x[i]>max не удовлетворяется ни одним из последующих элементов массива.

В этом случае *на каждой* итерации выполняется всего четыре действия i<=n x[i]>max и i++, следовательно, функция трудоёмкости *лучшего* случая имеет вид: $T_A^v(n)=4n+2$.

Худший случай возникает, когда каждый следущий элемент массива оказывается больше предудущего, то есть исходный массив *упорядочен* по возрастанию значений его элементов.

Именно в этой ситуации неравенство x[i]>max удовлетворяется *каждым следующим* элементом массива и, следовательно, на каждой итерации цикла добавляется ещё одно действие max=x[i]. Отсюда функция трудоёмкости худшего случая имеет вид $T_A^{\wedge}(n)=5n+2$.

Можно предположить, что аналогичные ситуации возникникают и для функции объёма памяти и из тех же самых соображений рассматривать наименьший и наибольший объём памяти, который потребуется алгоритму, и обсуждать функции объёма памяти лучшего и худшего случаев**.**

При анализе алгоритмов обычно интересуются самым плохим вариантом — *худшим случаем*, в котором для получения результата требуется выполнить *максимально* возможное количество действий, или выделить *максимально* возможный объём памяти.

Теперь мы можем применить введённые понятия для *сравнения* между собой различных алгоритмов решения одной и той же задачи.

Выберем для этого построенные выше алгоритмы задачи поиска и подсчитаем трудоёмкости и объёмы памяти для каждого из фрагментов, которые представляют основу алгоритмов линейного поиска. Классчический поиск:

```
flag:=false;  k:=0; i:=1;
  while (i<=n) and not Flag do
   begin
      if x[i]=a  then  begin flag:=true; k:=i  end  else i :=i+1
   end,
```

быстрый поиск: `i:=1; while (i<=n) and (x[i]<>a) do i:=i+1`

поиск с барьером `i:=1; x[n+1]:=a; while x[i]<>a do i:=i+1`

В этих фрагментах *невозможно* определить *точное количество итераций цикла*. Поиск *зависит* как от заданного массива, так и от искомого элемента, и он может быть закончен на любом из элементов массива, в том числе и на самом первом, и на самом последнем.

Расчёты лучших и худших случаев провсети самостоятельно.

	Классический	**Быстрый**	**Алгоритм с барьером**
Трудоемкость в лучше случае	12	9	4
Трудоемкость в в худшем случае	$6n+3$	$5n+1$	$3n+4$
Функция объёма памяти	$4n+20$	$4n+12$	$4n+16$

Во всех рассмотренных вариантах построения алгоритма поиска максимальное количество операций, которое необходимо выполнить для получения результата, *линейно* зависит от количества элементов в рассматриваемом массиве. Поэтому все эти алгоритмы называются алгоритмами **линейного поиска**. Говорят также, что эти алгоритмы имеют **линейную вычислительную сложность**.

Опираясь на проведённый для алгоритмов поиска анализ, мы можем уточнить использованные без точного определения понятия лучшего и худшего случаев функции трудоёмкости и функции объёма памяти.

Чтобы сформулировать такое определение найдём, например, трудоёмкость худшего случая для классического алгоритма, если на его вход подается массив с рамерностью, скажем, n=10. По второй строке таблицы находим, что трудоёмкость классического алгоритма в этом случае равна 63.

Теперь найдем трудоёмкость худшего случая алгоритма с барьером для массива из 20 элементов (то есть для n=20). По этой же таблице получаем, что трудоёмкость в этом случае равна 84. Другими словами, получается, что *не для любых входов* алгоритм с барьером лучше, чем классический.

Легко, конечно, заметить, что этот пример не совсем корректен, поскольку для сравниваемых алгоритмов выбраны *неодинаковые* входы. Однако же для определения лучшего и худшего случаев также выбираются неодинаковые входы.

Отсюда следует, что для того чтобы результат сравнения оказался корректным у всех неодинаковых входов , которые используются для выбора лучшего и худшего случаев функции трудоёмкости алгоритма , должно быть *нечто общее*, чтобы разница, например, в количестве элементов в массиве не повлияла на значение функции.

Это же пример даёт требуемый ответ: корректно выбирать лучший и худший случай только на всех входах *с одинаковым характерным размером*. Для примеров с массивами это означает, что при выборе лучшего и худшего случаев в качестве исходных могут выбираться любые массивы, но с одной и той же размерностью.

Эффективность алгорима бинарного поиска

На первом шаге количество *k* рассматриваемых элементов массива равно *n*

После первого шага количество *k* рассматриваемых элементов массива уменьшается в два раза *k*=*n*/2.

После второго шага — в четыре раза $k=n/2^2$.

Вообще, после шага с некоторым номером *m*: $k=n/2^m$.

В худшем случае процесс поиска закончится после того, как останется один элемент: $n/2^m=1$. Отсюда: $m=\log_2 n$.

Например, при *n*=1024 для линейного поиска в худшем случае потребуется 1023 шага, а в среднем 512 шагов. А для бинарного поиска в худшем случае потребуется 10 шагов.

Трудоёмкость (сложность) алгоритма линейного поиска пропорциональна n, а сложность алгоритма бинарного поиска пропорциональна $\log_2 n$.

Практически важный аспект анализа трудоёмкости связан с классификацией алгоритмов по типу зависимости $T_A(d)$ по скорости ее роста.

При неорганиченном увеличения длины входа n алгоритма, функцию его трудоёмкости принято называть **сложностью** алгоритма.

Существуют алгоритмы, в которых $T_A(d)$ растет линейно, квадратично, имеет кубическую зависимость и т.д. Такие алгоритмы принято называть ***полиномиальными*** или алгоритмами с полиномиальной сложность и говорить, что алгоритм относится к ***классу сложности P.***

Существую алгоритмы, временная сложность которых растет быстрее полинома любой степени, как a^n с некоторым основанием *a*. Такие алгоритмы принято называть алгоритмами с ***экспоненциальной*** сложностью и говорить, что алгоритм относится к ***классу сложности EXP***.

Зависимость от параметра	**Размер задачи *d* =*n***					
	10	**20**	**30**	**40**	**50**	**60**
n	0,00001 сек.	0,00002	0,00003	0,00004	0,00005	0,00006
n^2	0,0001	0,0004	0,0009	0,0016	0,0025	0,0036
n^3	0,001	0,008	0,027	0,064	0,125	0,216
n^5	0,1	3,2	24,3	102	5,2 мин	13 мин
2^n	0,001	1,0	18 мин	12,7 дн.	35,7лет	36600 лет
3^n	0,05	58 мин	6,5 лет	385500 лет	$2*10^{10}$	$1,3*10^{15}$

В качестве *образцов* скорости роста функций трудоёмкостей обычно выбирают следующий набор функций:

$\log_a n,\ a>1$

n

$n \log_a n,\ a>1$

$n^k,\ k \geq 1$

$a^n,\ a>1$

$n!$

n^n

Важно подчеркнуть, что для решения одной и той же задачи можно построить алгоритмы с разной трудоёмкостью.

Пример: трудоёмкость алгоритма вычисления значения полинома в точке.

Алгоритм, использующий прямое вычисление $p(x)=a_n x^n+a_{n-1}x^{n-1}+\ldots+a_1 x^1+a_0$.

Для вычисления $a_n x^n$ требуется n умножений.

Для вычисления всех слагаемых нужно $1+2+3+\ldots+n = n(n+1)/2$ умножений.

Для вычисления окончательного значения еще n сложений.

Обще количество операции $n(n+1)/2+n$ – имеет место квадратичная функция трудоёмкости.

Алгоритм, использующий схему Горнера, например, для полинома 4 степени:

$p(x)=(((a_4 x+a_3)x+a_2)x+a_1)+a_0$ требует 4 сложения и 4 умножения.

В общем случае по методу индукции получим n умножений и n сложений, или всего $2n$ операций.

Таким образом имеется линейная трудоёмкость алгоритма.
Но класс сложности также полиномиальный.

Кратные (вложенные) циклы

Структура алгоритма, в котором внутри цикла находится другой цикл, называется ***кратным*** или ***вложенным*** циклом. Глубина вложения циклов может быть любой.

Пример: задача сложения матриц.

Пусть имеются две матрицы $A^{n\times m}$ и $B^{n\times m}$. Требуется найти их сумму.

Во-первых, из определения суммы матриц следует, что суммировать можно только матрицы одинаковой размерности, то есть матрицы, состоящие из одинакового количества строк *n* и столбцов *m*. Следовательно, исходными данными задачи являются *n* — общее для *A* и *B* количество строк, и *m* — общее количество столбцов. Элементы матриц *A* и *B* могут быть целого или вещественного типов. Для определенности будем считать, что эти элементы относятся к вещественному типу.

Во-вторых, из того же определения следует, что сумма матриц *A* и *B* есть матрица *C* той же самой размерности $C^{n\times m}$, элементы которой c_{ij} определяются как сумма соответствующих элементов матриц слагаемых: a_{ij} и b_{ij}

$$c_{ij} = a_{ij} + b_{ij}, i = 1,2,...,n; j = 1,2,...,m$$

$$\begin{pmatrix} a_{11}, a_{12}, ..., a_{1m} \\ a_{21}, a_{22}, ..., a_{2m} \\ ... \\ a_{n1}, a_{n2}, ..., a_{nm} \end{pmatrix} + \begin{pmatrix} b_{11}, b_{12}, ..., b_{1m} \\ b_{21}, b_{22}, ..., b_{2m} \\ ... \\ b_{n1}, b_{n2}, ..., b_{nm} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11}, a_{12}+b_{12}, ..., a_{1m}+b_{1m} \\ a_{21}+b_{21}, a_{22}+b_{21}, ..., a_{2m}+b_{2m} \\ ... \\ a_{n1}+b_{n1}, a_{n2}+b_{n2}, ..., a_{nm}+b_{nm} \end{pmatrix}$$

Можно рассуждать следующим образом. Чтобы получить все элементы матрицы ***C*** следует перебрать все строки исходных матриц *A* и *B* и сложить все соответствующие их элементы. Пусть переменная *i* имеет смысл номера текущей строки матриц. Тогда для перебора всех строк матриц нужно заставить *i* изменяться от *1* до *n*. Получаем следующий набросок алгоритма:

```
i:=1;{перебор начнем с первой строки матриц}
while i ≤ n do {номер  текущей строки меньше номера последней, то есть
имеются еще не рассмотренные  строки}
    begin
        {Действия с элементами текущей i строки}
        i:=i+1{переход к следующей строке]
    end;
```

Теперь следует уточнить действия с каждой текущей строкой, то есть со строкой, имеющей номер *i*.

Из определения следует, что нужно перебрать все её элементы и осуществить сложение соответствующих элементов матриц слагаемых.

Пусть *j* — номер текущего элемента *i* строки матрицы, то есть *j* — это номер столбца, в котором находится элемент c_{ij}.

Перебор столбцов можно осуществить стандартным образом: начать перебор с первого столбца *(j:=1)*и перебирать столбцы, пока номер текущего столбца не превосходит номер последнего столбца $(j \le m)$. То есть перебирать следует пока имеются еще не рассмотренные элементы *i* -й строки матрицы.

Во время перебора следует получить каждый текущий элемент c_{ij} матрицы *C* как сумму соответствующих элементов матриц слагаемых a_{ij} и b_{ij} , а затем перейти к следующему столбцу:

```
{Действия с элементами i строки}
j:=1;{перебор столбцов i-й строки, начиная с первого}
while j ≤ m do {есть еще не просмотренные столбцы i-й строки}
   begin
      c[i,j]:=a[i,j]+b[i,j]; {получение текущего элемента матрицы C}
      j:=j+1 {переход к следующему столбцу i-й строки}
   end;
```

Осталось вставить уточняющий фрагмент в основной цикл.

```
    A,B,C: array of real; n, m, i, j : integer;
begin
    read (n. m, A, B); {Ввод количество строк, столбцов и всех элементов матриц
    слагаемых}
    i:=1;{начнем с получения элементов первой строки матрицы С}
    while i ≤ n do {есть имеются еще не рассмотренные  строки}
        begin
            {Действия с элементами текущей i строки}
            j:=1;{перебор столбцов i-й строки начиная с первого}
            while j ≤ m do {есть еще не просмотренные столбцы}
                begin
                    c[i,j]:=a[i,j]+b[i,j]; {получение текущего элемента матрицы С}
                    j:=j+1 {переход к следующему столбцу i-й строки}
                end;
            i:=i+1{переход к следующей строке]
        end;
    write (C)
end.
```

Оцените функцию трудоёмкости и объёма памяти для полученного алгоритма.

$$T_A(d)=5mn+4n+1 \sim 5N^2+4N+1, \text{ где } N=\max(n,m)$$

$$V_A(d)=12mn+4 \sim 12N^2+16, \text{ где } N=\max(n,m)$$

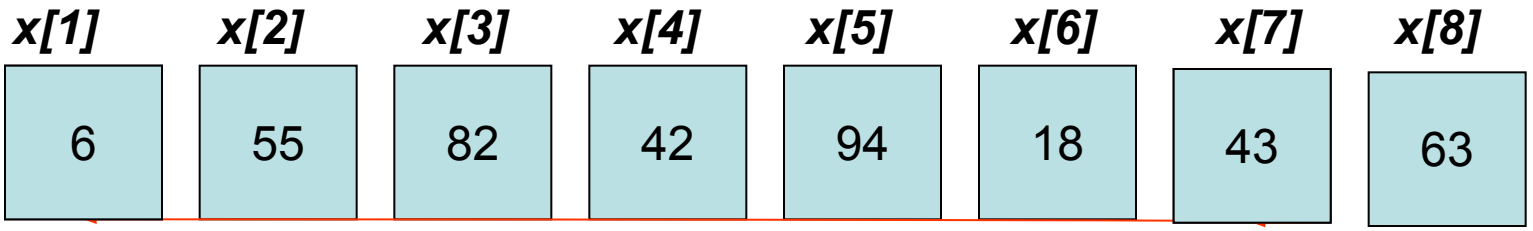
Задача сортировки элементов массива

Сортировкой в информатике называется переупорядочение рассматриваемых объектов по некоторому признаку или системе признаков. Например, упорядочение слов по алфавиту называется ***лексикографической*** сортировкой.

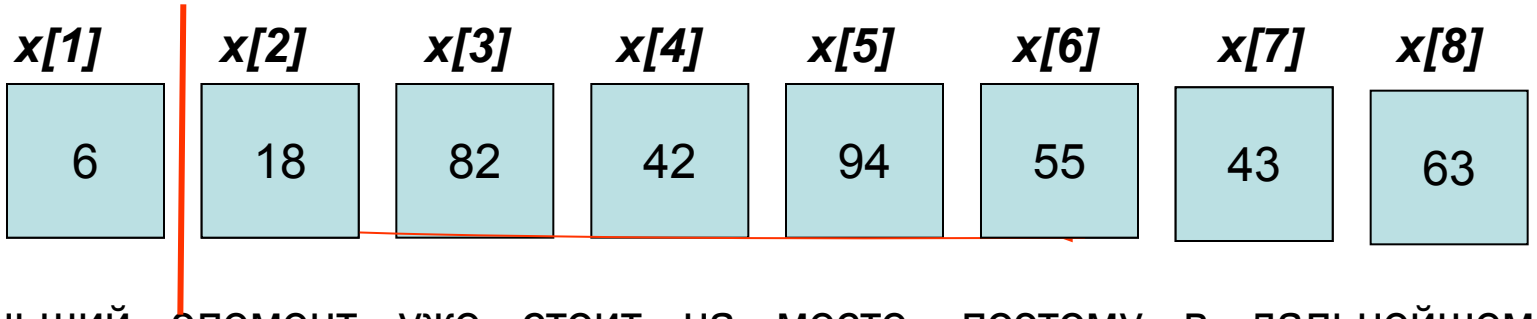
Существует три базовых способа сортировки массивов: прямым выбором, прямыми вставками и обменная, а также больше количество их модификаций.

Рассмотрим способ сортировки массива ***прямым выбором***. Идея алгоритма состоит в том, чтобы на каждом шаге переупорядочения выбирать наименьший элемент в массиве и помещать его в начальную позицию с тем, чтобы на следующем шаге его уже не рассматривать.

Более подробно. На первом шаге ищется минимальный элемент во всем рассматриваемом массиве.



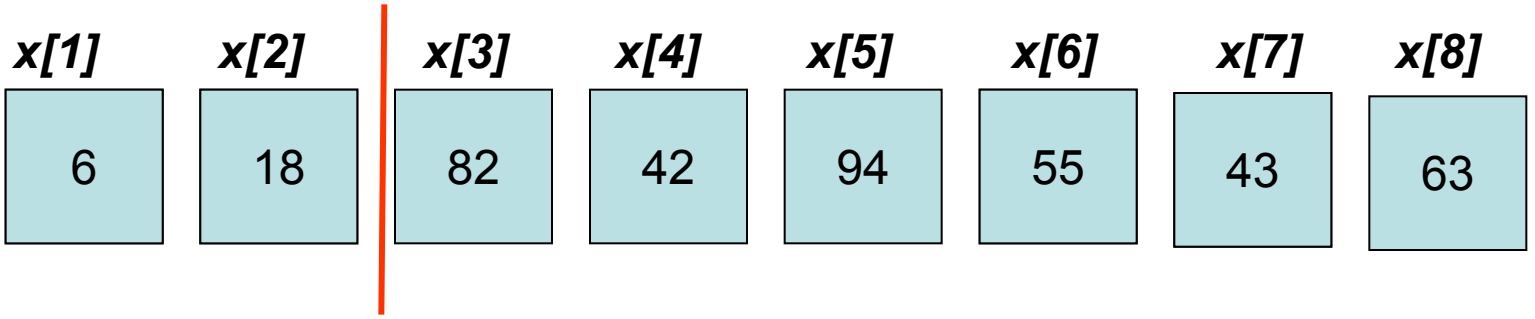
В данном случае это ***x[7]=6.*** Чтобы массив был упорядоченным этот элемент должен стоять на первом месте. Поэтому, совершим обмен значениями между найденным и начальным элементом массива.

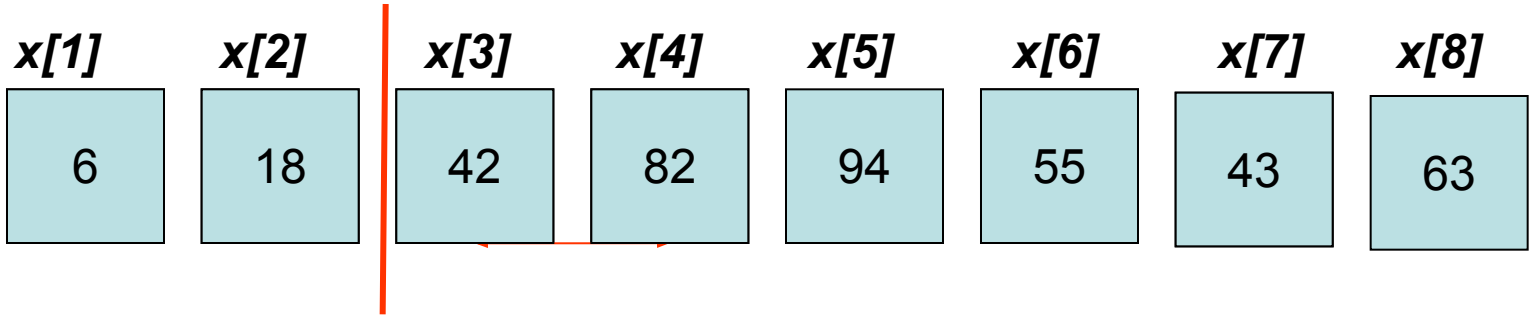


Наименьший элемент уже стоит на месте, поэтому в дальнейшем можно рассматривать уже не весь массив, а только его часть, начинающуюся со второго элемента.

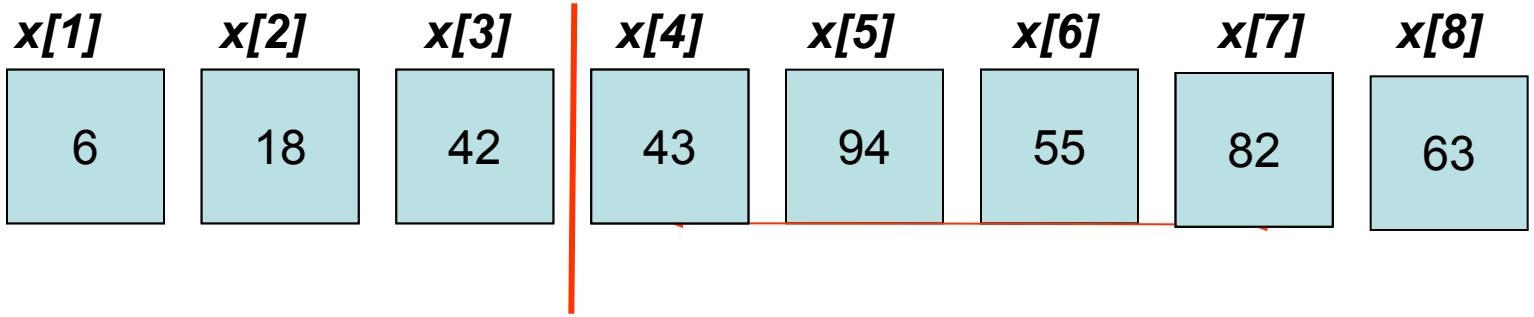
Итак, на <u>втором</u> шаге ищется минимальный элемент в части массива, начинающейся со <u>второго</u> элемента. В данном примере это ***x[6]=18***. И менять шестой элемент нужно с начальным элементом рассматриваемого участка массива, то есть со вторым элементом массива.

Теперь уже два элемента стоят на своих местах.

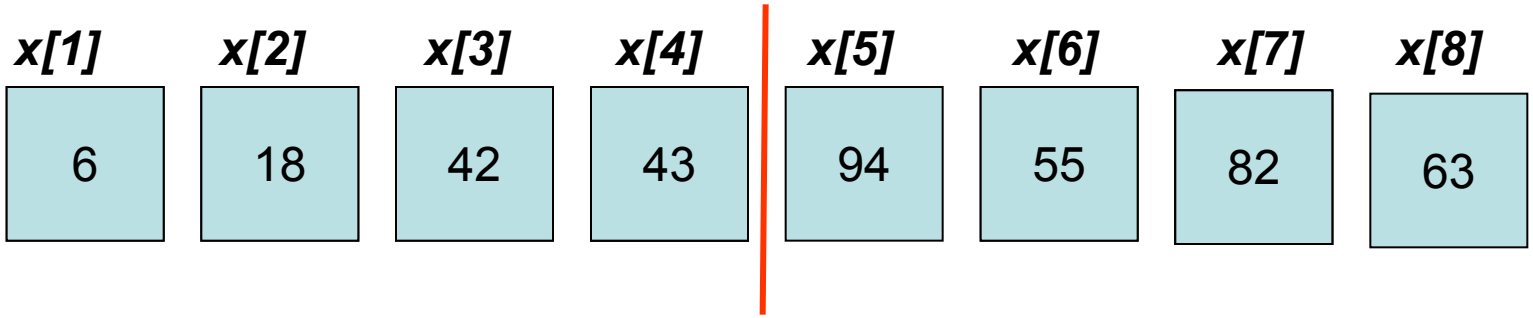


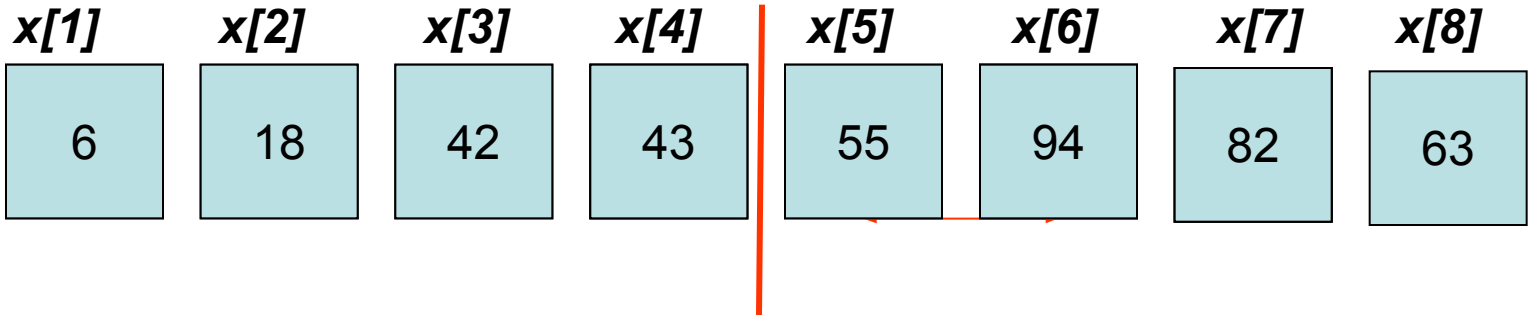


Третий шаг. Минимальный элемент ***x[4]=42***.

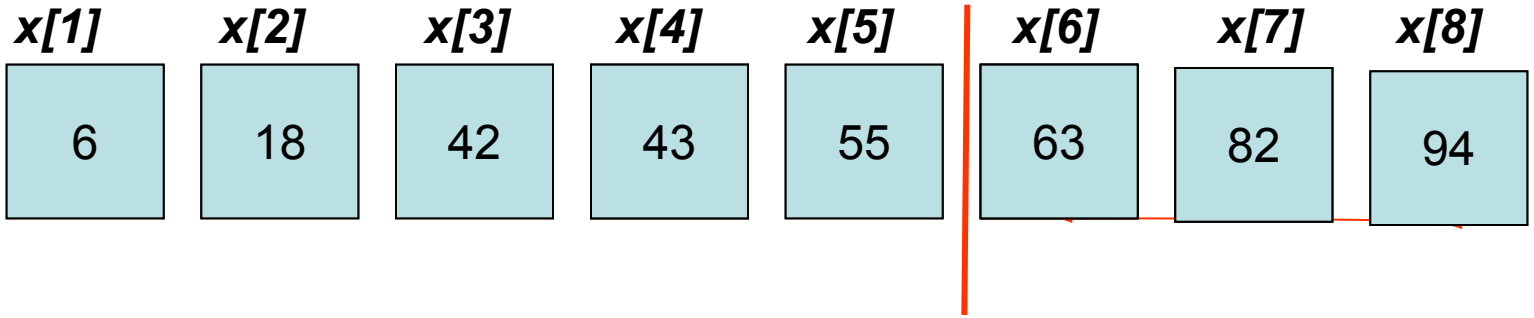


Четвертый шаг. Минимальный элемент ***x[7]=43***

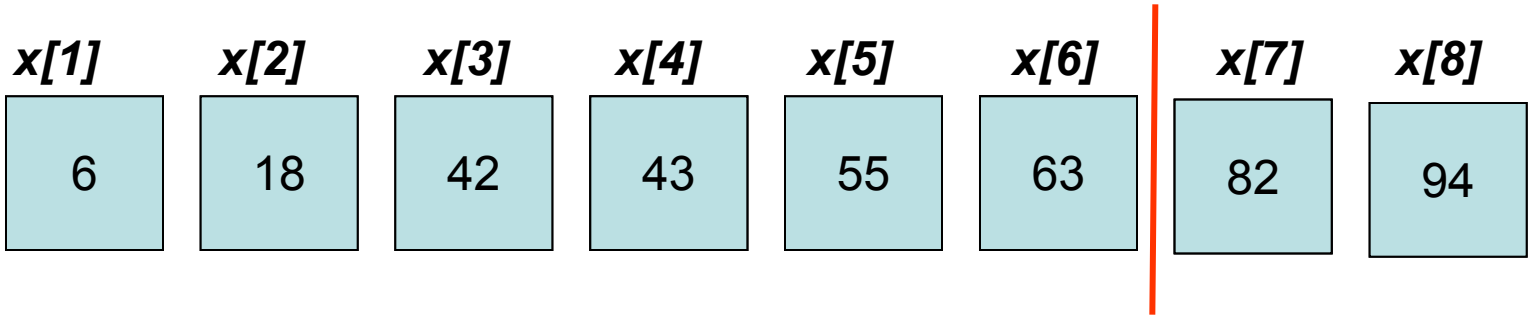


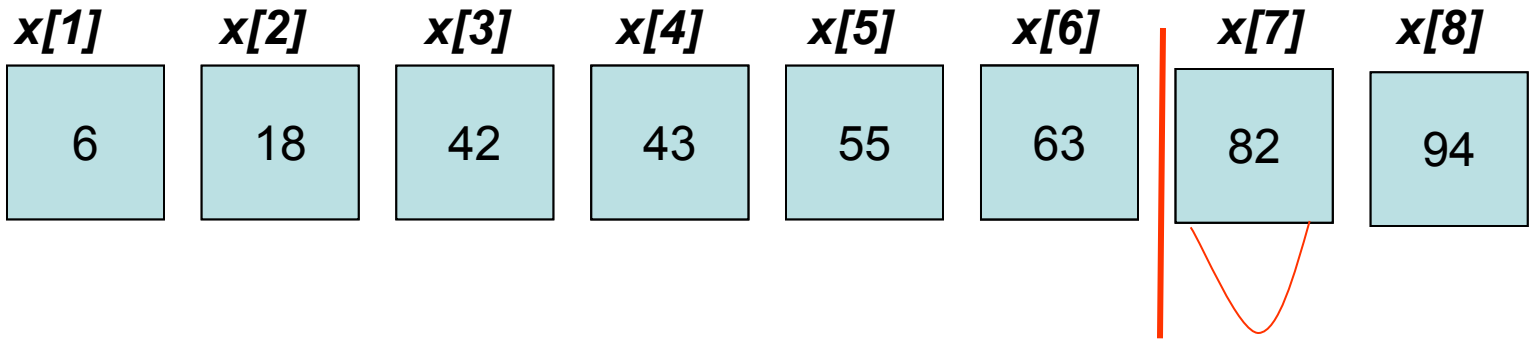


Пятый шаг. Минимальный элемент ***x[6]=55***

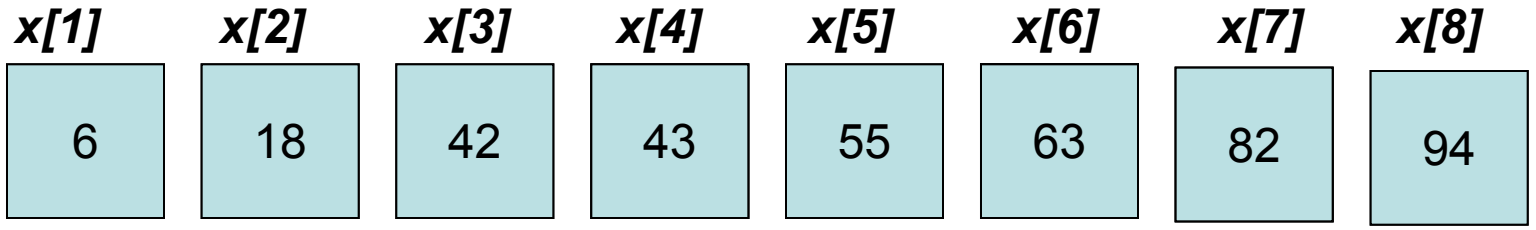


Шестой шаг. Минимальный элемент ***x[8]=63***





Седьмой шаг. Минимальный элемент ***x[7]=82***



Массив упорядочен.

Первый набросок алгоритма

Пусть массив содержит ***N*** элементов и ***i*** — номер шага переупорядочения. Из предыдущих рассуждений вытекает, что нужно проделать ***N-1*** шаг и на каждом шаге с номером ***i*** найти минимальный на участке массива от ***i*** элемента массива до ***N***-го, а затем поменять его местами с начальным ***i-***элементом.

```
begin
  {ввод (формирование) исходных данных}
  i:=1;  {Начальный номер шага}
  while i<=N-1 do   {Не все шаги сделаны. Номер текущего шага не
  превосходит номер предпоследнего шага}
      begin
        {найти номер k наименьшего элемента на участке от i до N}
        {выполнить обмен значениями между i и k элементами}
        i:=i+1;
      end;
  {вывод (использование) результатов}
end.
```

Уточнение алгоритма

```
begin
  {ввод (формирование) исходных данных}
  i:=1;  {Начальный номер шага}
while i<=N-1 do   {Не все шаги сделаны. Номер текущего шага не превосходит
номер предпоследнего     шага}
      begin
        {найти номер k наименьшего элемента на участке от i до N}
        k:=i; {номер кандидата — первый элемент участка}
        j:=i+1; {начинаем со следующего элемента участка}
        while j<=N do {Есть непросмотренные элементы на участке}
            begin if x[j]< x[k] then   {замена кандидата} k:=j; j:=j+1 end;

        {выполнение обмена значениями между i и k элементами}
        a:=x[k]; x[k]:=x[i]; x[i]:=a

        i:=i+1;
      end;
  {вывод (использование) результатов}
end.
```

Оценка функции трудоёмкости алгоритма сортировки прямым выбором

Подсчёт трудоёмкости удобнее начать с определения количества действий, связанных с вложенным циклом. Он содержит два сравнения j<=n и x[j]<x[k], а также два действия в операторе j:=j+1, которые выполняются в любом случае. При определении трудоёмкости следует рассматривать худший случай, поэтому будем считать, что условие x[i]<x[k] удовлетворяется при каждой проверке, и замена кандидата k:=j также происходит на каждой итерации. Следовательно, внутренний цикл содержит пять действий.

Теперь необходимо подсчитать количество итераций вложенного цикла. Его тело выполняется для каждого j от i+1 до n, где i — номер шага. Это означает, что количество итераций внутреннего цикла зависит от номера шага. А именно:

- на первом шаге, когда i=1 параметр j пробегает все значения от 2 до n выполняется n-1 итерация:
- на втором шаге i=2, поэтому параметр j пробегает все значения от 3 до n, следовательно, выполняется n-2 итерации;
- вообще, на шаге с любым номером i выполняется n-i итерации;
- на последнем шаге с номером n-1 выполняется только одна итерация

Эта закономерность дает возможность подсчитать *общее* количество итераций вложенного цикла *на всех шагах* сортировки. Такой подсчет удобнее начинать с последнего шага: M=1+2+3+…+(n-2)+(n-1).

Легко увидеть, что это выражение представляет собой сумму членов *арифметической* прогрессии, у которой первый и последний члены равны соответственно a_1 =1 и a_l =n-1, разность d равна 1 и количество членов равно n-1.

По известной формуле эта сумма в данном случае M=(1+(n-1))×(n-1)/2=n×(n-1)/2.

Таким образом, общее количество связанных с вложенным циклом действий равно M=5×n×(n-1)/2.

К этому результату необходимо добавить количество действий, связанных с внешним циклом. К ним относится два действия в условии повторения цикла i<=n-1, три действия участка инициализации вложенного цикла k:=i; j:=i+1, три действия, связанные с обменом a:=x[i]; x[k]:=x[i]; x[i]:=a, и два действия оператора i:=i+1. Итого 10 действий, каждое из которых выполняется на каждом шаге, то есть n-1 раз.

Кроме того имеется еще одно действие инициализации внешнего цикла i:=1, поэтому $T_A^{\wedge}(n)$=M+10(n-1)+1=5n(n-1)/2+10(n-1)+1=15n^2+15n/2-9.

Таким образом, трудоёмкость худшего случая сортировки массива прямым выбором описывается *квадратичной* функцией количества элементов в массиве.

Описания этого алгоритма дают нам линейую функцию объёма памяти: $V_A(n)$=4n+20, что совершенно естественно, так как в алгоритме используется только один вектор — одномерный массив.

Подпрограммы

Описанием подпрограммы называется поименованная часть алгоритма (программы), которая может быть использована для выполнения описанных в ней действий неоднократно, как в рамках одного и того же алгоритма (программы), так и в разных алгоритмах (программах).

Описание подпрограммы может содержать список входных и выходных величин, который принято называть списком ***формальных параметров***.

Пример

Предположим, что в рассматриваемом алгоритме в разных его местах приходится вычислять значение факториалов различных целых чисел.

■ ■ ■

```
{Нужно вычислить 8!}
P8:=1; i:=2: while i≤8 do begin P8:=P8*i;i:=i+1 end
```

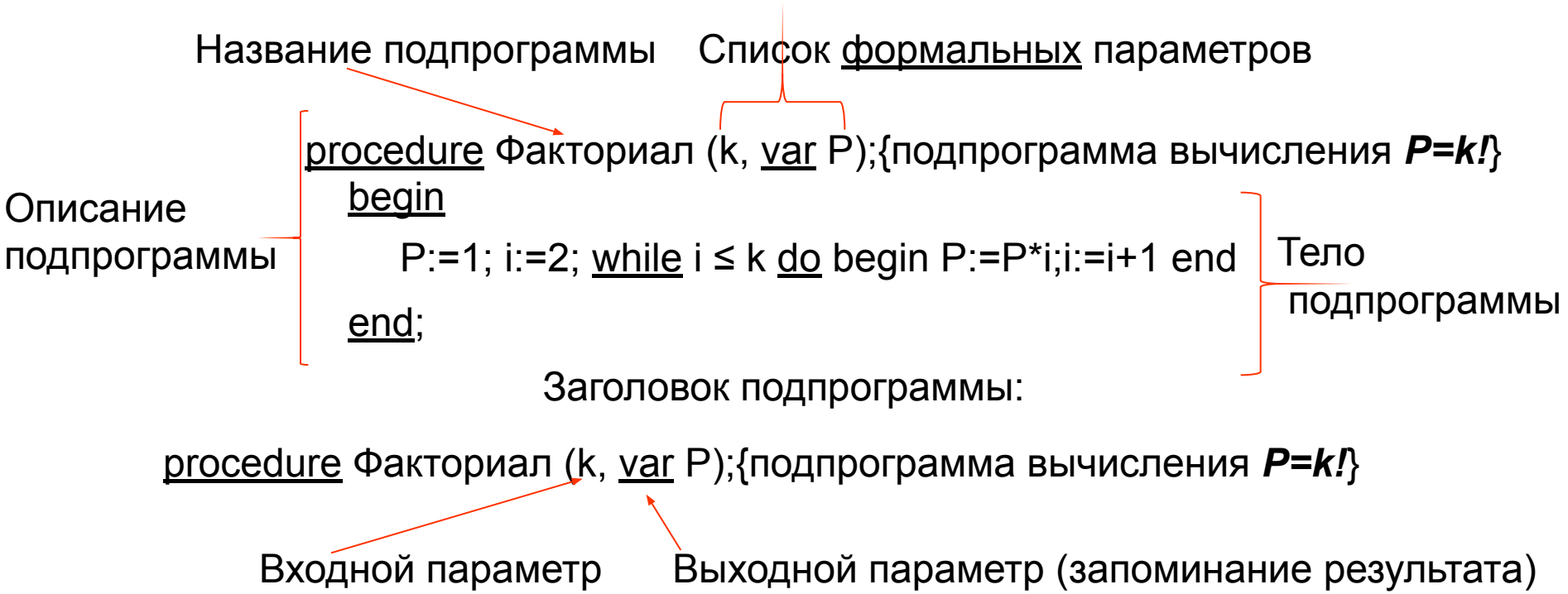
■ ■ ■

```
{Нужно вычислить 6!}
P6:=1; i:=2: while i≤6 do begin P6:=P6*i;i:=i+1 end;
```

■ ■ ■

```
{Нужно вычислить 10!}
P10:=1; i:=2: while i≤10 do begin P10:=P10*i;i:=i+1 end
```

Целесообразно задать — описать такую последовательности действий только один раз — описать подпрограмму. Назовем входную величину, факториал которой ищется именем ***k,*** результат обозначим именем ***P***, а саму подпрограмму назовем ***Факториал***.



Параметры называются формальными потому, что в описании подпрограммы они никаких конкретных значений не имеют. Они служат для того, чтобы задать, записать последовательность действий. Конкретные имена параметров не имеют никакого значения.

Теперь, используя обращение к подпрограмме тот же фрагмент алгоритма можно записать гораздо проще:

{Нужно вычислить 8! и приписать значение переменной P8}
Факториал(8,P8);

■ ■ ■

{Нужно вычислить 6! и приписать значение переменной P6}
Факториал(6,P6);

■ ■ ■

{Нужно вычислить 10! и приписать значение переменной P10}
Факториал(10,P10);

Вызов подпрограммы (обращение к подпрограмме):

Факториал(10,P10);

Название подпрограммы Список фактических параметров

procedure Факториал (k, var P);
begin
P:=1; i:=2; while i ≤ k do begin P:=P*i;i:=i+1 end
end;

Факториал(10,P10);

В математике для решения подавляющего большинства задач используются методы, которые в конечном счете могут быть сведены к одному из двух базовых способов: ***итерации*** или ***рекурсии***.

Итерация означает неоднократное повторение одних и тех же действий, которое после некоторого количества шагов приводит к желаемому результату. Характерным примером итерационного способа решения задачи являются методы последовательных приближений решения нелинейных уравнений, в том числе метод касательных, метод хорд и т.д.

$$f(x)=0; \quad x=\varphi(x); \quad x_0=a; \quad x_n=\varphi(x_{n-1}), n=1,2,... \quad |x_k-x_{k-1}|<\varepsilon$$

С точки зрения структуры алгоритма итерация представляет собой циклический алгоритм

Рекурсия представляет собой ссылку при описании объекта, действия на описываемый объект, действие. Рекурсия означает решение задачи с помощью сведения решения к самому себе. Полностью аналогичные механизмы используются в базовой теории рекурсивных функций, в методе математической индукции, а также в рекуррентных последовательностях.

Выше рассматривался итерационный способ вычисления факториала, основанный на многократном домножении величины, в которой накапливается результат, на очередной сомножитель: $P_i := P_{i-1} \times i,\ i=2,3,\ldots,k.$

```
procedure Факториал (k, var P);{подпрограмма вычисления P=k!}
begin  P:=1; i:=2; while i ≤ k do begin P:=P*i;i:=i+1 end  end;
```

Эту задачу можно решить и с помощью рекурсии, базируясь на следующих соображениях:

$$k! = 1 \times 2 \times 3 \times \ldots \times k = 1 \times 2 \times 3 \times \ldots \times (k-1) \times k = (k-1)! \times k$$

Следовательно, ***P(k)=k!*** можно определить таким образом:

$$P(k) = \begin{cases} 1, k = 1; \\ P(k-1) \times k, k > 1 \end{cases}$$

```
procedure Факториал1 (k, var P);{рекурсивная подпрограмма вычисления P=k!}
  begin
    if k=1 then P:=1 else begin Факториал1(k-1,P1); P:=P1*k end
  end.
```

Основные структуры данных

В информатике используется большое количество различных структур данных, которые применяются для моделирования объектов, встречающихся в рассматриваемых задачах.

Если структура данного по ходу выполнения алгоритма не изменяется, то такая структура считается ***статической***, в противном случае структуру относят к ***динамическим***. Статические структуры данных существуют в неизменном виде в течение всего времени выполнения алгоритма. Динамические структуры создаются, изменяются и уничтожаются по мере необходимости в любой момент исполнения алгоритма.

К данным со статической структурой относятся:

- скалярные типы:
  - целый;
  - вещественный;
  - логический (булевский);
  - символьный;
- структурированные типы:
  - массив;
  - запись;
  - файл (последовательность);
  - множество;
- всевозможные комбинации скалярных и структурированных типов.

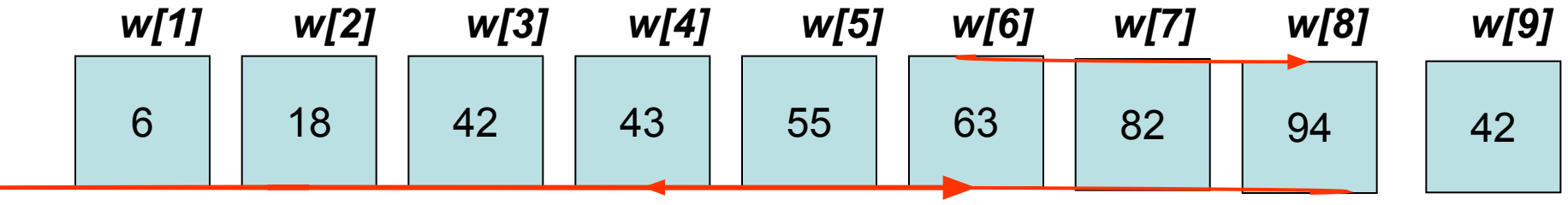
Значение скалярного типа представлено ровно одним компонентом (время, температура). Значение структурированного типа представлено более чем одним компонентом (вектор).

Структурированные типы характеризуются: количеством и возможным типом компонентов значения, а также способом доступа к отдельному компоненту значения.

Массив

$$\bar{x} = \{x_1, x_2, x_3\}; \bar{w} = (w_1, w_2, w_3, w_4, w_5, w_6, w_7, w_8, w_9); A = \begin{bmatrix} a_{11} a_{12} a_{13} \\ a_{21} a_{22} a_{23} \\ a_{31} a_{32} a_{33} \end{bmatrix}$$

Структуры аналогичные векторам и матрицам в информатике принято называть ***массивами***. Все элементы массива должны быть одного и того же типа. Для доступа (обращения) к отдельному элементу массива используется индекс или несколько индексов (**w[5]; w[i+2]; A[1,2]**). Индексы могут быть выражениями, значения которых могут произвольным образом изменяться в заранее заданных границах. Поэтому говорят, что к элементам массивов имеется ***прямой доступ***.



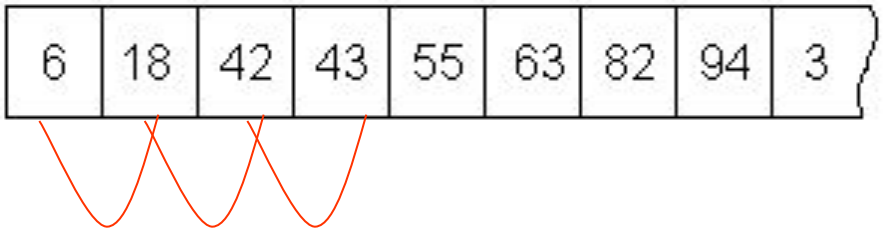
Запись

	День	Месяц	Год
День Победы:	9	май	1945
Полёт Гагарина:	12	апрель	1961

Структуры, аналогичные строкам таблицы, называют ***записями***. Компоненты записей принято называть ***полями***. Различные поля (столбцы таблицы) могут быть разных типов. Для доступа к отдельным полям записи используются их фиксированные и неизменные имена. Например: ***День Победы. Месяц := май***. Имена полей могут выбираться для обработки в произвольном порядке, поэтому говорят, что доступ к полям записи прямой.

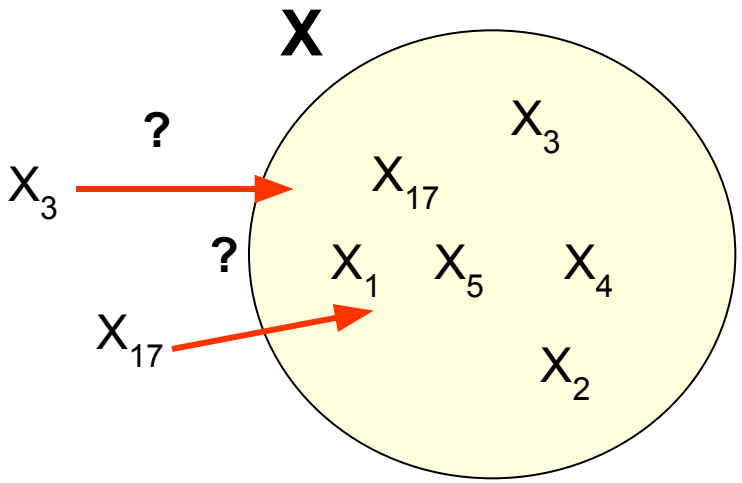
Файл (последовательность)

Основной структурой данных, которая используется для хранения информации на внешних устройствах (магнитных дисках, лентах и т.д.) являются ***файлы*** или ***последовательности***. Считается, что файл всегда находится на внешнем устройстве. При этом количество компонентов файла неизвестно, все компоненты должны быть одного и того же типа. Доступ к компонентам – ***последовательный***.



Множество

Во многих математических и информационных задачах возникает необходимость в прямом или косвенном использовании основного математического объекта множества. Соответствующая множеству тип данных по определению относится к структурированным, так как в общем случае множество может состоять более чем из одного элемента, и при этом со всеми элементами множества приходится выполнять операции как с единым целым. Количество элементов в множестве заранее не определяется, и с течением времени оно может изменятся. Все элементы множества должны быть одного и того же типа. Доступа к отдельным элементам множества нет. Можно только узнать принадлежит элемент множеству или нет. Можно также включить элемент в множество или исключить его из множества. Предусмотрены также стандартные операции над множествами: объединение, пересечение, вычитание и т.д.



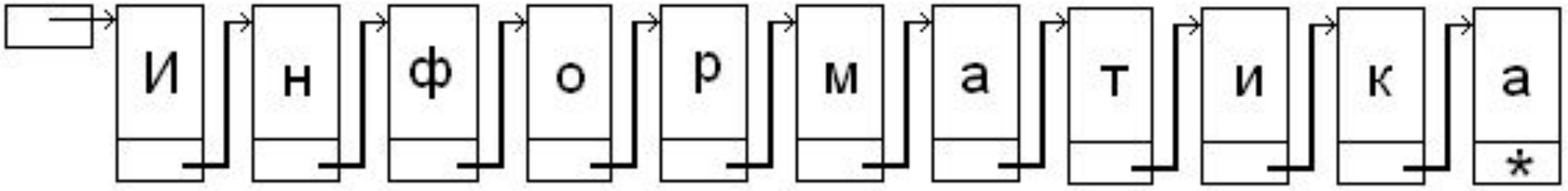
Динамические структуры данных

У данных с динамической структурой с течением времени изменяется сама структура, а не только количество элементов, как у файлов или последовательностей. Базовыми динамическими структурами данных являются:

- линейный список;
- дерево;
- граф.

Линейный список

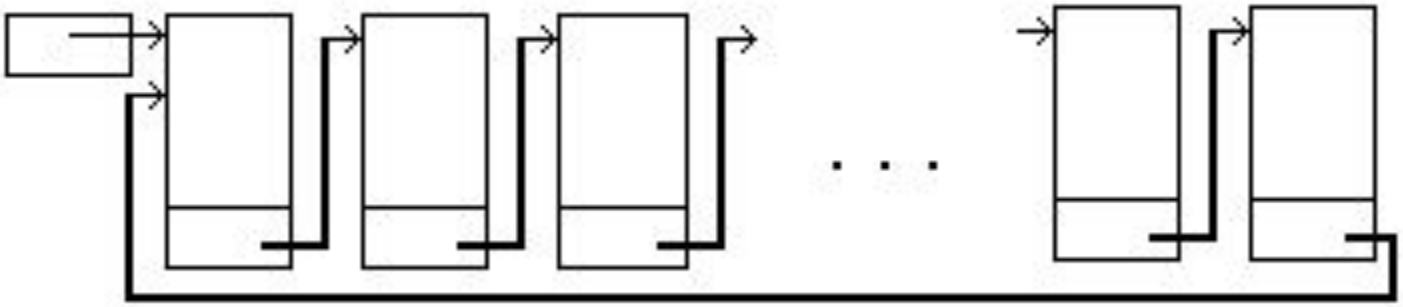
У линейного списка каждый элемент связан с предшествующим ему. У линейного списка известно, какой элемент находится в начале списка, какой в конце, а также, какой элемент стоит перед текущим. В линейном списке переходить от текущего элемента к следующему можно только с помощью указанных связей между соседними элементами.



В целом получается цепочка элементов, в которой можно осуществлять поиск, куда можно вставлять элементы или откуда исключать их.

На базе линейного списка организуются много других типов динамических структур. Это в частности: ***кольца***, ***очереди***, ***деки*** и ***стеки***.

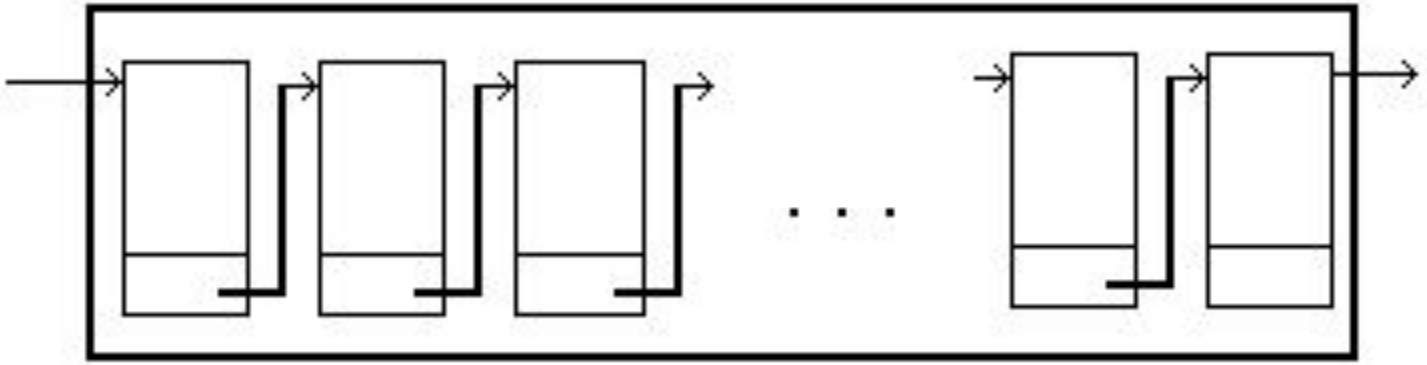
Структура кольца



Отличие кольца от линейного списка в том, что у кольца имеется связь между последним элементов списка и его первым элементом.

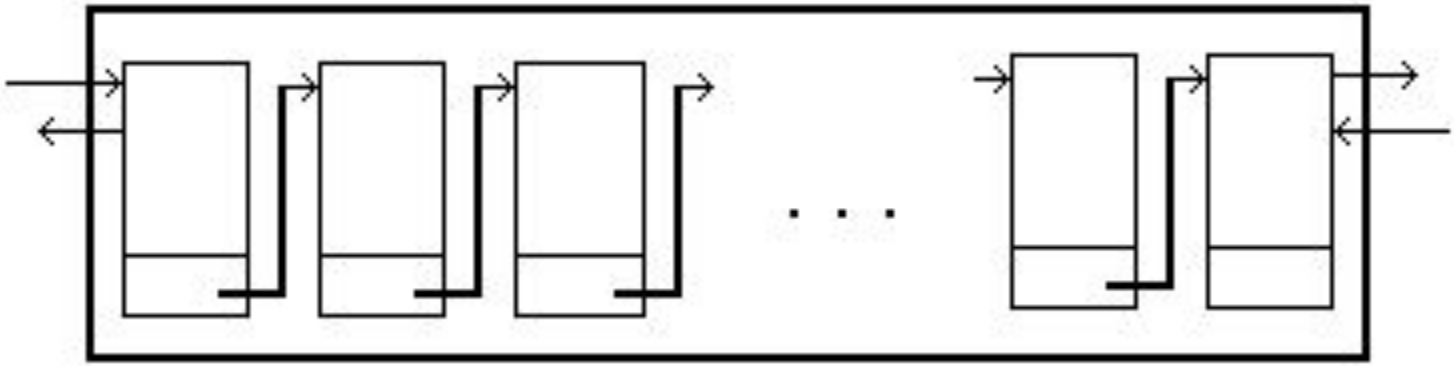
У линейного списка и у кольца возможен доступ к любому элементу структуры. Для этого нужно последовательно перемещаться от одного элемента к другому. Во многих реальных ситуациях такой доступ отсутствует. Можно взаимодействовать только с первым и последним элементами или же только с одним из них. Для моделирования таких объектов используются очереди, деки и стеки.

Структура очереди



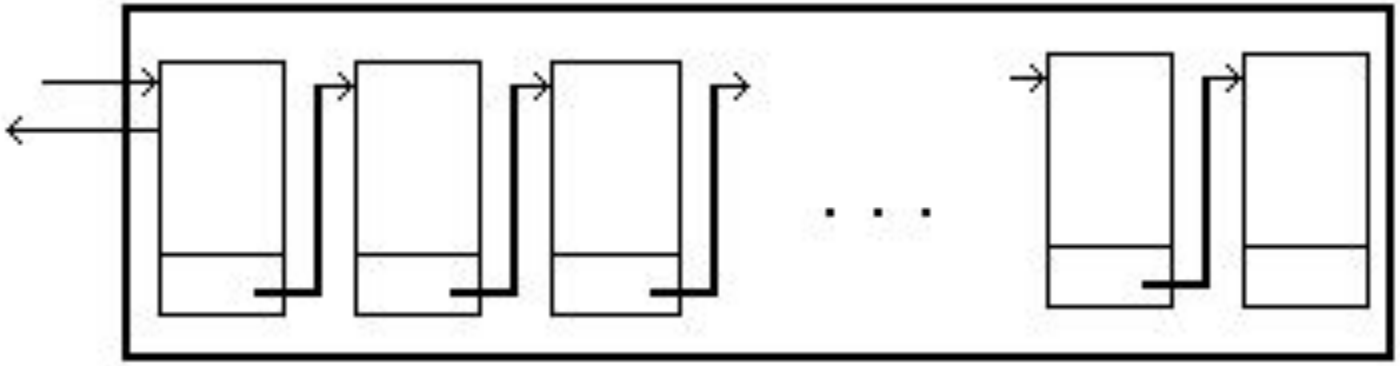
У очереди доступен для включения конец, а для исключения (выборки) ─ начало. Элемент, поступивший в очередь раньше и обслуживается раньше. Говорят, что очередь это структура с дисциплиной обслуживания ***FIFO*** (***F***irst ***I***n, ***F***irst ***O***ut) ─ «первый пришёл, первый ушел».

Структура дека



У дека оба конца доступны, как для включения, так и для выборки. Таким образом, можно сказать, что дек ─ это <u>двусторонняя очередь.</u>

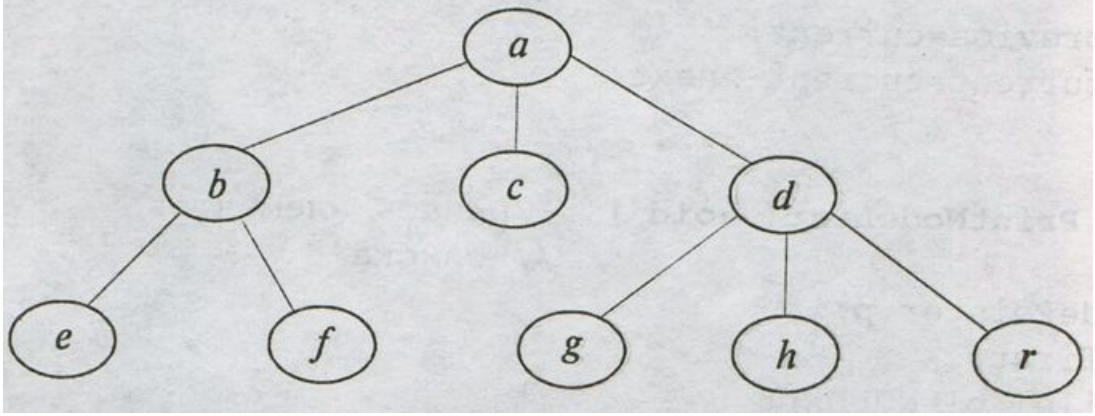
Структура стека



У стека для взаимодействия доступна только один конец структуры ─ вершина стека. И включение нового элемента в стек и выборка последнего ранее включенного идет через вершину стека. Таким образом на обслуживание попадает первым элемент, поступивший последним. Говорят, что стек ─ это структура с дисциплиной обслуживания ***LIFO*** (***L***ast ***I***n, ***F***irst ***O***ut) ─ «последним пришёл, первым ушёл».

Структура дерева

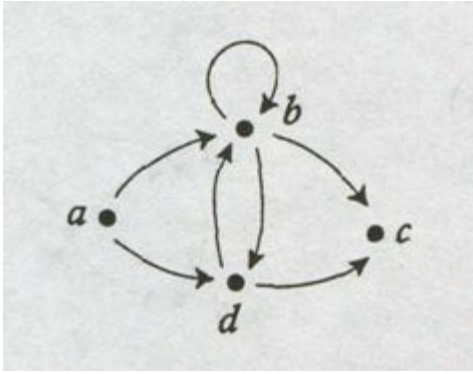
Деревом называется структура данных, элементы которой связаны отношениями подчиненности, когда одному элементу может быть подчинено несколько, но при этом он может сам быть подчинен только одному. В структуре имеется только один элемент, не подчиняющийся никаким другим. Это элемент называется ***корнем дерева***.



Корнем дерева является элемент ***a***, он находится на самом высоком уровне. В дереве нет элементов, которым подчиняется корень. Узлы ***b***, ***c*** и ***d*** подчиняются (порождаются, являются дочерними узлами) узлу ***a***. Узел ***a*** является для них ***родительским***, ***порождающим***. В свою очередь узлы ***b*** и ***d*** имеют по несколько дочерних узлов, а узел ***c*** таковых не имеет.

Структура графа

Структурой данных наиболее общего вида является ***граф***. Любые элементы графа могут быть связаны с любыми другими элементами.



Модели и моделирование

В любой научной дисциплине часто приходится использовать важнейшие *общенаучные* понятия **модель** и **система**. Эти понятия постоянно встречаются и в повседневной жизни человека. Например:

- большая часть детских игрушек — машинки, куклы, кубики, всевозможные конструкторы и т.д. являются моделями реальных объектов;
- разнообразные тренажеры для водителей автомобилей, летчиков и космонавтов также представляют собой модели реальных автомобилей, самолётов и космических аппаратов;
- к моделям относятся и манекены в магазинах, и макеты зданий и географические карты, и самые разные чертежи;
- картины художников, скульптуры, памятники также можно считать моделями природных пейзажей, животных, людей, а литературные произведения — моделями различных жизненных ситуации;
- многие простые понятия, которыми оперируют люди, такие как, например, точка, прямая, окружность, куб и т.д. являются идеализацией реальных объектов, их моделями;
- моделями являются представления человека о строении атома, о происхождении различных видов растений и животных, да и самой жизни на Земле;
- исторические представления — это модели событий прошлого.

Различные системы используются в быту, в общественном устройстве, в науке и технике. Например:

- различные системы заполняют наш дом: электрическая, отопительная, водопроводная, газоснабжения;
- кровеносная, нервная, костно-мышечная и другие системы являются важнейшими элементами организма человека;
- общегосударственные системы, такие как образовательная, здравоохранения, оборонительная, продовольственного снабжения и т.д. обеспечивают существование всего общества в целом;
- файловая и операционная системы, базы данных, интернет – примеры программных и аппаратных систем.

Важнейшая роль понятия «модель» является, по-видимому, следствием самого способа мышления человека. В результате наблюдения над любыми объектами, явлениями и процессами в сознании человека формируется собственное, вообще говоря, ***упрощенное*** представление об увиденном или услышанном, которое и является первоосновой существования *любых других* моделей.

Следовательно, на понятии «модель» базируется по существу любой теоретический или экспериментальный метод научного и технического исследования, в котором вместо реальных объектов используются их идеальные или материальные образы.

Важнейшая роль понятия «система» вытекает из того, что реальные объекты, явления и процессы всегда существуют как *совокупности взаимосвязанных, взаимодействующих* между собой элементов. В результате взаимодействия у этой совокупности появляются некоторые *новые качества*, она начинает вести себя *как целое*, а не как совокупность *разрозненных* элементов, что и обозначают термином «система».

Объект (от лат. objectum — предмет) — существующий в реальной действительности (то есть вне человеческого сознания) предмет, биологический организм, физическое тело, явление или процесс, на который направлена деятельность **субъекта** — отдельной личности, группы людей или всего общества в целом.

Часть реального мира, рассматриваемая, изучаемая в рамках какой-либо деятельности или решаемой задачи, совокупность объектов, явлений свойства которых и отношения между которыми рассматриваются в рамках некоторой задачи, называется **предметной областью.**

Характерные, неотъемлемые черты, свойства, качества принято называть ***атрибутами*** объектов, явлений, предметной области.

Любое изучение или использование предметной области, так или иначе, связано с формированием её *упрощенного* образа в сознании человека, то есть её модели.

Модель — это некоторый *заместитель* оригинала, с помощью которого осуществляется *изучение* важных для решаемой задачи свойств оригинала. При этом термин «модель» используется для обозначения устройства-аналога, каким-либо образом воспроизводящего реальный объект, или же его чертежа, схемы, макета, словесного или математического описания реального объекта

Моделью (от лат. modulus — образец) называется *материальный* или *идеальный* образ реального объекта или некоторой совокупности объектов, который при определенных обстоятельствах используется в качестве заменителя или представителя исходных объектов. Это образ, полученный с помощью концентрации внимания только на некоторых важнейших с точки зрения решаемой задачи свойствах рассматриваемых объектов и отбрасыванием всех их несущественных свойств. Реальный объект, для которого построена модель, называется **оригиналом** или **прототипом** модели. **Моделированием** называется процесс *построения* и *использования* модели.

Отвлечение от несущественных деталей принято называть ***абстрагированием***. При абстрагировании осуществляется определенное огрубление реальной действительности. Абстрагирование является одним из важнейших инструментов при построении модели какой-либо предметной области.

Модель считается ***адекватной,*** если она верно отображает важнейшие с точки зрения решаемой задачи особенности реальных объектов или явлений.

Наличие адекватной модели позволяет спрогнозировать свойства и поведение реального объекта в той или иной ситуации, описать развития явления, процесса во времени. Например, адекватная модель атмосферы и необходимые натурные измерения температуры, давления, скорости ветра и т.д. позволяют оценить вероятный характер погоды на ближайший период и на этом основании провести подготовку, соответствующую намечаемым действиям.

Заметим, что построение *абсолютно адекватной модели* принципиально невозможно, поскольку в этом случае модель просто окажется *тождественной* реальному объекту, полностью совпадёт с ним. Поэтому речь может идти только о большей или меньшей адекватности модели. Ясно, что для обеспечения большей адекватности необходимо учитывать как можно больше факторов, свойств реального объекта.

С другой стороны желательно, чтобы построенная модель была *проста* в использовании и изучении, чтобы полученную на её основе задачу можно было решить существующими способами. Но практика показывает, что большинство задач, которые *в полном объеме* учитывают основные влияющие факторы, *точно решить невозможно*.

Но практика показывает, что большинство задач, которые *в полном объеме* учитывают основные влияющие факторы, *точно решить невозможно*. Кроме того, некоторые факторы могут быть либо вообще неизвестны, либо невозможно получить их значения.

В таких случаях формируется более простая модель, постановка задачи упрощается и вместо неё решается другая, более простая. Однако эта более простая задача должна всё-таки правильно, *адекватно* отображать основные закономерности рассматриваемой предметной области, задачи.

Очевидным образом наблюдаем противоречие между требованиями адекватности и простоты: чем адекватнее модель, тем она сложнее, чем проще модель, тем она менее адекватна. Ясно, что между этими свойствами модели приходится искать компромисс.

Основой для такого компромисса может служить **точность** модели, которую можно характеризовать *степенью совпадения* полученных в процессе моделирования результатов с наблюдаемыми свойствами реальных объектов. При этом необходимая степень совпадения определяется решаемой задачей.

Обсуждая точность модели и моделирования важно помнить о том, что точность результатов расчетов *не может превысить* точность исходных данных, приборов, измерительных инструментов, используемых для наблюдения за реальными объектами.

Кроме того, точность модели должна быть согласована с желательной точностью результатов, которые должны соответствовать реальным потребностям практики. Скажем, бессмысленно стремиться составлять прогноз температуры воздуха на следующий день с точностью 0,1 градуса, достаточно указать прогнозное значение с разбросом в 2–3 градуса.

Одним из самых важных свойств модели является её **потенциальность** (от лат. potentia — мощь, сила) или **предсказательность**, под которой понимается возможность получения новых знаний о реальном объекте на основании изучения модели, а также возможность составления прогноза поведения объекта в тех или иных условиях.

Ярким примером предсказательной мощи моделей является периодический закон, на основании которого Д.И. Менделеев предсказал существование элементов экаалюминия, экабора и экасицилия. Спустя несколько лет были найдены элементы галлий, скандий и германий, физические и химические свойства которых с удивительной точностью соответствовали предсказаниям Д.И. Менделеева.

Французский учёный У. Леверье на основании математических расчётов, выполненных с использованием закона всемирного тяготения и экспериментально полученных данных о движении уже известной планеты Уран, предсказал положение ещё неизвестной планеты Солнечной системы Нептун.

Применяя для решения той или иной задачи некоторую модель необходимо всегда оценивать ***правомерность*** её использовании. Эта оценка всегда производится на основании анализа области применимости модели и существующих ограничений на её применение, так или иначе вытекающих из принятых при её построении упрощений и предположений. Необоснованное применение модели, пренебрежение ограничениями на её применение является очень распространённой ошибкой, которая может привести к очень неприятным последствиям.

- Моделирование это всегда упрощение. Это замена сложного более простым. Получается решение другой, более простой задачи. Это решение можно трактовать как некоторое приближенное, неточное решение исходной задачи. Но это решение может быть достаточно хорошим приближением к искомому и давать возможность строить прогноз, предсказывать поведение изучаемой системы с необходимой степенью точностью.
- Модель есть упрощение некоторого оригинала и без оригинала не может быть построена.
- Моделирование является единственным способом описания и изучения реального мира.
- Моделирование это обязательный, необходимый этап решения любой задачи.
- Одна и та же проблемная область может иметь несколько различных моделей, описывающих ее с различных точек зрения. Разные задачи изучающие одну и ту же проблемную область могут потребовать построения различных моделей.
- При наличии нескольких моделей и возможности сопоставления их с оригиналом обычно считается, что та модель лучше, которая более точно соответствует оригиналу, более адекватна реальности.

Классификация моделей

Классификацией называется распределение рассматриваемых объектов по группам в соответствии с выбранными признаками, свойствами.

Любая классификация включает выделение общих свойств, признаков у рассматриваемых объектов и, при необходимости, определение значений классификационных признаков, по которым объекты будут распределяться в различные группы, классы, категории

Классификация может быть представлена в виде графа (чертежа), в виде таблицы, списка или группы списков.

Натурные модели

Натурные модели представляют собой материальные объекты, которые адекватно отображают выбранные свойства объекта, предметной области.

Примеры натурных моделей: игрушечные или действующие модели автомобиля, самолета и т.д., манекены, фотографии, макеты зданий, тренажеры, протезы, заменяющие и частично выполняющие функции настоящих органов.

Информационные модели

Информационная модель представляет собой нематериальный, воображаемый образ объекта, явления, предметной области. Примеры: математический график функции, таблица Менделеева, различные описания, характеристики, запись шахматной партии, нотная запись мелодии.

Математические модели

Математическая модель это разновидность информационной модели, в которой для описания предметной области используется математический формализм ─ математические соотношения, уравнения, зависимости. ***Формальным*** или ***формализованным*** называется представление подчиняющееся фиксированной систем правил. Пример: математическое описание движения спутника Земли.

Математическая модель (в узком смысле, с точки зрения математики) ─ это одно или несколько множеств элементов произвольной природы, на которых определено конечное множество отношений.

Структурные модели

Модели, в которых отображается структура и состояние предметной области, называются ***структурными***. Пример: чертеж, макет.

Функциональные модели

Модели, в которых отображается изменение состояния изучаемых объектов, явлений, процессов с течением времени, называются ***функциональными***. Пример: макет, демонстрирующий работу двигателя внутреннего сгорания.

Имитационные модели

Имитационными моделями считаются натурные или информационные модели, которыми заменяются реальные изучаемые объекты или явления в ходе проведения натурных или вычислительных экспериментов с целью получения информации об структуре и поведении изучаемых объектов, явлений. Примеры: натурная модель самолета в аэродинамической трубе, математическая модель атмосферы в задачах метеопрогноза.

Системы

Системой называется сложная структура, состоящая из взаимодействующих компонентов, каждый из которых в отдельности не обладает свойствами, присущими системе в целом. Свойства, которые присущи системе в целом и не присущи никакому ее компоненту, называют ***системными свойствами***. Примеры систем: телевизионный приемник, дом, самолет.

Целостность системы означает, что удаление из нее какого-либо компонента приводит к тому, что система теряет определенное системное свойство, т.е. фактически приводит к исчезновению системы.

В общем случае системы строятся из простых объектов, которые не имеют внутренней структуры, и из сложных, которые в свою очередь состоят из взаимодействующих простых, и, следовательно, являются системами. Системы, являющиеся компонентами других систем, принято называть ***подсистемами***. .

Изучение системы осуществляется с помощью построения ее модели. Модель сложного объекта также будет сложной структурой, это значит, что сама модель может быть системой.

Выделение систем связано с постановкой и решением следующих задач:

- Изучение предметной области, объекта, совокупности объектов, явления, выяснение строения, взаимосвязей между компонентами, характера протекающих в системе процессов, влияния внешних и внутренних факторов.
- Описание системы языковыми или графическими средствами.
- Построение новой системы.
- Использование существующей системы в практических целях.

Для решения задач, связанных с системами, используются методы ***анализа*** и ***синтеза*** систем.

Анализ представляет собой метод исследования, основанный на выделении отдельных компонентов системы, изучении их свойств и взаимосвязей.

Синтез представляет собой метод исследования системы в целом, то есть всех ее компонентов во всех их взаимосвязях. Синтезом также считается создание системы путем соединения всех ее компонентов на основании законов, задающих их взаимосвязь.

Статические и динамические системы

Система называется ***статической*** если множество компонентов, из которых она состоит, множество их свойств и взаимосвязей, а также множество системных свойств не изменяется с течением времени.

Любое изменение указанных множеств считается ***процессом***, происходящим в системе, а сама система становится ***динамической***.

Статическую систему можно рассматривать как мгновенное состояние динамической системы.

Частным случаем динамической системы является **равновесная** система, в которой одновременно происходит несколько уравновешивающих друг друга процессов. Равновесная система может также считаться частным случаем статической системы, если рассматривать неизменность ее состояния в течение длительного отрезка времени.

Замкнутые и незамкнутые системы

Система считается ***замкнутой***, если ее компоненты не взаимодействуют с внешним миром, с объектами, не принадлежащими системе. Это значит, что отсутствуют потоки вещества, энергии и информации из системы или в систему. В противном случае система считается ***незамкнутой***.

Различают ***естественные***, ***искусственные*** и ***информационные*** системы. Естественные и искусственные системы материальны, информационные системы нематериальны.

Частным случаем информационных систем являются ***формальные*** системы. Точнее, формальная система является разновидностью математической модели. В формальной системе фиксируется исходное множество компонентов и связей (отношений) между ними, а также множество правил порождения новых объектов. Формальные системы служат для создания различных математических дисциплин (математическая логика, теория множеств), а также являются важным инструментом информатики, служащим для создания языков программирования и трансляторов.

Тема 4. Введение в теорию информации и теорию кодирования

Одно и то же дискретное сообщение можно представить в *различных* алфавитах, но при этом очень важно, чтобы используемое преобразование не приводило к потере исходной информации.

Для естественных языков это преобразование представляет собой *перевод* с одного языка на другой. Для алфавитов, используемых в технических устройствах и в компьютерах, такое преобразование называется **кодированием сообщений**.

Вопросы, связанные с кодированием сообщений, рассматриваются в *теории информации* и *теории кодирования* — двух важных составных частях теоретической информатики

Теория информации определяет *предельные* возможности эффективности кодирования. Она определяет границы возможного, но не дает никаких практических рецептов, как достичь желаемого результата.

Теория кодирования предлагает конкретные, готовые варианты построения кодов и определяет условия их применимости.

Количество информации

Для решения многих практических задач информатики, например, для определения места на носителе, которое потребуется для хранения сообщения или времени, которое займет его передача, необходимо уметь некоторым образом *измерять сообщения*, также как, например, при решении задач перевозки грузов требуется умение измерять их вес.

Следовательно, необходимо иметь некоторую *количественную характеристику* сообщений и располагать способом её определения или измерения. Такой характеристикой сообщений является *количество информации*.

Количество информации представляет собой количественную характеристику *сообщений*, численную меру, связанную с объёмом носителя (жёсткого или оптического диска, «флешки» и т.д.), необходимого для хранения сообщения, требуемым временем передачи сообщения между источником и приёмником и другими параметрами хранения, передачи и обработки сообщений.

Эта характеристика не имеет ничего общего с измерением информации, понимаемой как *смысл сообщения*. Более правильным было бы название *количество сообщения*, но в научной терминологии закрепилось исторически сложившееся название «количество информации».

Существует два основных способа определения количества информации, один из которых называется **объемным**, а другой, рассматриваемый в этом и последующих разделах, называется **вероятностным**. Более простой объёмный подход чаще всего применяется в практических приложениях информатики, а вероятностный — в теоретических её разделах.

Вероятностный подход, предложенный К. Шенноном в 1948 году, базируется на следующих исходных представлениях.

При выяснении свойства полноты информации, было отмечено, что если принятое сообщение содержит *неполную* информацию, то остается некоторая *неопределённость*, которая в свою очередь приводит к необходимости выбирать действие из некоторого набора вариантов.

Если в сообщении о встрече приезжающего указана только дата встречи, то можно выбрать наугад один из приходящих в указанный день поездов и, кроме того, какой-либо вагон этого поезда. Ясно, что такой *произвольный* выбор, *скорее всего*, окажется неудачным. Поэтому для получения гарантированного результата необходимо организовать *полный перебор* всех приходящих поездов и всех вагонов в каждом из них, что потребует значительных ресурсов.

Теперь представим, что получено сообщение, содержащее более полную информацию — указаны дата, время прибытия и номер поезда. В этом случае *неопределённость* ситуации значительно меньше. Меньше и количество возможных вариантов при выборе действий — остается выбрать только один из вагонов указанного поезда.

И, наконец, при получении сообщения, содержащего всю необходимую информацию (день, номер поезда, время прибытия, номер вагона), неопределённость ситуации полностью исключена, выбор варианта действий отсутствует.

Имеющуюся в сообщении неопределённость, *недостающую информацию* принято называть **информационной энтропией** или просто **энтропией**.

Итак, между понятиями информация, энтропия (неопределенность) и возможность выбора существует тесная связь. Наличие любой неопределённости, то есть ненулевая энтропия, предопределяет необходимость и возможность выбора, а отсутствие неопределённости, то есть нулевая энтропия — исключает возможность выбора.

Получение в сообщении некоторой дополнительной информации приводит к соответственному уменьшению *имеющейся* энтропии, а получение полной информации сводит энтропию к нулю.

Проведённые выше *качественные* рассуждения представляют собой близкую к существу дела, но не точную аналогию, которая *опирается на смысловую сторону* сообщения. Для получения *не связанной со смыслом* сообщения *количественной* его характеристики следует уточнить обсуждаемые представления.

Мы установили, что любые дискретные сообщения могут быть представлены в виде конечной последовательности знаков, принадлежащих некоторому конечному алфавиту.

Перед приёмом сообщения естественно считать, что существует *полная неопределенность* в том какие знаки и в каком порядке поступят. Это означает, что энтропия сообщения равна некоторому исходному значению.

После получения первого знака *неопределенность уменьшается*, то есть исходная энтропия убывает на какую-то величину.

Но ещё останется неопределенность с получением второго и всех последующих знаков. Получение второго, третьего и каждого из последующих знаков также приведёт к соответствующим уменьшениям энтропии. После получения последнего знака энтропия окажется равной нулю.

Итак, с появлением ***каждого*** очередного знака сообщения связано некоторое количество полученной информации, приводящей к соответствующему уменьшению исходной энтропии.

Всё количество полученной информации оказывается равным исходному значению энтропии только после появления последнего знака сообщения.

Вместо измерения количества информации можно измерять энтропию, считая, что количество полученной информации численно равно исходной энтропии сообщения.

Целесообразность измерять количество информации с помощью энтропии основано на том, что определение энтропии относится к задачам, способы решения которых хорошо известны.

Энтропия является числовой характеристикой так называемых **случайных событий**, которые отличаются принципиальной *неопределенностью* в их появлении.

Измерить каким бы то ни было способом информацию *в её смысловом аспекте*, измерить нематериальное *понимание* человеком содержания принятого сообщения *в принципе невозможно*. Как-то измерить можно только материальное её представление — сообщение.

Вероятность

Изучение присущих природе закономерностей, выявление существующих взаимосвязей, взаимозависимостей в тех или иных реальных ситуациях производится с помощью *неоднократных наблюдений* над объектами, явлениями, процессами, а также *многократно осуществляемых опытов*, *экспериментов*, которые в дальнейшем для единообразия называются **опытами** (или **испытаниями)**.

Результаты таких наблюдений, испытаний или опытов принято называть **событиями** (или **исходами)** и обозначать большими латинскими буквами *A*, *B*, *C* и т.д.

Все события можно разделить на две группы — **детерминированные** и **случайные**.

Детерминированными называются события, которые определяется однозначными причинно-следственными связями и всегда воспроизводятся в неизменном виде. То есть исходы *всех* опытов оказываются *одинаковыми* при любом количестве их наблюдений или повторений.

В качестве примера детерминированных событий можно рассмотреть результаты бросаний металлической монеты в воду. Сколько бы наблюдений над такими бросаниями не производилось, исход будет один и тот же — монета тонет в воде.

События, исход которых существенно зависит от множества неизвестных или трудно учитываемых факторов, называются ***случайными.*** Случайные события воспроизводятся различным образом при различных повторениях опыта.

Случайным событием называется опыт, который можно повторять любое количество раз, и каждый раз исход опыта может отличаться от предыдущего — имеется неопределенность в исходе единичного опыта, при возрастании количества опытов наблюдается стабилизация их результатов

Множество всех возможных исходов опыта называется **пространством элементарных исходов**.

Элементарные исходы, во-первых, должны быть неразложимы на более простые, во-вторых, должны взаимно исключать друг друга и, в-третьих, в любой опыт обязательно должен заканчиваться одним из элементарных исходов.

Подбрасывая монету, несложно убедиться в том, что результатом может быть выпадение либо «герба», либо «решки».

Каждый такой результат и есть элементарный исход. Видно, что любой из исходов исключает другой, и оба они не сводятся к более простым. Пространством элементарных исходов этого опыта является множество {«выпал герб», «выпала решка»}, а общее количество исходов равно двум.

Попутно заметим, что два — это *минимально* необходимое количество возможных исходов для существования *случайного* события.

Если пространство элементарных исходов состоит только из *одного* элемента, то это *детерминированное* событие.

Пусть опыт заключается в бросании игрального кубика, тогда выпадение одного очка, двух очков, трёх очков и т.д. — это элементарные исходы, а множество {1, 2, 3, 4, 5, 6} — представляет собой пространство элементарных исходов опыта с количеством элементов равным шести.

Вытаскивание наугад игральной карты из колоды в 36 карт имеет пространство из 36 элементарных исходов вида {6♠, 7♠, …,T♠, 6♣, 7♣, …,T♣, 6♦, 7♦, …,T♦, 6♥, 7♥, …,T♥}.

Пусть в ящике находятся десять шаров из них два белых, три зелёных и пять красных. Пространством элементарных исходов для опыта, состоящего в вытаскивании наугад одного шара, является множество с *десятью* исходами.

Дело в том, что, например, исход «вынут белый шар» не является *элементарным*, так как он разложим на два более простых случая «вынут первый белый шар» и «вынут второй белый шар». Аналогичным образом обстоят дела и с шарами остальных цветов.

В общем случае наступление изучаемого события может быть связано как с одним, так и с несколькими элементарными исходами, так что появление любого из них приводит к наступлению события.

Например: событие «выпадение чётного количества очков» при бросании игрального кубика происходит при наступлении любого из трёх элементарных исходов: выпадение двух, четырёх или шести очков.

Событие «выбор карты красной масти» происходит в результате наступления любого из 18 элементарных исходов: {6♦, 7♦, …,Т♦, 6♥, 7♥, …,Т♥}, а «выбор чёрной восьмерки» — только в результате двух исходов {8♠, 8♣}.

Событие «вынут белый шар» происходит при выборе любого из двух, имеющихся в ящике белых шаров.

Элементарные исходы, при осуществлении которых происходит событие называются ***благоприятствующими*** этому событию. Элементарные исходы, при осуществлении которых событие не происходит называются **неблагоприятствующими**.

Событие ***Z*** считается ***невероятным*** (***невозможным)***, если его наступление не происходит ни при каких обстоятельствах. Для такого события нет благоприятных исходов. Событие ***D*** считается ***достоверным***, если оно наступает всегда, при любых выполнениях опыта. Достоверному событию благоприятствуют любые исходы.

Часто приходится рассматривать *два или более* случайных события, которые происходят *одновременно* или *последовательно* друг за другом. Например, можно бросать два кубика одновременно или рассматривать два последовательных бросания одного. Можно изучать последовательные вытаскивания карт из колоды, цветных шаров из ящика, выстрелы орудия, приём последовательных знаков дискретного сообщения и т.д.

Два события считаются ***совместными***, если они могут произойти одновременно.

Два события называются ***несовместными***, если они не могут произойти одновременно ни при каких обстоятельствах.

Два события считаются ***зависимыми***, если появление одного из них влияет на появление другого. Два события называются ***независимыми***, если между ними отсутствует причинно-следственная связь и появление одного из них не влияет на появление другого.

Суммой двух событий ***A*** и ***B*** считается событие $C=A \vee B$, состоящее в том, что произошло либо событие ***A***, либо событие ***B***, либо и то и другое. ***Произведением*** двух событий ***A*** и ***B*** считается событие $C=A \wedge B$, состоящее в том, что произошли оба события.

Измерение вероятностей

Вероятность p(A) случайного события ***A*** является численной мерой возможности осуществления этого события в опыте. Вероятность невозможного события ***Z*** равна нулю ***p(Z)=0***, вероятность достоверного события ***D*** равна единице ***p(D)=1.*** Для любого события ***A***: ***0 ≤ p(A) ≤ 1***.

Пусть пространство элементарных исходов содержит ***N*** исходов, а событию ***A*** благоприятствуют N_A исходов, тогда:

$$p(A) = \frac{N_A}{N}$$

Пусть в ***K*** опытах событие ***A*** произошло K_A раз. Отношение K_A/K называется ***частотой*** события, при этом:

$$p(A) = \lim_{N \to \infty} \frac{K_A}{K}$$

Если все элементарные исходы, опыта обладают *равными возможностями* для осуществления, то они называются **равновероятными**.

Если разные элементарные исходы опыта имеют различные вероятности, то их называют **неравновероятными**.

Вероятность суммы двух несовместных событий ***A*** и ***B*** равна:

$$p(A \vee B) = p(A) + p(B)$$

Сумма вероятностей всех элементарных исходов любого опыта всегда равна единице

Вероятность произведения двух независимых событий ***A*** и ***B*** равна:

$$p(A \wedge B) = p(A) \times p(B)$$

Энтропия

Анализируя и сравнивая между собой *различные* опыты, можно заметить, что опыты в целом имеют *разную меру неопределённости* в результатах.

Численная мера этой неопределённости, характеризирующая опыт *в целом* со всеми его исходами, представляет собой величину, которую называют **энтропией**.

Вероятность характеризует *отдельный исход опыта*, или некоторое событие, связанное с несколькими исходами, в то время как энтропия характеризует *весь опыт в целом*, а не отдельные его исходы

Наблюдение о неопределенностях при бросаниях кубика и монеты даёт основание предположить, что *чем больше количество исходов, тем больше энтропия*, то есть неопределённость в результате опыта.

Неопределённость в исходе опыта зависит ещё и от вероятностей его элементарных исходов — неопределённость опытов с равновероятными исходами выше неопределённости опытов с неравновероятными исходами

Свойства, которыми должна обладать функция определяющая энтропию:

$$f(1)=0,\ f(n)\underset{n\to\infty}{\to}\infty,\ f(n\times m)=f(n)+f(m)$$

Единственной функцией, которая обладает всеми перечисленными свойствами является функция

$$f(n)=\log_2 n$$

В информатике энтропию принято обозначать буквой *H.*

По определению энтропия любого опыта с *n* равновероятными исходами равна:

$$H=\log_2 n$$

Эта формула называется формулой Хартли.

Если выбрать опыт с двумя равновероятными исходами *n*=2, то получим, что его энтропия равна единице: $\log_2 2=1$. Энтропию этого опыта выбрали в качестве единицы измерения энтропии.

Единица измерения энтропии называется ***бит***

Один бит равен энтропии любого опыта с двумя равновероятными исходами.

Пусть некоторый опыт имеет n элементарных исходов $\{A_1, A_2, \ldots, A_n\}$. Тогда по определению энтропия всего опыта равна сумме энтропий каждого из его *n* его исходов: $H=h(A_1)+h(A_2)+\ldots+h(A_n)$.

Если исходы равновероятны, то их энтропии одинаковы, поэтому энтропии h, вносимые каждым из ***n*** равновероятных исходов одинаковы и равны:

$$h = \frac{1}{n}\log_2 n = -\frac{1}{n}\log_2 \frac{1}{n}$$

Так как вероятность *p* каждого из исходов в этом случае *p=1/n*, можно записать:

$$h = -\frac{1}{n}\log_2 \frac{1}{n} = -p\log_2 p$$

Для опытов с неравновероятными исходами энтропию отдельного исхода с учетом того, что вероятности $p(A_i)$ различных исходов A_i различны, также можно записать в виде

$$h = -p(A_i)\log_2 p(A_i)$$

Тогда полная энтропия опыта с ***n*** неравновероятными исходами $\{A_1, A_2, \ldots, A_n\}$ равна:

$$H = -\sum_{i=1}^{n} p(A_i)\log_2 p(A_i)$$

Эта формула называется формулой Шеннона.

Энтропия является мерой неопределенности опыта, в котором проявляется случайные события. Энтропия равна средней неопределенности всех возможных его исходов.

По определению энтропия произведения независимых опытов равна сумме энтропий отдельных опытов.

$$H(A \wedge B) = H(A) + H(B)$$

Энтропия и количество информации

Количество информации *это числовая характеристика*, отображающая ту меру неопределенности опыта, которая ***исчезает*** после его проведения

Следовательно, количество информации *I*, которое оказывается полученным в результате осуществления опыта, численно ***равно*** его исходной энтропии *H*.

Отсюда следует, что *количество информации*, как и энтропия *измеряется в битах*.

Количество информации, которое получено в результате осуществления опыта численно равно его исходной энтропии.

Для опытов с *n* равновероятными исходами количество информации подсчитывается по

формуле Хартли: $I = \log_2 n;$

Для опытов с *n* неравновероятными исходами и вероятностями исходов $p(A_1)$, $p(A_2)$, ,…, $p(A_n)$ количество информации подсчитывается по

формуле Шеннона: $I = -\sum_{i=1}^{n} p(A_i) \times \log_2 p(A_i)$

Значение входящего в каждое слагаемое выражения $-\log_2 p(A_i)$ можно рассматривать, как **частную информацию**, связанную с одним элементарным исходом A_i.

А *общее* количество информации как *среднее* этих частных информаций по всем возможным исходам опыта.

В качестве примера, определим количество информации, полученное в результате опытов с монетой и с вытаскиванием карты из колоды в 32 карты.

В каждом из этих случаев элементарные исходы *равновероятны*, поэтому применяя формулу Хартли для опыта с монетой, получим 1 бит, а для опыта с вытаскиванием карты 5 бит.

Пусть вероятность перегорания лампочки при её включении равна p=1/1000, то есть лампочка перегорает только в 1 случае из 1000. Тогда вероятность её нормального включения q= 999/1000, то есть 999 случаев из 1000.

Так как исходы *неравновероятны*, то расчёт информации следует проводить по формуле Шеннона. Её применение дает $1/1000\log_2(1/1000)-999/1000\log_2(999/1000) \approx 0{,}0014$ бита.

Обратите также внимание на частную информацию исхода, приводящего к перегоранию $Ip = \log_2(1/1000) \approx 9{,}96$ бита, и на частную информацию исхода, не приводящего к перегоранию: $I_q = \log_2(999/1000) \approx 0{,}001$ бита

Появление *редко* встречающегося исхода несет с собой гораздо *больше* количество информации, чем появление часто встречающегося исхода.

Для практики это означает, что сообщение о редко встречающемся исходе окажется, вообще говоря, длиннее и на его передачу потребуется больше времени, чем для сообщения о более часто встречающемся исходе.

Количество информации и алфавит

Приём дискретного сообщения можно рассматривать как *последовательное* осуществление *независимых* опытов по определению каждого очередного знака сообщения, то есть по случайному выбору одного знака из совокупности знаков, имеющихся в алфавите.

Для каждого такого опыта можно определить его энтропию, которая определяется количеством элементарных исходов — количеством знаков в алфавите и вероятностями их появления в сообщениях.

Следовательно, можно найти численно равное энтропии количество информации, получаемое с каждым очередным знаком сообщения.

А из этого вытекает, что количество информации, связанное *с одним знаком* сообщения, *зависит только от используемого алфавита* и совершенно не зависит от конкретного принимаемого сообщения.

Если известно, что все знаки алфавита имеют *равную* вероятность появиться в сообщении, то определить количество информации, которое несёт с собой *каждый знак любого сообщения* можно по формуле Хартли

Обозначение	**Способ**	**Английский**	**Русский**
I_0	Хартли	**4,76**	**5,09**

Разумеется, из этих результатов не следует делать вывод о том, что русский язык более богат, чем английский, так как такой расчёт не имеет никакого отношения к *смысловой* стороне текстов.

Лингвистическое богатство языка определяется количеством различных слов и их сочетаний, а это мало связано с количеством букв в алфавите.

С точки зрения практики этот результат означает, что сообщения из *равного* количества букв на английском и русском языках имеют разную длину *в битах* (и соответственно, разное время передачи), при этом большие длины (и времена передачи) имеют сообщения на русском языке.

Вместе с тем, хорошо известно, что одни буквы встречаются в текстах гораздо чаще, чем другие. Например, чаще всего в русских текстах встречается буква «о», а реже всего — буква «ф». Таким образом предположение о равной вероятности появления знаков алфавита *несправедливо*, и формула Хартли может использоваться только для вычисления *максимального* количества информации, которое приходится на один знак.

К сожалению, определить *реальные вероятности* появления, например, букв русского алфавита в различных текстах невозможно, поскольку общее количество всевозможных текстов *бесконечно* и, следовательно, невозможно определить ни общее количество элементарных исходов, ни количество благоприятствующих исходов.

Поэтому путём анализа *большого количества* разных текстов специалисты определяют *частоты* появления букв, которые естественно считать *приближёнными* значениями обсуждаемых вероятностей.

Частоты некоторых букв русского алфавита

о	е,ё	а	и	т	н	с	р	в	л
0,090	0,072	0,062	0,062	0,053	0,053	0,045	0,040	0,038	0,035

Соображение о неравновероятных исходах появления знаков алфавита в принимаемых сообщениях приводит к выводу о необходимости использования для подсчёта количества информации формулы Шеннона.

Вместе с тем, анализ текстов показывает, что для любого естественного языка существуют более часто встречающиеся *пары* знаков, так же как существуют и такие пары, которые нигде и никогда не встречаются. Так, в русском языке пары букв «щц» и «фъ» не встречаются ни в каких словах.

Следствием из этих рассуждений является вывод о том, что необходимо проводить более тонкие расчеты, считая независимыми появление уже всевозможных *пар* соседних знаков и отслеживая частоты их появления.

Подобные рассуждения можно распространить не только на соседние пары знаков, но и *блоки*, состоящие из трёх, четырёх, пяти и т.д. знаков, и для каждого случая выполнить соответствующие расчёты количества информации

Обозначение	Способ	Английский	Русский
I_1	Шеннон	**4,04**	**4,36**
I_2	Пары	**3,32**	**3,52**
I_3	Тройки	**3,10**	**3,01**

$I_0, I_1, I_2, \ldots$ — убывающая последовательность.

С практической точки зрения вычисление количества информации с учётом связей между отдельными знаками естественных алфавитов, то есть вычисление значений I_2, I_3, I_4 … и т. д. вызывает значительные сложности, поэтому, как правило, ограничиваются расчётом $I_0(A)$ или $I_1(A)$, для которых появление каждого следующего знака происходит *независимо* от всех предшествующих.

Итак, будем считать, что ***каждый знак*** дискретного сообщения несёт с собой количество информации $I_0(A)$ или $I_1(A)$, которое полностью определяется только свойствами используемого алфавита A.

Общее количество информации I_S в сообщении S равно *сумме* количеств информации от всех его знаков: $I_S = L * I(A)$, где L — количество знаков в сообщении, а сомножитель $I(A)$ равен $I_0(A)$ или $I_1(A)$ в зависимости от выбранного способа подсчёта.

Количество информации в сообщении численно равно *минимально* необходимому количеству двоичных цифр, которые требуются для представления сообщения в виде машинного кода без потери его первоначального смысла.

Этот подход позволяет рассматривать понятие «информации» (точнее — сообщения) только с *количественной* стороны, безотносительно к её смысловому содержанию и ценности для получателя.

При таком подходе одна страница печатного текста содержит одно и то же количество информации, которое определяется только количеством знаков на странице и её оформлением, вне зависимости от того, что на этой странице напечатано: бессмысленный, случайный набор знаков, отрывок из романа «Война и мир» или же доказательство математической теоремы.

Такой подход вполне правомерен для решения сугубо технических вопросов, связанных с организацией безошибочной и экономичной передачи сообщений по каналам связи, а также задач их хранения и обработки

Кодирование сообщений

Дискретное сообщение исходя из технических соображений или из особенностей органов чувств человека, участвующих в приеме или передаче сообщений, разбивают на конечные подпоследовательности знаков, которые принято называть ***словами***. Количество знаков в слове называется ***длиной слова.*** Если все слова языка имеют одно и то же количество знаков, то такие слова называются ***n-разрядными*** (n ─ длина слова)

Слова записанные в двоичном, шестнадцатеричном и некоторых других алфавитах, принято называть ***кодом*** (двоичное слово ─ двоичный код и т.д.).

Исходный алфавит *A1*, в котором записано сообщение, называется ***первичным*** алфавитом. Целевой алфавит *A2*, в который преобразуется сообщение называется ***вторичным*** алфавитом.

Правило, описывающее соответствие между знаками первичного алфавита и знаками или сочетаниями вторичного называется ***кодом***.

Совокупность всех используемых в преобразовании соответствий называется **кодовой таблицей**. Для обеспечения необходимых свойств операции кодирования между знаками первичного и знаками или группами знаков вторичного алфавитов должно быть установлено *взаимно однозначное соответствие* (биекция), при этом кодовая таблица представляет собой способ задания этого соответствия.

Пример таблицы кодирования:

Знак	Частота	Код	Длина кода
А	0,34	1110	4
К	0,14	10001	5
М	0,15	10101	5
Н	0,26	1010	4
У	0,11	001001	6

Последовательность знаков вторичного алфавита *A2*, которыми представлен *один* знак первичного алфавита *A1* называется **кодовым словом**, **кодовой цепочкой** или просто **кодом знака**.

Кодированием называется последовательность действий по переводу сообщения из первичного во вторичный алфавита.

Декодирование представляет собой операцию, обратную кодированию, то есть это последовательность действий по восстановлению сообщения в исходном алфавите по его виду во вторичном.

Операции кодирования и декодирования считаются ***обратимыми***, если их последовательное применение обеспечивает восстановление исходной информации без потерь.

Кодером называется устройство, обеспечивающее выполнение операции кодирования. ***Декодером*** называется устройство, обеспечивающее выполнение операции декодирования.

Существует огромное количество способов построения кодов при выбранных или заданных первичном и вторичном алфавитах. Поэтому возникает проблема выбора *оптимального*, то есть в некотором смысле наилучшего кода.

Оптимальным называется кодирование сообщения, результат которого обеспечивает *минимально* возможное для используемых алфавитов и передаваемого сообщения время передачи по каналам связи и минимальные требования к памяти при хранении.

Эффективным считается кодирование с результатом *достаточно близким* к оптимальному. Эффективное кодирование применяется в тех случаях, когда для получения оптимального кода требуются значительные временные ресурсы или не хватает каких-либо данных для его построения.

Задачи эффективного и оптимального кодирования часто решаются с помощью **сжатия** сообщений.

Различают два вида сжатия — без потери информации и с потерей информации.

Сжатие без потери информации применяется при кодировании текстовых и числовых видов данных, то есть там, где потеря информации недопустима, поскольку может привести к неправильному пониманию текста или к вычислениям с непредсказуемым результатом.

Сжатие с потерей информации широко применяется при кодировании графики, звука и видео, так как в этих случаях возможно удаление некоторой части данных без существенных отличий для восприятия человеком оригинала от полученного результата сжатия. В основе применения сжатия с потерями лежит естественное квантование органами слуха и зрения человека сообщений, получаемых из внешней среды.

Выбор оптимального кода это *технический и экономический фактор*, поскольку касается затрат (и не только временных) на выполнение кодирования и декодирования, на хранение и на передачу кода из одного места в другое и т.д. Естественно желание сделать такие затраты минимально возможными.

Однако свести их к нулю в принципе невозможно. Существующие границы эффективности кодирования, которые могут быть достигнуты, а также условия, при выполнении которых эти границы достижимы, определены в работах К. Шеннона, сформулировавшего и доказавшего ряд базовых теорем кодирования.

Пусть в качестве первичного используется алфавит *A1*, в котором количество информации приходящейся на один знак равно *I*(*A1*)бит.

Пусть далее вторичным является алфавит *A2*, в котором количество приходящейся на один знак информации равно *I*(*A2*) бит.

Если длина исходного сообщения равна *L1*, то общее количество информации, которое связано с сообщением в *первичном* алфавите, равно *L1*×*I*(*A1*) бит .

Допустим, что во вторичном алфавите на преобразованное сообщение потребовалось *L2* знаков. Тогда общее количество информации, связанное *с этим же* сообщением во *вторичном* алфавите, равно *L2*×*I*(*A2*).

Рассмотрим, например, алфавиты *A1*={А, К, М, Н, У} и *A2*={0, 1}. Пусть также имеется кодовая таблица вида:

Знак *A1*	Код из знаков *A2*
А	1110
К	10001
М	10101
Н	1010
У	001001

Пусть требуется закодировать исходное сообщение «НАУКА».

Во вторичном алфавите это сообщение получим в виде «1010 1110 001001 10001 1110» (пробелы между кодами соседних знаков сообщения включены только для удобства чтения). Рассчитайте количество информации на один знак и общее количество информации в сообщении в первичном и вторичном алфавитах.

Используя для подсчёта количества информации формулу Хартли, получим: $I_0(A1)=\log_2 5 \approx 2{,}32$ бита, $I_0(A2)=\log_2 2=1$ бит.

Длины сообщений равны соответственно *L1*=5 и *L2*=23 знака (пробелы не учитываются, так как знак пробела не входит во вторичный алфавит)

Таким образом, количество информации, связанное с сообщением, в первичном алфавите равно *I*(*A1*)≈ 11,6 бита, а во вторичном — *I*(*A2*)= 23 бита.

Это результат означает, что при использовании первичного алфавита требуется примерно в два раза меньше места для хранения (или времени для передачи), чем при использовании вторичного алфавита.

В соответствии с определением, операция обратимого кодирования не может *уменьшить количество информации в сообщении*, потому что в этом случае будет потеряна некоторая часть исходного сообщения. Связанная с этой потерянной частью сообщения информация (смысл этой части) будет также утеряна.

Условие неисчезновения информации при кодировании:

$$L1 \times I(A1) \le L2 \times I(A2) \qquad \overline{L1 \times I(A1)} \le \overline{L2 \times I(A2)}$$

$$I(A1) \le \overline{L2 / L1} \times I(A2)$$

Отношение *L2/L1* имеет смысл количества знаков вторичного алфавита, которое требуется для кодирования одного знака первичного алфавита в конкретном *рассматриваемом* сообщении.

В обсуждавшемся примере *L2*/*L1*=23/5=4,6, то есть на один знак исходного алфавита потребовалось 4,6 двоичных знака.

Значение *L2*/*L1* существенно зависит от кодируемого сообщения. Например, для сообщения «МАМА» и результата кодирования «10101 1110 10101 1110» получим *L1*=4, *L2*=18 и *L2*/*L1* = 18/4=4,5.

Усреднённый коэффициент $\overline{L2/L1}$ может пониматься как *среднее* количество знаков вторичного алфавита *A2*, которое требуется на кодирование *одного* знака первичного алфавита *A1*, при передаче ***<u>любых</u>*** сообщений.

Среднее количество знаков $\overline{L}_{кода} = \overline{L2/L1}$ вторичного алфавита *A2*, которое приходится использовать для кодирования *одного* знака первичного алфавита *A1*, зависит *<u>только</u>* *<u>от выбранного способа кодирования</u>* и совершенно не зависит от конкретных кодируемых сообщений. Эта величина называется **средней длиной кодового слова**, **средней длиной кодовой цепочки**, **длиной кодового слова**, **длиной кодовой цепочки**

Неравенство $I(A1) \le \overline{L2/L1} \times I(A2)$, выражающее условие неисчезновения количества информации при кодировании, можно переписать в виде:

$$I(A1) \le \overline{L}_{кода} \times I(A2) \qquad \text{или} \qquad \overline{L}_{кода} \ge I(A1)/I(A2)$$

Введём обозначение $L_{\min}(A1, A2) = I(A1)/I(A2)$

Тогда получим, что для ***<u>любых вариантов</u>*** кодирования *без потери* количества информации длина кодовой цепочки выбранного способа кодирования ограничена снизу значением:

$$\overline{L}_{кода} \ge L_{\min}(A1, A2)$$

При преобразовании сообщений без потери количества информации из алфавита *A1* в алфавит *A2* *минимально* возможная средняя длина кодовой цепочки L_{min} равна отношению количеств информации, которые в этих алфавитах приходятся на один знак: *I*(*A1*)/*I*(*A2*).

Обычно $I(A1)>I(A2)$, следовательно $L_{min}>1$

Теорема Шеннона о кодировании при отсутствии помех

При отсутствии помех всегда возможен такой вариант кодирования сообщения, при котором средняя длина кодовой цепочки (среднее количество знаков кода, приходящее на один знак первичного алфавита) будет сколь угодно мало отличаться от отношения средних информаций на знак в первичном и вторичном алфавите.

Математически содержание этой теоремы можно выразить следующим образом: для любого сколь угодно малого наперёд заданного числа ε можно построить способ кодирования, для которого средняя длина кодовой цепочки удовлетворяет неравенству

$$|\overline{L}_{кода} - \frac{I(A1)}{I(A2)}| < \varepsilon \qquad \text{или} \qquad |\overline{L}_{кода} - L_{\min}(A1, A2)| < \varepsilon$$

Обратите внимание! Здесь $\overline{L}_{кода}$ обозначает *фактически* полученную при выбранном ***способе кодирования*** среднюю длину кодовой цепочки,

а $L_{\min}(A1, A2)$ обозначает минимально возможную для используемых алфавитов длину кодовой цепочки.

Относительной избыточностью кода ***Q(A1,A2)*** называется отношение

$$Q(A1,A2)=\frac{\overline{L}_{кода}-L_{\min}(A1,A2)}{L_{\min}(A1,A2)}=\frac{\overline{L}_{кода}}{L_{\min}(A1,A2)}-1$$

Второй вариант формулировки теоремы Шеннона: «При отсутствии помех всегда возможен такой вариант кодирования сообщения, при котором относительная избыточность кода будет сколь угодно близкой к нулю».

Если относительная избыточность в точности равна нулю, то кодирование называется **оптимальным**.

Способы кодирования *без потери информации*, обеспечивающие минимизацию относительной избыточности, принято называть **энтропийным кодированием** в связи с тем, что *энтропия* источника сообщения дает нижнюю границу средней длины кодовых цепочек, меньше которой её сделать невозможно.

Для подсчёта минимальной возможной длины кодовой цепочки

$$L_{\min} = I(A1)/I(A2)$$

из сравнения двух используемых вариантов расчёта $I(A)$ вытекает, что для получения *минимально возможного теоретического* значения средней длины кодовой цепочки $L_{\min}$ следует выбирать:

1. для первичного алфавита — как можно меньшую меру $I(A1)=I_1(A1)$, которая подсчитывается по формуле Шеннона,
2. для вторичного алфавита — как можно большую меру определения количества информации $I(A2)=I_0(A2)$, которая подсчитывается по формуле Хартли.

Тогда при переходе из первичного алфавита *A1* во вторичный алфавит *A2*, состоящий из M знаков, получим:

$$L_{\min} = \frac{I_1(A1)}{\log_2 M}$$

Если вторичный алфавит является двоичным, то:

$$L_{\min} = I_1(A1)$$

Третий вариант формулировки теоремы Шеннона: «При отсутствии помех средняя длина двоичного кода может быть сколь угодно близкой к среднему количеству информации, приходящемуся на знак первичного алфавита».

Пусть каждому знаку первичного алфавита *A1* с известными частотами

$$\{\widetilde{p}_1, \widetilde{p}_2, ..., \widetilde{p}_n\}$$

появления знаков в сообщениях, сумма которых по определению равна единице, поставлены в соответствие коды из знаков вторичного алфавита *A2* с длинами

$$\{l_1, l_2, ..., l_n\}.$$

На практике вероятности появления знаков алфавита в сообщении, как правило, неизвестны, поэтому в формуле Шеннона используются частоты знаков и количество информации определяется приближённо:

$$L_{min} = I_1(A1) = -\sum_{i=1}^{n} \widetilde{p}_i \log_2 \widetilde{p}_i$$

а фактическая *средняя* длина кодовой цепочки равна

$$\overline{L}_{кода} = \sum_{i=1}^{n} l_i \widetilde{p}_i$$

Пример. Пусть первичный алфавит состоит из пяти знаков с известными частотами появления в сообщениях, а в качестве вторичного выбран двоичный алфавит. Найти минимально возможную длину кодовой цепочки и фактическую среднюю длину, а также относительную избыточность кода.

Ответ: 2,19 знака; 4,51 знака; 106%.

В своей теореме К. Шеннон только доказал теоретическую возможность построения кода с требуемой избыточностью, но конкретных рекомендаций по его построению в доказательстве теоремы нет.

Из общих соображений можно заметить, что для построения эффективного кода следует:

- учитывать частоты появления знаков первичного алфавита в сообщениях, то есть для знаков, имеющих высокую частоту появления в текстах, следует выбирать коды с меньшей длиной, тогда, естественно, для редко встречающихся знаков *придётся* выбирать коды с большей длиной;
- добиваться *равной* вероятности (частоты) использования знаков *вторичного* алфавита;
- при необходимости группировать символы первичного алфавита в блоки и каждому блоку выделять отдельный код.

Фундаментальный смысл обсуждаемой теоремы Шеннона в том, что она, во-первых, определяет минимально возможную для выбранных алфавитов длину кодовой цепочки, а, во-вторых, утверждает, что путем выбора подходящего способа кодирования к этой границе можно приблизиться как угодно близко.

Алфавитное неравномерное двоичное кодирование

Основной принцип неравномерного кодирования: коды, встречающиеся в сообщениях чаще должны иметь меньшую длину.

Проблема распознавания кодовых цепочек

00100010000111010101110000110

Способы распознавания кодовых цепочек отдельных знаков:

- Использование разделительных знаков
- Применение префиксных кодов

Пример неравномерного кода с разделителями:
00 – код разделителя

1) Код разделителя (признак конца знака) можно включить в код знака;
2) Код разделителя не должен находится в середине кода знака;
3) Код знака (кроме пробела) должен начинаться с 1;

Код с разделителями для русского алфавита

Буква	Частота	Код	Длина	Буква	Частота	Код	Длина
Пробел	0,174	000	3	я	0,018	1011000	7
о	0,090	100	3	ы	0,016	1011100	7
е	0,072	1000	4	з	0,016	1101000	7
а	0,062	1100	4	ь,ъ	0,014	1101100	7
и	0,062	10000	5	б	0,014	1110000	7
т	0,053	10100	5	г	0,013	1110100	7
н	0,053	11000	5	ч	0,012	1111000	7
с	0,045	11100	5	й	0,010	1111100	7
р	0,040	101000	6	х	0,009	10101000	8
в	0,038	101100	6	ж	0,007	10101100	8
л	0,035	110000	6	ю	0,006	10110000	8
к	0,028	110100	6	ш	0,006	10110100	8
м	0,026	111000	6	ц	0,004	10111000	8
д	0,025	111100	6	щ	0,003	10111100	8
п	0,023	1010000	7	э	0,003	11010000	8
у	0,021	1010100	7	ф	0,002	11010100	8

K(r,2)=4,964; Q(r,2)=4,964/4,356-1=0,14

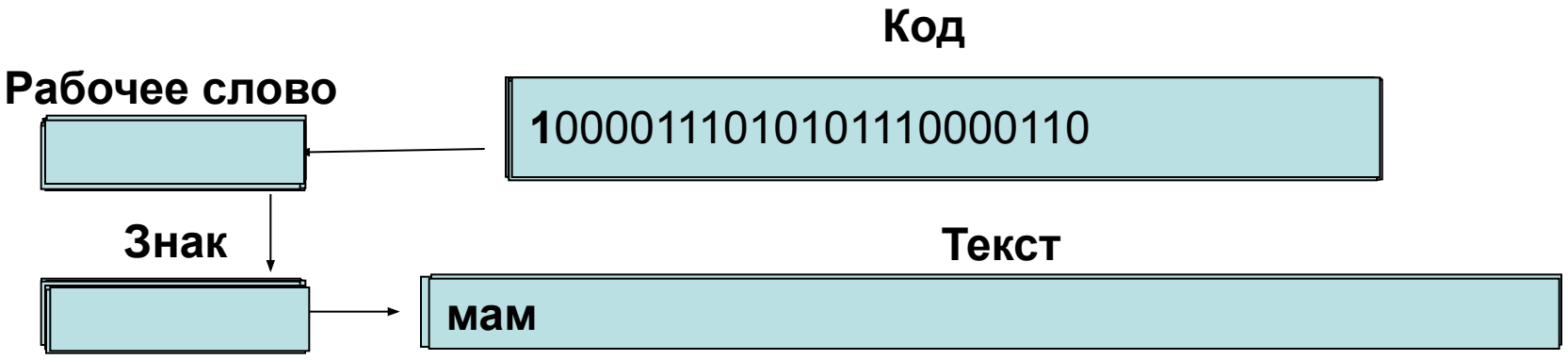
Префиксные коды. Условие Фано.

Неравномерный код может быть однозначно декодирован, если никакой из кодов знаков не совпадает с началом (префиксом) любого другого кода знака.

Например, если некоторому знаку выделен код 110, то недопустимы такие коды:
1, 11 ─ совпадают с начальными битами 110;
1100, 1101, 11011 и т.д. ─ их начало совпадает с 110.

Пример декодирования кода, подчиняющегося условию Фано:

а	л	м	р	у	ы
10	**010**	**00**	**11**	**0110**	**0111**



Префиксный код Шеннона - Фано

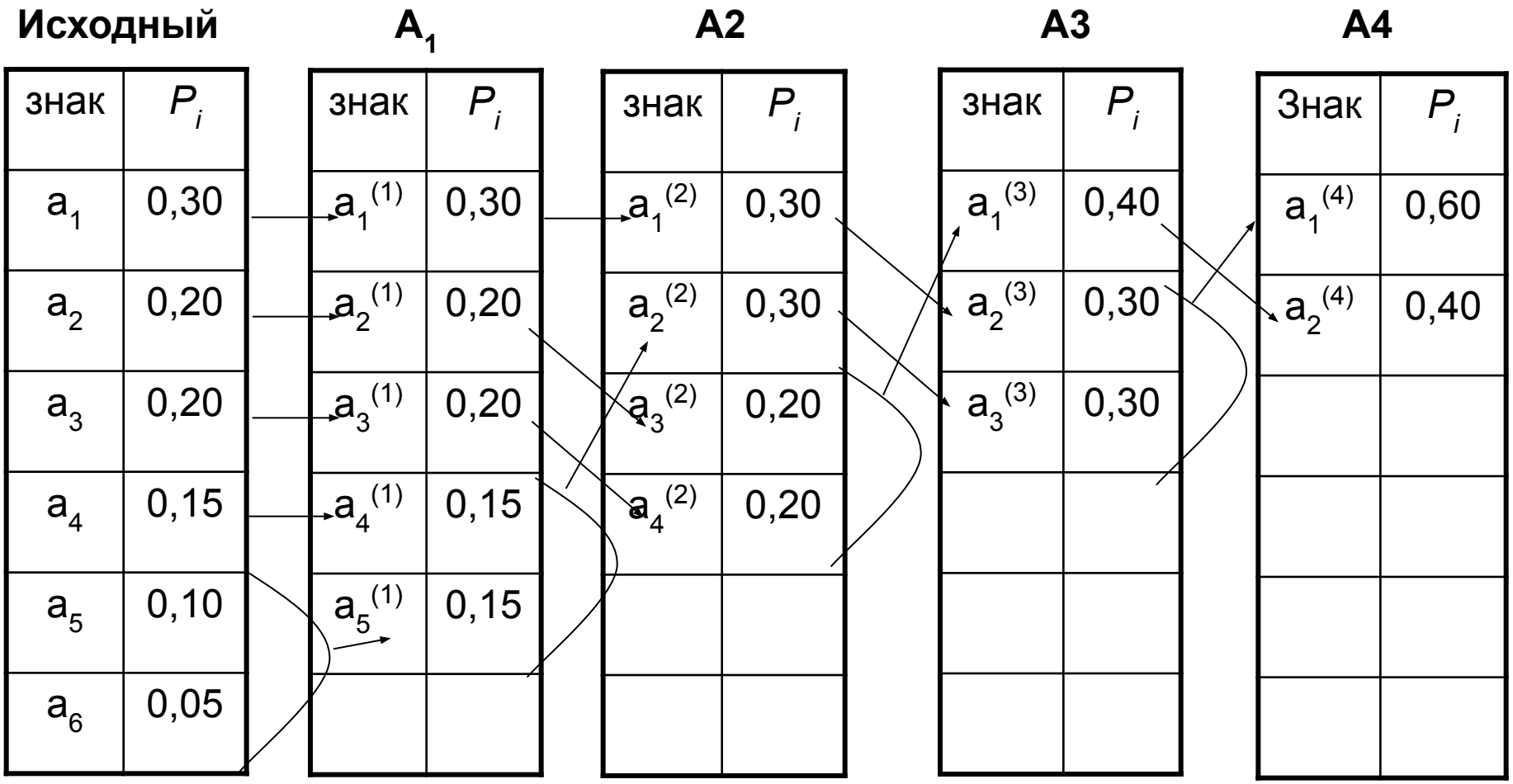
знак	P_i		
a_1	0,30	0,5	**00**
a_2	0,20		**01**
a_3	0,20		**10**
a_4	0,15	0,5	**110**
a_5	0,10		**1110**
a_6	0,05		**1111**

К(А,2)=2,45; I_1(А)=2,390; Q(А,2)=0,0249; *p*(0)=0,35; *p*(1)=0,65;

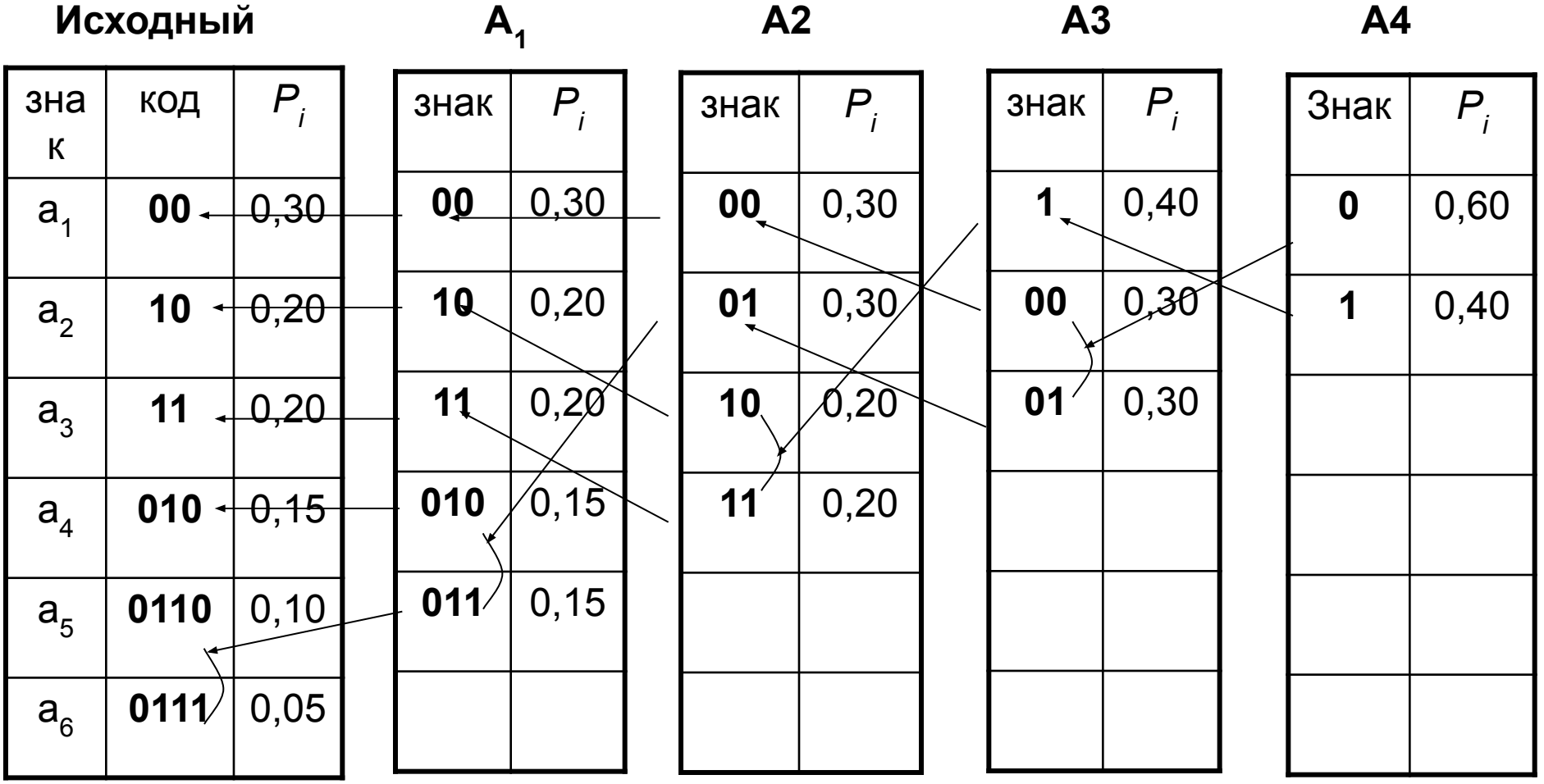
Применение способа Шеннона - Фано к русскому алфавиту дает Q(R,2)=0,0147

Префиксный код Хаффмана

Прямой ход



Обратный ход построения кода Хаффмана



К(А,2)=2,45; I(A)=2,390; Q(A,2)=0,0249; *p*(0)=0,47;*p*(1)=0,53; Q(R,2)=0,009

Код Хаффмана для русского алфавита

Буква	Частота	Код	Длина	Буква	Частота	Код	Длина
Пробел	0,174	000	3	я	0,018	001101	6
о	0,090	111	3	ы	0,016	010110	6
е	0,072	0100	4	з	0,016	010111	6
а	0,062	0110	4	ь,ъ	0,014	100001	6
и	0,062	0111	4	б	0,014	101100	6
т	0,053	1001	4	г	0,013	101101	6
н	0,053	1010	4	ч	0,012	110011	6
с	0,045	1101	4	й	0,010	0011001	7
р	0,040	00101	5	х	0,009	1000000	7
в	0,038	00111	5	ж	0,007	1000001	7
л	0,035	01010	5	ю	0,006	1100101	7
к	0,028	10001	5	ш	0,006	00110000	8
м	0,026	10111	5	ц	0,004	11001000	8
д	0,025	11000	5	щ	0,003	11001001	8
п	0,023	001000	6	э	0,003	001100010	9
у	0,021	001001	6	ф	0,002	001100011	9

Блочное кодирование

Коды, в которых кодовая цепочка ставится в соответствие последовательности из нескольких знаков первичного алфавита, называются ***блочными***

Пример блочного кодирования для алфавита {a,b}

Алфавитное кодирование

a	**0,75**	**0**
b	**0,25**	**1**

I(A)=0,811; K(A,2)=1; Q(A,2)=0,233

Блочное кодирование

aa	**0,5625**	**0**
ab	**0,1875**	**11**
ba	**0,1875**	**100**
bb	**0,0625**	**101**

I(A)=0,811, K(A,2)=0,844; Q(A,2)=0,04

Пусть словарный запас некоторого языка составляет А составляет 200000 слов. При равномерном двоичном кодировании слов $\mathbf{K(A,2) \geq \log_2 200000}$; **K(A,2)=18.** Средняя длина русских слов считается равной **6,3** знака (вместе с пробелом между словами). Таким образом, среднее количество информации составляет **2,85 бит на знак**. При равномерном алфавитном кодировании ─ **5 бит на знак**.

несет $I0(A) = log2n = 5$ бит информации и

несет $I0(A) = log2n = 5$ бит информации и

При использовании *объемного способа* измерения количество информации $I_{об}$, которое связано с представленным в кодировке ASCII одним знаком первичного алфавита, равно 8 битам или 1 байту

Выраженное в байтах количество информации, полученное с представленным в кодировке ASCII текстовым сообщением, численно равно количеству знаков, из которых состоит сообщение.

Очевидно, что количество информации на один знак, полученное с учётом различных вероятностей появления знаков , не может превышать количества информации, рассчитанного объемным способом, который базирует на предположении о *равновероятном* появлении знаков в сообщениях.

То есть измеренное объёмным способом количество информации определяет верхнюю границу для количества информации, рассчитываемого точным вероятностным способом:

$$I_1(A) \leq I_{об}$$