# Введение в алгоритмы и программирование

## Основы алгоритмизации

Что делает программист? Кажется, странный вопрос — программист пишет код. Хорошо. Что должен делать этот код? Тоже вроде бы понятно — код должен решать некую задачу. Например, решать квадратное уравнение, или анализировать курс акций, или давать возможность игровому персонажу бежать по экрану и собирать монеты. То есть код (он же — программа) должен решать задачу. Отлично. Но как же написать такой код, чтобы он точно решал задачу? Что должно быть в основе нашей программы?

Когда писатель собирается написать книгу, он в первую очередь решает задачу — рассказать историю. Как будет строиться эта история, хороший писатель знает. Он знает, как интересно рассказать историю, чтобы читатель все понимал, сопереживал. Одним словом, он пользуется накопленным знаниями из истории литературы и собственным опытом. Плохой писатель, будет путаться, перескакивать с одного на другое, занудно описывать неинтересное, в общем напишет плохую книгу.

Программист тоже пишет. Пишет код. И его код не рассказывает историю, а «реализует» алгоритм действий. Алгоритм. Вот. Мы подошли к одному из фундаментальных терминов в it сфере. Что же такое алгоритм? Можно ли программировать без этих самых алгоритмов? Можно ли сравнивать какой алгоритм лучше, какой хуже? На эти и другие вопросы нам необходимо будет получить ответы.

Сейчас же хотелось отметить вот что — если проводить аналогию с книгой — то именно алгоритмы — это ключевая идея, ясность мысли и связность повествования. Ведь если писатель действительно имеет отличную историю и умеет талантливо ее рассказать, то неважно на каком языке он напишет свою книгу. Ее все равно будет интересно читать. Переводчики ее переведут на другие языки, и она не утратит интересность.

Также и в программировании — если вы понимаете построение алгоритмов, их качества, возможности, требования; если вы знаете и умеете ими пользоваться, а также создавать на их основе новые, то язык описания алгоритмов — любой язык программирования — не так уж и важен. Именно поэтому, начало обучения программированию мы начнем со знакомством с алгоритмами.

### Небольшая историческая справка.

Термин **«алгоритм» (algorithm)** с одной стороны достаточно молод — в словарях 50-60 годов XX века определения этого слова нет, с другой его история уходит в глубь веков — приблизительно с 825 года (только тогда это слово писалось как algorism).

Итак, приблизительно в 825 году увидели свет несколько персидских трудов по математике. Их автором был выдающийся ученый того времени Абу Абд Аллах Мухаммед ибн Муса аль-Хорезми (что буквально означало: Отец Абдулы, Мухаммед, сын Мусы, родившийся в Хорезми). Именно его имя после перевода (транскрипции) на латинский язык приобрело написание «algorismi».

В одном трактате он сформулировал правила для выполнения арифметических вычислений над десятичными числами (порядок определенных действий). Другой назывался «Китаб аль-джебр валь-мукабала» и от его транскрипции появилось название раздела математики «алгебра».

Долгое время понятие алгоритм применялось к правилам арифметических вычислений, а затем и к различным наборам правил и инструкций, позволяющим получать искомые величины из исходных данных.

Типичным примером такого правила можно указать знаменитый Алгоритм Евклида, который позволяет находить НОД (наибольший общий делитель) двух натуральных чисел.

Есть два натуральных числа *a* и *b*, причем a больше или равен *b*.

Разделим с остатком число *a* на *b*. Остаток равен *r.*

**Если** *r* равен нулю,

**То** *b* это искомый НОД. Завершаем вычисление.

**Иначе** за *a* принимаем *b*, за *b* принимаем *r*

Продолжаем деление с остатком.

Например, a = 585 и b = 135.

585 = 135 × 4 + 45; r = 45; a = 135, b = 45

135 = 45 × 3; r = 0;

НОД = b = 45;

Обратите внимание, на то, что в этом алгоритме четко указаны все шаги и порядок действий, и этот порядок очень важен! Давайте посмотрим, что если в инструкции «Иначе за *a* принимаем *b*, за *b* принимаем *r*» поменять местами перестановки: «за *b* принимаем *r*, за *a* принимаем *b*», тогда в нашем примере мы сначала положили бы в *b* число 45, а потом в *a*… опять 45, потому что это сейчас значение *b*!

### Алгоритмы

На самом деле, как это ни удивительно, но нет строгого определения у понятия «Алгоритм». Однако на интуитивном уровне, так сказать, исходя из здравого смысла и житейского опыта, можно считать, что алгоритм — это некое описание вычислительного процесса.

Вот какое определение дается в Математическом энциклопедическом словаре:

Алгоритм — точное предписание, которое задает вычислительный процесс (называемый в этом случае алгоритмическим), начинающийся с произвольного исходного данного (из некоторой совокупности возможных для данного Алгоритма исходных данных) и направленный на получение полностью определяемого этим исходным данным результата. (<https://rus-math.slovaronline.com/169-АЛГОРИТМ>)

Мы с вами возьмем в качестве рабочего определения понятия «алгоритм», текст от Международной организации по стандартизации:

Алгоритм (Algorithm) — конечный набор предписаний, определяющий решение задачи посредством конечного количества операций.

Теоретические аспекты изучения алгоритмов (в математике — Теория алгоритмов), мы с вами обсудим позже. В частности, нас будет интересовать оценки эффективности алгоритмов. Сейчас же давайте определим основные и обязательные свойства алгоритмов.

#### 1. Дискретность.

Алгоритм состоит из набора ОТДЕЛЬНЫХ ЭЛЕМЕНТАРНЫХ команд и операций. В алгоритме нельзя сказать — *найти корни квадратного уравнения*. Нет, нам надо по ЭЛЕМЕНТАРНЫМ шагам описать как найти эти корни.

#### 2. Определенность (детерминированность)

Каждая команда или операция в алгоритме должны быть ясно и однозначно определены, и закончены. Мы не можем говорить «компьютеру» — пойди туда, не знаю куда, сделай то, не знаю, что.

#### 3. Конечность

Алгоритм должен всегда приводить к конечному результату. Даже если алгоритм реализует так называемый бесконечный цикл — всегда должна быть возможность из него «выпрыгнуть».

#### 4. Результативность и правильность

Конечный результат алгоритма, должен быть корректным, при корректных исходных данных. Невозможно пользоваться алгоритмом, если каждое его использование может приводить или к неправильным, или неопределенным результатам.

#### 5. Массовость

Алгоритм, разработанный для определенного класса задач, должен решать все подобные задачи, при любых корректных входных данных. Например, если мы разработали алгоритм поиска, то он должен уметь искать книги в каталоге, товар в прайс-листе, автомобиль в базе данных и т.п.

#### Порядок выполнения алгоритма:

* Все команды и операции в алгоритме исполняются строго в порядке их записи
* Нельзя менять местами никакие два действия алгоритма
* Нельзя не завершив очередного действия перейти к следующему

### Способы представления (описания) алгоритмов.

Существует множество способов представления алгоритмов, для их массового распространения и использования.

#### Естественным образом.

Изложение алгоритма в словесной форме на обычном человеческом языке. Например, различные инструкции по использованию бытовой техники, кулинарные рецепты и т.п.

Такая запись в словесной форме обладает рядом заметных недостатков:

* строго не формализуема;
* страдает многословностью записей;
* допускает неоднозначность толкования отдельных действий.

Эту форму логично использовать на начальных этапах разработки алгоритма.

Прежде чем алгоритм будет преобразован из естественного языкового представления в описание на одном из формальных языков программирования, его для общего понимания логики записывают в виде псевдокода или представляют в графическом виде.

**Словесное (естественное) описание Алгоритма Евклида:**

Чтобы найти НОД двух чисел, Большее число разделите на Меньшее с Остатком. Если Остаток равен нулю, то Меньшее число и есть НОД, иначе примите за Большее число нынешнее Меньшее, а за Меньшее текущий Остаток. Повторяйте деление с остатком.

#### Псевдокод.

Псевдокод является неким промежуточным состоянием описания алгоритма. С одной стороны, он все еще понятен носителю обычного естественного языка, с другой — обладает набором формальных правил, которые обычно повторяют правила реальных языков программирования.

**Алгоритм Евклида на псевдокоде:**

**алг** НОД

**арг** a, b

**рез** b

**нач**

**ввод** a, b

r = b

**пока** r ≠ 0

**нц**

r = a mod b

**если** r ≠ 0

**то**

a = b

b = r

**кц**

**вывод** b

**кон**

Основное назначение использования псевдокода — сделать лучшим восприятие и как следствие — понимание алгоритма человеком, по сравнению с исходным программным кодом. Обычно псевдокод используется на начальных этапах проектирования информационных систем (обычных программ), а также в публикациях и в учебных материалах. В псевдокоде допускается использовать фразы из естественного языка (русского, английского и т.п.), а также использовать общепринятую математическую нотацию (например, x2 вместо x^2).

mod — «деление по модулю», или проще говоря — остаток от целочисленного деления

Графические способы описания алгоритма обычно представлены в различных статьях и учебниках в виде «блок-схем» и «диаграмм Насси-Шнейдермана»

#### Блок-схемы

Графическое представление алгоритмов, во-первых, создает единый и понятный формат проектирования, который может быть «прочитан» специалистами разных языковых групп и специализаций, во-вторых, он представляет алгоритм в структурированном виде, в котором очень удобно проследить логику всех действий.

Существует стандарт для блок-схем. Мы не будем вдаваться в его тонкости и опишем основные блоки.

При начертании отдельных блоков, рекомендуется следовать основным принципам выбора размеров этих блоков: w, h. Причем выбирают определенную ширину w (например, 15, 20, 30 мм), а высоту h высчитывают из пропорции: 3w = 2h. Имейте в виду — это в общем случае только рекомендация, но практика показывает, что в таком виде блок-схема выглядит наиболее аккуратно и как следствие, наиболее удобна для чтения и понимания.

**Блок «Действие»**

Блок отображает определенное линейное действие, операцию, выражение (можно указывать несколько действий)

**Блок «Данные» («Ввод-вывод»)**

Данный блок указывает на осуществление ввода данных извне или отображение данных на любом периферийном устройстве (монитор, принтер). Точные источники или получатели данных не указываются.

**Блок «Предопределенный процесс» («Подпрограмма», «Функция»)**

Вызов иного процесса, который определен в другом месте. Аналог в программировании, вызов на исполнение другой подпрограммы.

**Блок «Условие»**

Этот блок показывает на разветвление в алгоритме. Имеет один вход, и в зависимости от результата действий, описанных внутри блока, осуществляет перенаправление на альтернативные выходы из этого блока.

**Блок «Цикл»**

Блок описывает так называемый цикл «с известным количеством повторов»

В блоках указываются начальные значения «счетчика» цикла, условие продолжения цикла, а также приращение счетчика на каждой итерации (каждом повторе) цикла.

**Блок «Ограничитель» («Начало-конец»)**

Блок указывает на вход в программу и выход из программы. Выход может быть естественным (завершение алгоритма) или, например, из-за получения ошибки (исключительной ситуации) в процессе исполнения алгоритма.

**Блок «Соединитель внутренний»**

Блок, помогающий распределить блок схему на одной странице.

**Блок «Соединитель внешний»**

Блок, помогающий распределить блок схему на разных страницах

**Блок схема алгоритма Евклида:**



Блок схемы можно назвать структурированным предшественником более сложного и более объемного языка графических диаграмм UML, с которым мы будем знакомиться гораздо позже.

С точки зрения программирования, конечно, вряд ли вы будете строить блок схемы для каждой задачи, однако на основании опыта очень многих программистов, можно сказать, что в сложных структурных алгоритмах и их комбинаций, построение блок схемы позволяет четко выстроить логику будущего программного кода.

Заметим, что блок схемы не единственный вариант графической записи алгоритмов. Как вариант отметим достаточно известную диаграммы Насси-Шнейдермана.

повторять

действие 1

истина

ложь

условие

действие

действие

действие 1

действие 2

действие 3

### Алгоритмические конструкции

Итак, мы получили начальные сведения о СВОЙСТВАХ алгоритмов и способах ОПИСАНИЯ алгоритмов. Настало время узнать, а собственно, на каких основных конструкциях строятся алгоритмы. В разной литературе количество таких конструкций может различаться, но всегда и везде будут указаны и описаны три основных:

* линейная
* условная
* циклическая

Прежде чем приступить к их детальному рассмотрению, давайте определим понятие **«простой операции» (простой команды)** — это элементарная и неделимая единица оперирования данными. Например, сложение двух чисел это простая операция, а сложение трех чисел — нет.

**«Составная операция»** — это композиция (соединение, объединение) нескольких простых операций. Сложение трех чисел это композиция двух простых сложений.

#### Линейная алгоритмическая конструкция (структура Следование)

Это набор однократно исполняемых, следующих строго друг за другом простых или составных операций.

Элементарный пример — приготовление кулинарного рецепта. Обычно в них последовательно описывается процесс обработки, смешивания и приготовления продуктов.

Алгоритм решения линейного уравнения:

**нач**

**ввод** a, b

x = -b / a

**вывод** x

**кон**



Надо отметить, что обычно, при проектировании информационной системы (то есть любого программного кода), изначально вся задача описывается линейной структурой Следования, и уже потом каждый укрупненный блок «раскрывается» на подробное описание частей алгоритма.

#### Разветвляющаяся алгоритмическая конструкция (структура Ветвление, Условная)

Эта конструкция используется в том случае, когда в зависимости от выполнения некоторого условия выбирается одна из двух последовательностей действий.

Например, действия законопослушного пешехода на перекрестке: если горит зеленый свет, то следует ИДТИ, если нет, то следует ЖДАТЬ.

Структура Ветвление может быть в полной или сокращенной форме.

**если**

**то**

**иначе**



**если**

**то**



Как видно из описаний на блок-схемах и псевдокоде, алгоритмическая структура Ветвление (очень часто можно встретить название — Условная Операция), так вот, эта структура может быть сокращенной или полной.

Сокращенное условие дополняет действия алгоритма, в случае выполнения некоторых условий и не предполагает альтернативных действий, если условие не выполняется. Соответственно полная форма — это разветвление на две ветки операций. Эти ветви альтернативные и взаимно не пересекаются.

Простой пример сокращенной формы, например при походе в магазин, вас просят — если будет что-то, то купи это. И все. То есть, если этого не будет и покупать взамен ничего не надо.

А вот если вас отправляют в магазин купить хлеб, то могут сказать — купи пшеничного, а если не будет — купи ржаного. (Вариант, когда хлеба нет никакого, здесь не рассматривается 😊)

По-прежнему, важно понимать, что внутри из каждых веток могут быть составные команды, включающие в себя все варианты алгоритмических структур.

Алгоритм решения квадратного уравнения, который мы должны помнить еще со школы (или вспомнить прямо сейчас):

Для получения корней этого уравнения, необходимо вычислить Дискриминант:

А, дальше, в зависимости от значения дискриминанта мы вычисляем корни уравнения:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  | *корней нет* |

Алгоритм решения на псевдокоде:

**нач**

**ввод** a, b, c

D = b \* b – 4 \* a \* c

**если** D >= 0

**то если** D == 0

**то**

x = -b / 2 / a

**вывод** x

**иначе**

x1 = (-b + sqrt(D)) / 2 / a

x2 = (-b - sqrt(D)) / 2 / a

**вывод** x1, x2

**иначе**

**вывод** "Корней нет"

Кстати, возможен другой вариант (обратите внимание на выделенные условия):

**нач**

**ввод** a, b, c

D = b \* b – 4 \* a \* c

**если** D > 0

**то**

x1 = (-b + sqrt(D)) / 2 / a

x2 = (-b - sqrt(D)) / 2 / a

**вывод** x1, x2

**иначе если** D == 0

**то**

x = -b / 2 / a

**вывод** x

**иначе**

**вывод** "Корней нет"

Блок-схема алгоритма решения квадратного уравнения.



Блок-схему второго варианта, нарисуйте сами.

#### Циклическая конструкция. (Структура Цикл, Повторение)

В процессе выполнения задач, очень часто требуется повторять одни и те же действия. Количество повторов может быть известно, а может быть и нет, тогда идет повтор, пока не выполнится какое-то условие.

Например, для приготовления по рецепту, нам необходимо почистить картофель. В кастрюле будет лежать определенное количество клубней и именно такое количество повторов нам предстоит: повторяем N раз — берем нечищеный картофель, чистим, бросаем в воду. N — количество картофеля в кастрюле.

Или, допустим, необходимо забросить баскетбольный мяч в корзину. Здесь заранее нельзя сказать, сколько раз будет повторятся бросок. Пока не забросим. Количество повторов неизвестно.

Еще одно разделение циклических конструкций проходит по тому, когда мы проверяем условие повторения: перед тем как повторять, или уже после того, как повторили.



**выполнять**

**пока**

**пока**

**нц**

**кц**





Между этими видами циклов есть важнейшее принципиальное различие: тело цикла с постусловием, хотя бы один раз ВЫПОЛНИТСЯ ОБЯЗАТЕЛЬНО!

Как вы уже заметили, набор инструкций (операций), находящийся внутри цикла, называется **телом цикла**. Один проход по телу цикла, называется **итерацией**. Поэтому зачастую можно услышать, про алгоритм с применением цикла — итерационный алгоритм, циклический алгоритм.

Напоминаем, что в теле цикла могут быть Составные конструкции, а также конструкции Ветвления и даже другие Циклические конструкции (вложенные циклы, с которыми мы будем знакомиться позже).

Например, решим задачу получения суммы первых N натуральных чисел:

**Словесное описание**

Принять сумму за ноль. Принять счетчик цикла за единицу.

Пока прибавить к Сумме значение счетчика. Увеличить счетчик на единицу. Повторить.

Псевдокод:



**нач**

**ввод** N

S = 0, i = 1

**пока** i <= N

**нц**

S = S + i

i = i + 1

**кц**

**ввыод** S

При работе с циклами, необходимо соблюдать корректность изменение параметра цикла, для того чтобы быть уверенными, что однажды условие выполнится. Иначе может случится логическая ошибка, которая приведет к **«бесконечному» циклу** и как результат — «зависанию» программы. Хотя бывают случаи, когда бесконечные циклы являются осмысленным инструментом и применяются программистом для решения некоторых задач. Однако, мы также познакомимся с применением таких циклов позже.

### Переменная. Операция присваивания.

#### Переменная

При знакомстве с основами алгоритмов, мы использовали некоторые сущности, в которых хранились какие-то числовые значения. Для того чтобы мы могли их различать между собой, им давались разные названия: a, b, N, S, D, x, i и т.д.

Из курса школьной математики нам известно, что математическая переменная это некий объект, который может принимать некоторое значения из заданного множества значений (область определения переменной).

В программировании понятие **переменной**, близко к математическому. Но имеет свои особенности.

Итак, мы знаем, что современные компьютеры строятся на «фон Неймановской» архитектуре, а это значит, что и сама программа (исполняемый код), и данные, которыми она оперирует, располагаются в оперативной памяти. То есть переменные, которые мы будем использовать в программировании, располагаются в оперативной памяти. Напомним, что память делится на ячейки (размер ячейки 1 байт). То есть переменная будет занимать некоторое количество ячеек (байт) памяти, в которых и будет храниться некоторое числовое значение.

Именно целое числовое — потому что ничего другого компьютер хранить не умеет, а значит если нам понадобится хранить данные другого типа (например, символьные), то на самом деле в соответствующей переменной (а по факту в ячейках памяти) будет храниться число, которое будет «кодом» для данных другого типа. Например, вместо символа 'W' будет храниться его код — число 87.

В программировании переменные, как и в математике имеют имена — **идентификаторы**. И хотя математические переменные обычно называются буквально одной буквой (переменная ), то в программировании обычно переменным дают имена, имеющие смысл. Например count, array, amount, power и т.д.

В имени переменной могут быть символы латинского алфавита (в любом регистре), символы цифр и символ нижнего подчеркивания (\_). Начинаться имя переменной может с символа буквы или с подчеркивания.

Большинство языков программирования являются **регистро-зависимыми**, то есть различающие регистр в именах переменных. Скажем переменные count, Count, COUNT — это разные переменные в таких языках.

Также за десятилетия, программисты выработали некие правила написания идентификаторов переменных. Такие правила называются — **нотациями**.  
Например мы хотим создать переменную для хранения количества качественного товара (англ.: count of quality product)

Ниже, в таблице, приведены общеупотребительные нотации с примерами:

|  |  |  |  |
| --- | --- | --- | --- |
| Название нотации | Английское название | Примеры | Пояснения |
| Плоская нотация | flat case | countqualityproduct | Используется для односложных имен переменных, в наименованиях пакетов и сборок (например, в языке Java) |
| Кебаб (шашлычная) нотация | kebab case | count-quality-product | Используется обычно для URL адресов, для наименований стилей в CSS и тегов в HTML. Также используется в наименовании файлов. |
| Змеиная нотация | snake case | count\_quality\_product | Используется обычно в языках Python, PHP, Rust |
| Нотация Паскаля | pascal case | CountQualityProduct | Впервые нотация появилась в разработках на языке Pascal (Паскаль).  Используется в основных языках (C++, C#, Java, JavaScript и пр) для именования функций, методов и Пользовательских типов (классов, структур, перечислений) |
| Кэмел (верблюжья) нотация | сamel case | сountQualityProduct | Обычно используется в основных C-подобных языках для именования переменных и объектов (переменных классов и структур) |

#### Операция присваивания

Как следует из определения переменной, она предназначена для хранения каких-то значений. Значение переменной (**value**), в процессе выполнения программы обычно меняется. Для того чтобы поместить в переменную некоторое значение используют **операцию присваивания**.

Операция присваивания в большинстве языков обозначается символом равно '=' и будьте внимательны — в программировании это именно знак операции присваивания, а не операции сравнения на равенство.

Особенность операции присваивания таково, что, когда происходит присваивание переменной нового значения, предыдущие навсегда теряется. То есть, после того как вы присвоили переменной, например значение 10, то выяснить, что хранила переменная до этого будет теперь невозможно в принципе.

Происходит это в следствии так называемого «поверхностного копирования». Образно говоря, операционная система «создает» в неконтролируемой нами памяти «образ значения» — набор ноликов и единиц (битов), составляющих число присваиваемого значения, затем просто копирует эти биты по адресу регистров, в которых хранится наша переменная.

Можно отчасти сравнить этот процесс, как стирание со школьной доски старой надписи и нанесение новой, с той разницей, что операционная система не «стирает», а просто «заменяет» значения битов на новые.

В связи с этим возникает первая для нас с вами задача, имеющая важнейшее повсеместное применение:

Есть две переменных, допустим a и b, в этих переменных хранятся какие то значения (нам абсолютно не важно, какие именно, напомним, что алгоритм решения должен быть универсальным), так вот — необходимо осуществить обмен значений между этими переменными.

Например:

**алг** Обмен переменными

**арг**

a, b

**нач**

a = 10

b = 20

**вывод**(a, b)

// некие манипуляции с нашими переменными

**вывод**(a, b)

**кон**

Во время первого вывода, должно быть напечатано:

10, 20

Во время второго:

20, 10

Подумайте, какие пути решения, вы можете предложить?

Допустим мы напишем такой код:

a = b

b = a

Решит ли этот код нашу задачу? Очевидно, что нет: при первом присваивании переменной a будет присвоено новое значение 20, при этом старое значение 10 безвозвратно исчезнет. А значит при втором присваивании в переменную b будет присвоено новое значение a, а именно то же самое число 20.

Нетрудно убедиться, что если сначала в переменную b присвоить значение a, а потом в b значение a, то к положительному результату это точно также не приведет.

Обычно, для наглядной иллюстрации проблемы и нахождения очевидного решения, приводят пример с двумя стаканами:

В левом стакане допустим красная вода, в правом — синяя. Чтобы в левом оказалась синяя, а в правом красная, нам очевидно понадобится… Правильно! Третий стакан.

Итак, для решения нашей задачи обмена значениями между двумя переменными, нам необходимо ввести третью переменную:

temp = a

a = b

b = temp

Поздравляем. Нам стал известен первый фундаментальный алгоритм. И, поверьте, применять этот алгоритм, вам предстоит многократно.

Кстати. Если разобрать эту задачу только для переменных, которые хранят только целые числа, то мы можем найти еще один алгоритм обмена значениями между переменными:

a = a + b

b = a – b

a = a – b

Давайте проанализируем эти два алгоритма. Можем ли мы однозначно сказать, что какой-то из них лучше? Грамотный подход к ответу на такой вопрос, конечно должен включать уточнение: а по какому критерию мы сравниваем? Или — по каким критериям, если их несколько.

Ведь, например невозможно однозначно сказать, какой автомобиль лучше: спортивный или внедорожник. Если сравнивать по скорости, быстрее первый, если по проходимости и грузоподъемности, очевидно предпочтительнее — второй.

Так же и с алгоритмами — необходимо обозначить критерии оценки. И алгоритмы сравнивают по количеству выполняемых операций (вычислительная сложность) и по объему памяти, необходимой для выполнения алгоритма.

И вот получается, что по первому критерию (по количеству операций) оптимальнее первый алгоритм — в нем только три операции присваивания. А по второму критерию (объем необходимой памяти) оптимальнее второй — в нем, конечно, шесть операций, но зато памяти требуется только для двух переменных, в то время как в первом необходима дополнительная память под третью переменную.

К счастью ли, к сожалению, но в повседневной работе, программисту часто приходится выбирать по какому критерию оптимизировать свою программу — потому что выиграть и по времени исполнения, и по объему памяти — практически невозможно 😊.

Очень часто операция присваивания осуществляет «модификацию» (то есть изменение) переменной. Выглядит это, например, так:

a = a + 10

В данном случае, сначала выполняются все операции справа от операции присваивания, а именно: извлекается текущее значение переменной a, к этому значению присваивается 10, и получившийся результат опять помещается в переменную a.

То есть говоря об операции присваивания, можно отметить ее «низкий приоритет», то есть очередность выполнения: чаще всего она выполняется самой последней в большом, составном выражении.

### Вспомогательные алгоритмы. Подпрограммы. Процедуры и функции.

Мы рассмотрели три вида алгоритмических конструкций: линейная, ветвления, циклическая. И хотя этих трех конструкций достаточно для реализации ЛЮБОГО алгоритма, стоит отметить еще одну. Это Подпрограмма.

Подпрограмма (в современных языках называется функцией и процедурой), это оформленный специальным образом алгоритм (комбинация алгоритмов) для многократных вызовов. Подпрограмма может принимать на входе (при вызове) некоторые параметры (аргументы)

При этом подпрограмма создается таким образом, что может принимать различные входные аргументы (параметры), что добавляет универсальности подпрограмме.

Подпрограммы могут возвращать какое-то значение, а может просто выполнять некоторые действия, без возврата конечного значения.

Например, мы можем оформить в отдельную подпрограмму возведение в степень числа.

Подпрограмма power(x, n)



Теперь можно вызывать эту подпрограмму в других алгоритмах. Например, при вычислении площади круга:



В некоторых языках программирования (Pascal, Delphi) подпрограммы могут называться функциями, в том случае если возвращается некоторое значение, а также процедурами — если подпрограмма ничего не возвращает.

Однако в большинстве C-подобных языках программирования (C, C++, C#, Java, JavaScript, Php и т.д.) все подпрограммы называются функциями.

### Тестирование ПО

В свое время вы познакомитесь с тестированием, гораздо полнее, сейчас же стоит отметить важные моменты.

Как гласит один из «законов Мерфи для программистов» — каждая строчка кода увеличивает процент ошибок в нем. Практически не бывает программного продукта без ошибок.

Все ошибки можно укрупненно поделить на три категории:

* Синтаксические (syntax error) — обычно выявляются на моменте трансляции программы
* Ошибки выполнения (runtime error) — возникают в процессе работы программы, обычно приводя к некорректному ее завершению.
* Алгоритмические ошибки (logical error) — наиболее трудно выявляемые ошибки. Обычно не приводят к фатальному завершению программы, однако выдают в итоге некорректные результаты.

Если ошибки первых двух категорий выявляются или компилятором или средой исполнения, то ошибки третьей категории выявить на стадии написание программы, можно только тестированием.

**Тестирование** — процесс проверки работоспособности программы путем ввода в неё различных, даже намеренно ошибочных данных, и последующей контрольной проверке выводимого результата.

Если программа работает правильно с одними наборами исходных данных, и неправильно с другими, то это свидетельствует о наличии алгоритмической ошибки. Алгоритмические ошибки иногда называют логическими, обычно они связаны с неверной реализацией алгоритма программы: вместо "+" ошибочно поставили "-", вместо "/" - "\*", вместо деления значения на 0,01 разделили на 0,001 и т.п. Такие ошибки обычно не обнаруживаются во время компиляции, программа нормально запускается, работает, а при анализе выводимого результата выясняется, что он неверный. При этом компилятор не укажет программисту на ошибку - чтобы найти и устранить её, приходится анализировать код, пошагово "прокручивать" его выполнение, следя за результатом. Такой процесс называется отладкой.

**Отладка** — процесс поиска и устранения ошибок, чаще алгоритмических. Хотя отладчик позволяет справиться и с ошибками времени выполнения, которые не обнаруживаются явно.

Программные ошибки на программистском сленге называют багами (англ. bug - жук), а программы отладки кода - дебаггерами (англ. debugger - отладчик). Сам процесс отладки — дебаггинг (debugging)

В свое время, вы познакомитесь с основами тестирования.