Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА
Институт Кибернетики
Кафедра Промышленной Информатики

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Тема лекции «Программное управление – основа вычислительного процесса. Сложность алгоритмов. Программное обеспечение»

Лектор Каширская Елизавета Натановна (к.т.н., доцент, ФГБОУ ВО "МИРЭА - Российский технологический университет") e-mail: liza.kashirskaya@gmail.com

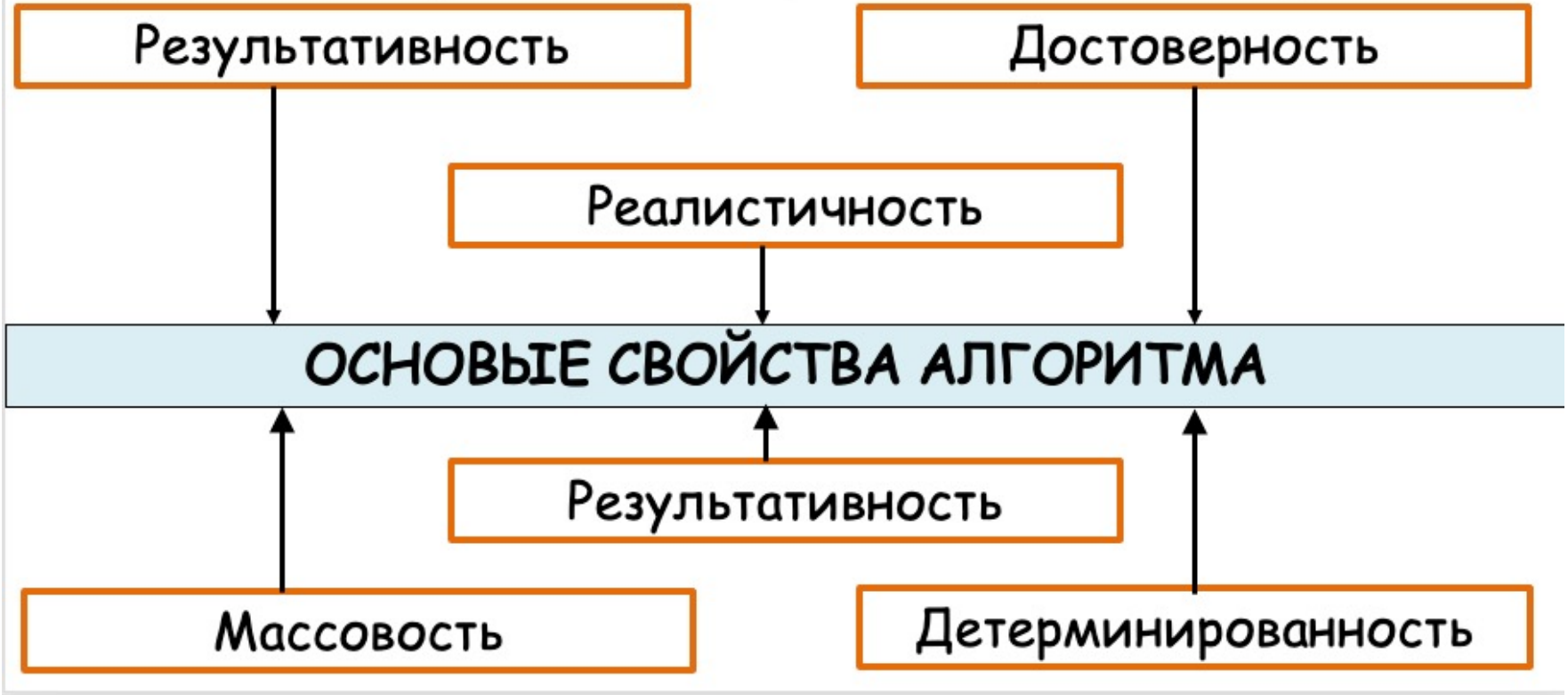
Лекция № 5

АЛГОРИТМ


АЛГОРИТМ
«Создать свою программу или игру может каждый уже сегодня!»


Мухаммед аль-Хорезми

АЛГОРИТМ — это точно определенная последовательность действий, которые необходимо выполнить над исходной информацией, чтобы получить решение задачи.



Свойства алгоритма

РЕЗУЛЬТАТИВНОСТЬ — алгоритм должен давать конкретное конструктивное решение, а не указывать на возможность решения вообще.

ДОСТОВЕРНОСТЬ — алгоритм должен соответствовать сущности задачи и формировать верные, не допускающие неоднозначного толкования решения.

РЕАЛИСТИЧНОСТЬ — возможность реализации алгоритма при заданных ограничениях: временных, программных, аппаратных.

МАССОВОСТЬ — алгоритм должен быть воспроизводимым, пригодным для решения всех задач определенного класса на всем множестве допустимых значений исходных данных.

ДЕТЕРМИНИРОВАННОСТЬ (определенность) — алгоритм должен содержать набор точных и понятных указаний, не допускающих неоднозначного толкования.

Способы записи алгоритмов

СЛОВЕСНЫЙ способ записи содержит последовательные этапы алгоритма и описывается в произвольной форме на естественном языке;

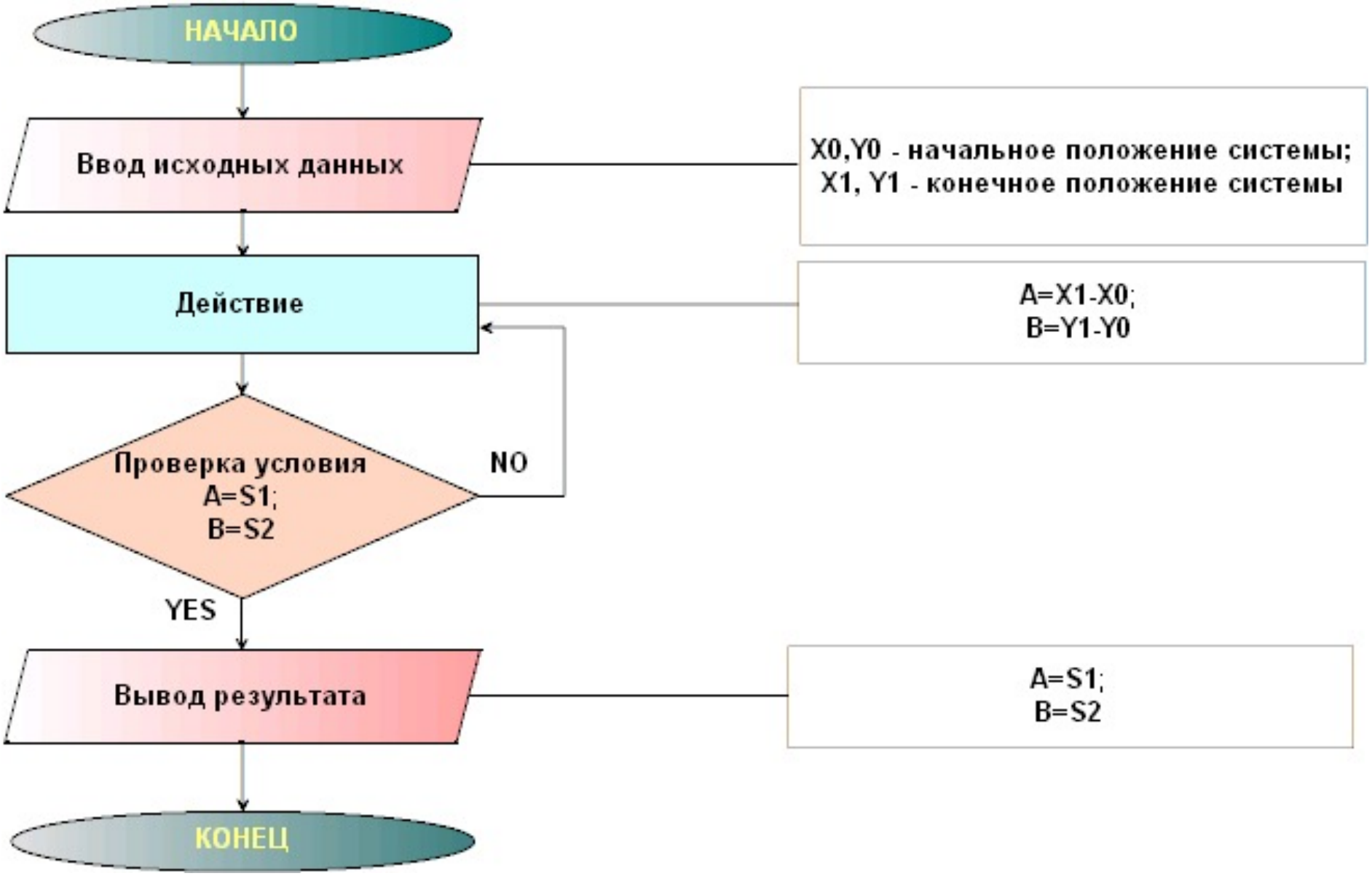
ФОРМУЛЬНЫЙ способ основан на строго формализованном аналитическом задании необходимых для исполнения действий;

ТАБЛИЧНЫЙ способ подразумевает отображение алгоритма в виде таблиц, использующих аппарат реляционного исчисления и алгебру логики для задания подлежащих исполнению взаимных связей;

ОПЕРАТОРНЫЙ способ базируется на использовании для отображения алгоритма условного набора специальных операторов: арифметических, логических, печати и т. д.;

ГРАФИЧЕСКОЕ отображение алгоритмов в виде блок-схем — самый распространенный способ.

Пример построения алгоритма



Пример построения алгоритма

Псевдокод

1. Ввод исходных данных:
   X0, Y0 -информация о начальном положении системы;

   X1, Y1-информация о конечном положении системы.

2. Вычисление приращения положения системы относительно друг друга:

   A=X1-X0;
   B=Y1-Y0

Если [Условие]
[A=S1; B=S2]

То [Действие1]
[S1=X1-X0; S2=Y1-Y0]

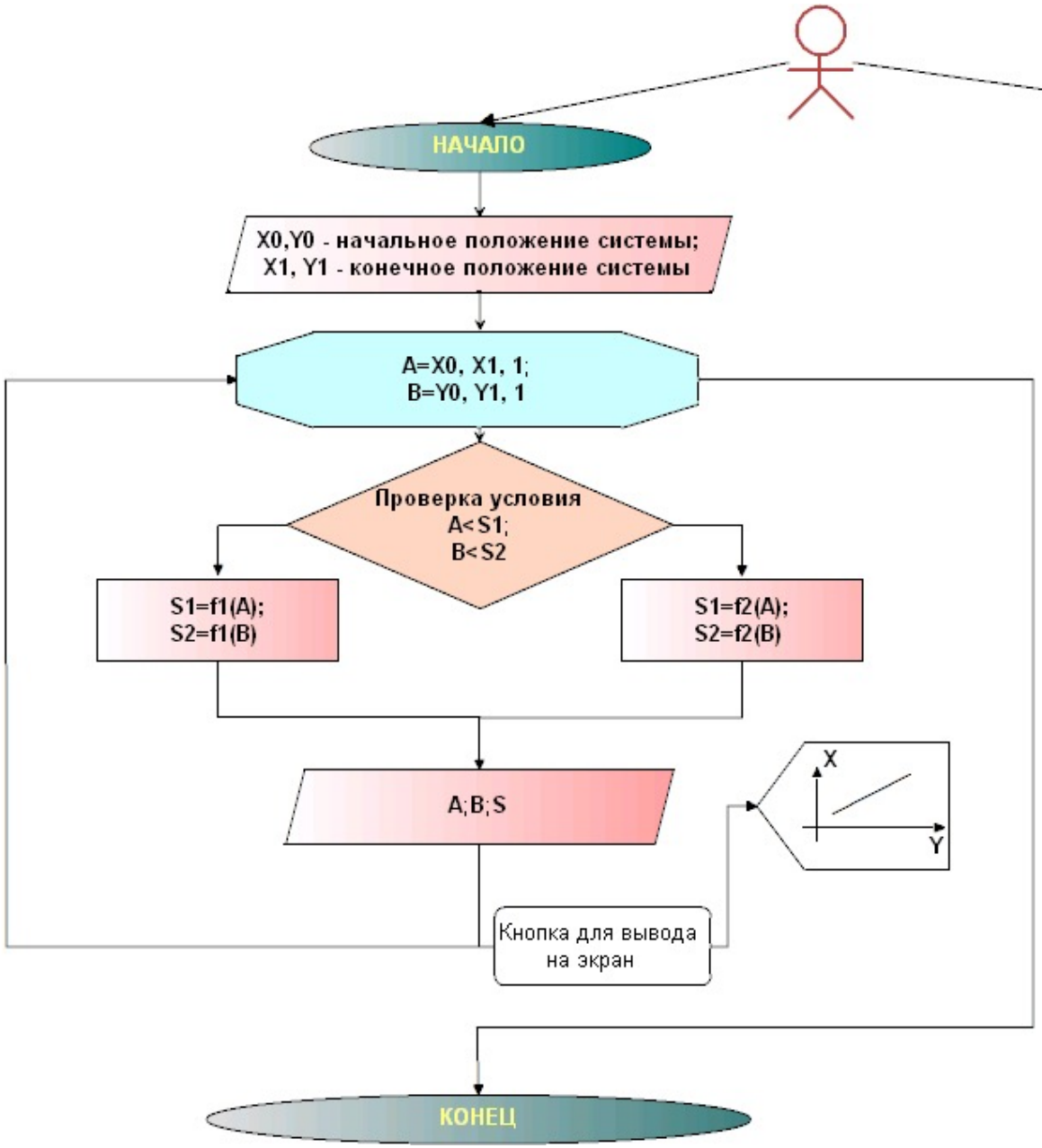
Иначе [Действие2]
[S1≠X1-X0; S2≠Y1-Y0]

Конец если

3. Вывод A, B

4. Конец

Пример построения алгоритма



Псевдокод

1.Ввод исходных данных:

X0, Y0 -информация о начальном положении системы;

X1, Y1-информация о конечном положении системы.

2. Начало цикла

Для

A=X0, X1, 1; B=Y0, Y1, 1

повторить

а) проверка условия:

Если

[A<S1; B<S2]

То

S1=f1(A);S2=f1(B)

Иначе

S1=f2(A);S2=f2(B)

б) вывод

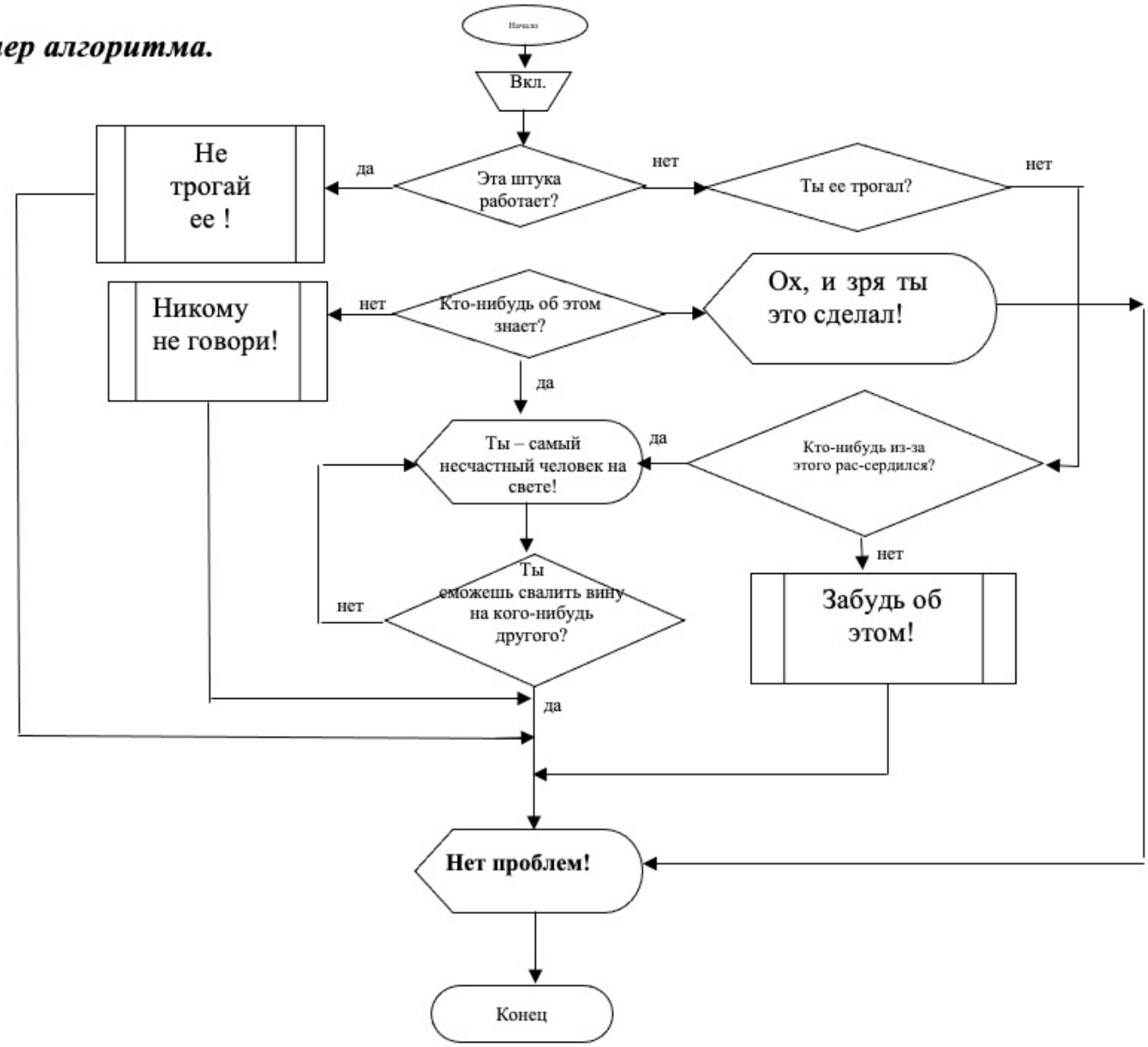
A; B; S

Конец цикла

3. Конец

Алгоритм решения проблем

Пример алгоритма.



СЛОЖНОСТЬ АЛГОРИТМОВ

Многие современные программисты, пишущие классные и широко распространённые программы, имеют крайне смутное представление о теоретической информатике. Это не мешает им оставаться прекрасными творческими специалистами, и мы благодарны за то, что они создают. Тем не менее, знание теории тоже имеет свои преимущества и может оказаться весьма полезным. Эта часть лекции посвящена анализу сложности алгоритмов. Как специалист, работавший и в области академической науки, и над созданием коммерческого программного обеспечения, я считаю этот инструмент по-настоящему полезными на практике. Надеюсь, что после сегодняшней лекции вы сможете применить его к собственному коду, чтобы сделать его ещё лучше.

Мы знаем, что существуют инструменты, измеряющие, насколько быстро работает код. Это программы, называемые профайлерами (profilers), которые определяют время выполнения в миллисекундах, помогая нам выявлять узкие места и оптимизировать их. Но, хотя это и полезный инструмент, он не имеет отношения к сложности алгоритмов. **Сложность алгоритма — это то, что основывается на сравнении алгоритмов на идеальном уровне, на котором игнорируются низкоуровневые детали** вроде реализации языка программирования, «железа», на котором запущена программа, или набора команд в данном процессоре. Подсчёт миллисекунд тут мало поможет. Вполне может оказаться, что плохой алгоритм, написанный на низкоуровневом языке (например, ассемблере), будет намного быстрее, чем хороший алгоритм, написанный на языке программирования высокого уровня (например, на Python или C++). Так что пришло время определиться, что же такое «лучший алгоритм» на самом деле.

Алгоритм — это программа, которая представляет собой исключительно вычисление, без других часто выполняемых компьютером вещей — сетевых задач или пользовательского ввода-вывода. Анализ сложности позволяет нам узнать, насколько быстра эта программа, когда она совершает вычисления. Примерами чисто вычислительных операций могут послужить операции над числами с плавающей точкой (сложение и умножение), поиск заданного значения из находящейся в ОЗУ базы данных, или определение игровым искусственным интеллектом движения своего персонажа таким образом, чтобы он передвигался только на короткое расстояние внутри игрового мира. Очевидно, что вычисления встречаются в компьютерных программах повсеместно.

Анализ сложности также позволяет нам объяснить, как будет вести себя алгоритм при возрастании входного потока данных. Если наш алгоритм выполняется одну секунду при 1000 элементах на входе, то как он себя поведёт, если мы удвоим это значение? Будет работать также быстро, в полтора раза быстрее или в четыре раза медленнее? В практике программирования такие предсказания крайне важны. Например, если мы создали алгоритм для web-приложения, работающего с тысячей пользователей, и измерили его время выполнения, то, используя анализ сложности, мы получим весьма неплохое представление о том, что случится, когда число пользователей возрастёт до двух тысяч. Для соревнований по построению алгоритмов анализ сложности также даст нам понимание того, как долго будет выполняться наш код на наибольшем из тестов для проверки его правильности. Так что если мы определим общее поведение нашей программы на небольшом объёме входных данных, то сможем получить хорошее представление и о том, что будет с ней при больших потоках данных.

Давайте начнём с простого примера: поиска максимального элемента в массиве.

Пример. Подсчёт инструкций
Максимальный элемент массива можно найти с помощью простейшего отрывка кода.

```
int M = A[0];
for(int i = 1; i < n; ++i)
{
   if(A[i] >= M)
   {
      M = A[i];
   }
}
```

В процессе анализа данного кода имеет смысл разбить его на простые инструкции — задания, которые могут быть выполнены процессором тотчас же или близко к этому. Предположим, что наш процессор способен выполнять как единые инструкции следующие операции:

- присваивать значение переменной,
- находить значение конкретного элемента в массиве,
- сравнивать два значения,
- инкрементировать значение,
- складывать,
- вычитать,
- умножать,
- делить.

Мы будем полагать, что ветвление (выбор между **if** и **else** частями кода после вычисления if-условия) происходит мгновенно, и не будем учитывать эту инструкцию при подсчёте.

Для первой строки в приведенном коде:

int M = A[0];

требуются две инструкции:

- для поиска A[0] и
- для присвоения значения M (мы предполагаем, что **n** всегда как минимум равно 1).

Эти две инструкции будут требоваться алгоритму вне зависимости от величины n.

Инициализация цикла **for** тоже будет происходить постоянно, что даёт нам ещё две команды - присвоение и сравнение:

i = 0; i < n;

Всё это происходит до первого запуска **for**. После каждой новой итерации мы будем иметь на две инструкции больше: инкремент **i** и сравнение для проверки, не пора ли нам останавливать цикл:

++i; i < n;

Таким образом, если мы проигнорируем содержимое тела цикла, то количество инструкций у этого алгоритма **4 + 2*n*** — четыре на начало цикла **for** и по две на каждую итерацию, которых мы имеем **n** штук.

Теперь мы можем определить математическую функцию ***f*(n)** - такую, что, зная **n**, мы будем знать и необходимое алгоритму количество инструкций.

Для цикла **for** с пустым телом $f(n) = 4 + 2n$.

Анализ наиболее неблагоприятного случая

В теле цикла мы имеем операции поиска в массиве и сравнения, которые происходят всегда:

if (A[i] >= M) { ... }

Но тело **if** может запускаться, а может и нет, в зависимости от актуального значения из массива. Если произойдёт так, что **A[i] >= M**, то у нас запустятся две дополнительные команды - поиск в массиве и присваивание:

M = A[i]

Мы уже не можем определить ***f*(*n*)** так легко, потому что теперь количество инструкций зависит не только от **n**, но и от конкретных входных значений. Например, для **A = [1, 2, 3, 4]** программе потребуется больше команд, чем для **A = [4, 3, 2, 1].**

Анализ наиболее неблагоприятного случая

Когда мы анализируем алгоритмы, мы чаще всего рассматриваем наихудший сценарий. Каким он будет в нашем случае? Когда алгоритму потребуется больше всего инструкций до завершения? Ответ: когда массив упорядочен по возрастанию, как, например, **A = [1, 2, 3, 4].** Тогда **M** будет переприсваиваться каждый раз, что даст наибольшее количество команд. Теоретики имеют для этого причудливое название — анализ наиболее неблагоприятного случая, который является ничем иным, как просто рассмотрением максимально неудачного варианта. Таким образом, в наихудшем случае в теле цикла из нашего кода запускаются четыре инструкции, и мы имеем $f(n) = 4 + 2n + 4n = 6n + 4$.

С полученной выше функцией мы имеем весьма хорошее представление о том, насколько быстр наш алгоритм. Однако нам нет нужды постоянно заниматься таким утомительным занятием, как подсчёт команд в программе. Более того, количество инструкций у конкретного процессора, необходимое для реализации каждого положения из используемого языка программирования, зависит от компилятора этого языка и доступного процессору набора команд.

Ранее же мы говорили, что собираемся игнорировать условия такого рода. Поэтому сейчас мы пропустим нашу функцию f через «фильтр» для очищения её от незначительных деталей, на которые теоретики предпочитают не обращать внимания.

Наша функция **6*n* + 4** состоит из двух элементов: **6*n*** и **4.** При анализе сложности важность имеет только то, что происходит с функцией подсчёта инструкций при значительном возрастании ***n***. Это совпадает с предыдущей идеей «наихудшего сценария»: нам интересно поведение алгоритма, находящегося в «плохих условиях», когда он вынужден выполнять что-то трудное. Заметьте, что именно это по-настоящему полезно при сравнении алгоритмов. Если один из них побивает другой при большом входном потоке данных, то велика вероятность, что он останется быстрее и на лёгких, маленьких потоках. Вот почему **мы отбрасываем те элементы функции, которые при росте *n* возрастают медленно, и оставляем только те, что растут сильно**. Очевидно, что 4 останется 4 вне зависимости от значения **n**, а **6n** наоборот будет расти. Поэтому первое, что мы сделаем, — это отбросим 4 и оставим только ***f*(*n*) = 6*n*.**

Имеет смысл думать о **4** просто как о «константе инициализации».

Разным языкам программирования для настройки может потребоваться разное время. Например, Java сначала необходимо инициализировать свою виртуальную машину. А поскольку мы договорились игнорировать различия языков программирования, то попросту отбросим это значение.

Второй вещью, на которую можно не обращать внимания, является множитель перед **n**. Так что наша функция превращается в $f(n) = n$. Как вы можете заметить, это многое упрощает. Ещё раз, константный множитель имеет смысл отбрасывать, если мы думаем о различиях во времени компиляции разных языков программирования. Поиск в массиве для одного языка программирования может компилироваться совершенно иначе, чем для другого. Например, в Си выполнение **A[i]** не включает проверку того, что **i** не выходит за пределы объявленного размера массива, в то время как для Паскаля она существует. Таким образом, данный паскалевский код:

```
M := A[ i ]
```

эквивалентен следующему на Си:

```
if ( i >= 0 && i < n )
{
 M = A[ i ];
}
```

Так что имеет смысл ожидать, что различные языки программирования будут подвержены влиянию различных факторов, которые отразятся на подсчёте инструкций. В нашем примере, где мы используем «немой» паскалевский компилятор, игнорирующий возможности оптимизации, требуется по три внутренних инструкции на Паскале для каждого доступа к элементу массива вместо одной на Си. Пренебрежение этим фактором идёт в русле игнорирования различий между конкретными языками программирования с сосредоточением на анализе самой идеи алгоритма как таковой.

Описанные выше фильтры — «отбрось все факторы» и «оставляй только наибольший элемент» — в совокупности дают то, что мы называем **асимптотическим поведением**. Для $f(n) = 2n + 8$ оно будет описываться функцией $\boldsymbol{f(n) = n}$. Говоря языком математики, нас интересует предел функции $\boldsymbol{f}$ при $\boldsymbol{n}$, стремящемся к бесконечности. Строго говоря, в математической постановке мы не могли бы отбрасывать константы в пределе, но для целей теоретической информатики мы поступаем таким образом по причинам, описанным выше. Давайте проработаем пару задач, чтобы до конца вникнуть в эту концепцию.

Найдём асимптотики для следующих примеров, используя принципы отбрасывания константных факторов и оставления только максимально быстро растущего элемента.

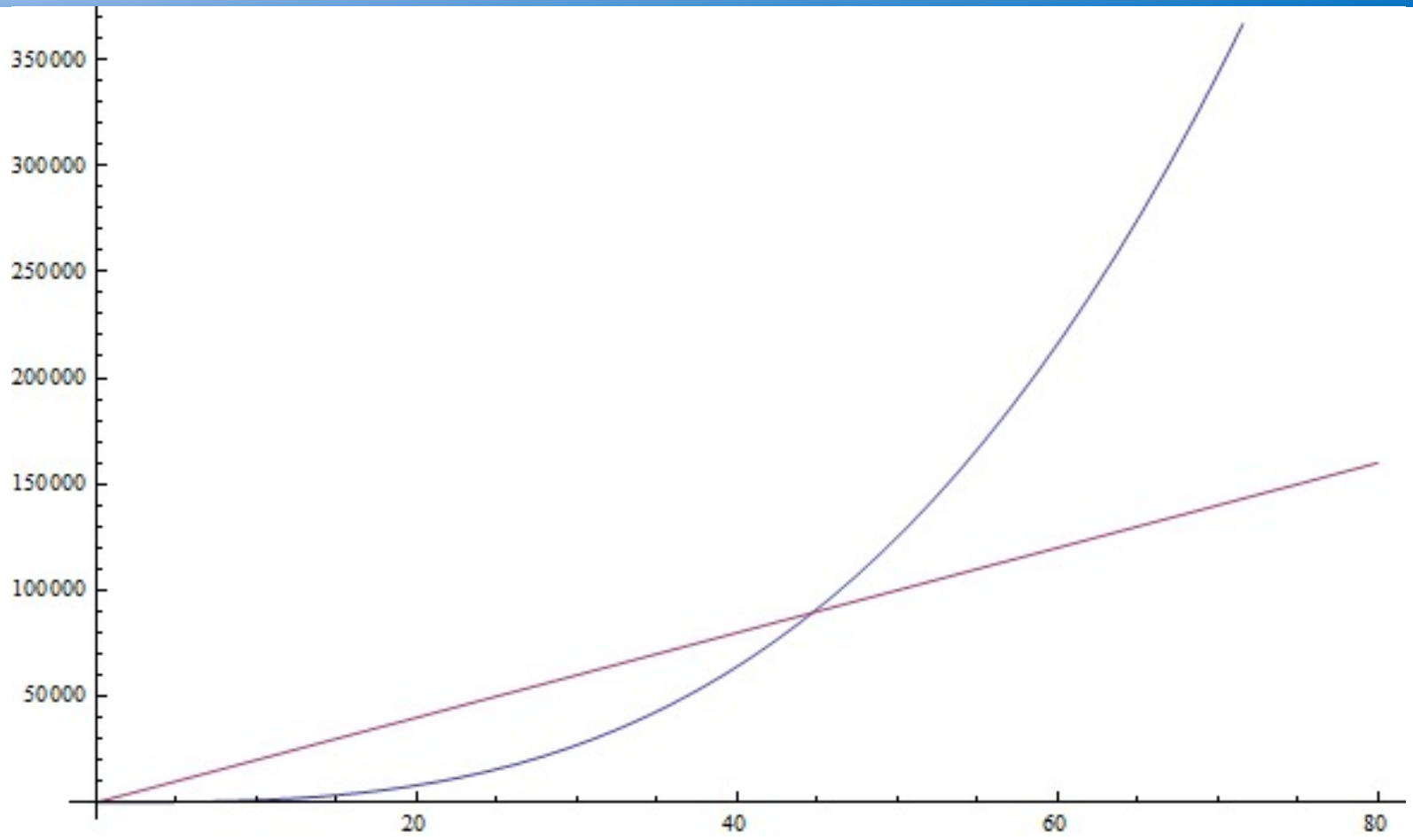
Пример 1. $f(n) = 5n + 12$ даст $\boldsymbol{f(n) = n}$**.** Основания — те же, что были описаны выше.

Пример 2. $f(n) = 109$ даст $\boldsymbol{f(n) = 1}$. Мы отбрасываем множитель в 109*1, но 1 по-прежнему нужна, чтобы показать, что функция не равна нулю.

Пример 3. $f(n) = n^2 + 3n + 112$ даст $\boldsymbol{f(n) = n^2}$. Здесь n^2 возрастает быстрее, чем $3n$, который, в свою очередь, растёт быстрее 112.

Пример 4. $f(n) = n^3 + 1999n + 1337$ даст $\boldsymbol{f(n) = n^3}$. Несмотря на большую величину множителя перед n, мы по-прежнему полагаем, что можем найти ещё больший n, поэтому $\boldsymbol{f(n) = n^3}$ всё ещё больше $1999n$ (рисунок на следующем слайде).

Асимптотическое поведение



Пример 5. $f(n) = n + \sqrt{n}$ даст $\boldsymbol{f(n) = n}$, потому что n при увеличении аргумента растёт быстрее, чем $\sqrt{n}$.

Упражнение. Определите асимптотическое значение $f(n)$.

1. $f(n) = n^6 + 3n$
2. $f(n) = 2^n + 12$
3. $f(n) = 3^n + 2n$
4. $f(n) = n^n + n$

Из всего сказанного можно сделать вывод, что если мы сможем отбросить все эти декоративные константы, то говорить об асимптотике функции подсчёта инструкций программы будет очень просто. Фактически любая программа, не содержащая циклы, имеет ***f*(*n*) = 1**, потому что в этом случае требуется постоянное число инструкций (конечно, при отсутствии рекурсии). Одиночный цикл от 1 до *n* даёт асимптотику ***f*(*n*) = *n***, поскольку до и после цикла выполняет неизменное число команд, а постоянное количество инструкций внутри цикла выполняется ***n* раз**.

Руководствоваться подобными соображениями менее утомительно, чем каждый раз считать инструкции, так что давайте рассмотрим несколько примеров.

Пример. Следующая программа проверяет, содержится ли в массиве A размера *n* заданное значение v:

```
ex = false;
for (i = 0; i < n; ++i )
{
   if (A[i] == v )
      {
        ex = true; break;
      }
}
```

Такой метод поиска значения внутри массива называется линейным поиском. Это обоснованное название, поскольку программа имеет ***f*(*n*) = *n***, то есть является функцией первого порядка (линейной) от *n*. Инструкция **break** позволяет программе выйти из цикла раньше, даже после единственной итерации. Однако, напоминаю, что нас интересует самый неблагоприятный сценарий, при котором массив **A** вообще не содержит заданного значения. Поэтому по-прежнему считаем, что ***f*(*n*) = *n***.

Пример. Рассмотрим программу, которая складывает два значения из массива и записывает результат в новую переменную:

```
v = a[ 0 ] + a[ 1 ]
```

Здесь у нас постоянное количество инструкций, следовательно, ***f*(*n*) = 1**.

Пример. Следующая программа на C++ проверяет, содержит ли массив A размера n два одинаковых значения:

```
bool duplicate = false;
for ( int i = 0; i < n; ++i ) {
   for ( int j = 0; j < n; ++j ) {
      if ( (i != j) && (A[ i ] == A[ j ]) )
{duplicate = true;break;
         }
                        }
   if ( duplicate ) {
      break;
   }
}
```

Два вложенных цикла дадут нам асимптотику вида $f(n) = n^2$.

Практическая рекомендация: простые программы можно анализировать с помощью подсчёта в них количества вложенных циклов. Одиночный цикл в n итераций даёт $f(n) = n$. Цикл внутри цикла — $f(n) = n^2$. Цикл внутри цикла внутри цикла — $f(n) = n^3$. И так далее.

Если в нашей программе в теле цикла вызывается функция, и мы знаем число выполняемых в ней инструкций, то легко определить общее количество команд для программы целиком.

Пример. Рассмотрим в качестве примера следующий код на C:

```
int i;
for ( i = 0; i < n; ++i )
{
   F( n );
}
```

Если нам известно, что $F(n)$ выполняет ровно n команд, то мы можем сказать, что количество инструкций во всей программе асимптотически приближается к n^2, поскольку $F(n)$ вызывается n раз.

Практическая рекомендация: если у нас имеется серия из последовательных **for-циклов**, то асимптотическое поведение программы определяет наиболее медленный из них. Два вложенных цикла, идущие за одиночным, асимптотически то же самое, что и вложенные циклы сами по себе. Говорят, что вложенные циклы доминируют над одиночными.

«О» большое и «о» малое

О большое и **о** малое – это математические обозначения для сравнения двух функций, означающие следующее:

1) функция $f(x) = \mathbf{O}(g(x))$, если существует константа C > 0, такая, что $|f(x)| \leq C \cdot |g(x)|$;

2) функция $f(x) = \mathbf{o}(g(x))$, если существует константа ε > 0, при которой $|f(x)| \leq \varepsilon \cdot |g(x)|$ в некоторой окрестности точки x_0.

Так или подобным образом описываются **О** большое и **о** малое на математическом языке. «По-русски» это означает следующее.

1) $\lim\limits_{x \to x_0} \frac{f(x)}{g(x)} = C$, например, $\lim\limits_{x \to 0} \frac{3x}{x} = 3$. Это происходит, когда функции $f(x)$ и $g(x)$ – одного порядка (одного порядка малости).

2) $\lim\limits_{x \to x_0} \frac{f(x)}{g(x)} = 0$, например, $\lim\limits_{x \to 0} \frac{x^3}{x} = x^2 \to 0$. Это происходит, когда функция $f(x)$ – более высокого порядка (более высокого порядка малости), чем функция $g(x)$.

Пример. $x^2 = \mathbf{o}(x)$ и $x^3 = \mathbf{o}(x)$, потому что, например, $x = 0{,}1 >> x^2 = 0{,}01$

O большое и **o** малое отвечают за сравнение двух бесконечно малых величин, в нашем примере: $x^2=\mathbf{o}(x)$, $x^3=\mathbf{o}(x)$.

По сути, **O** большое и **o** малое (**O** и **o**) — математические обозначения для сравнения асимптотического поведения (асимптотики) функций.

Под асимптотикой понимается характер изменения функции при её стремлении к определённой точке.

o(*f*), «**o** малое от *f*» обозначает «**бесконечно малое относительно *f***», пренебрежимо малую величину при рассмотрении *f*.

Смысл термина «**O** большое» зависит от его области применения, но всегда **O**(*f*) растёт не быстрее, чем *f*.

Историческая справка. Обозначение «**O** большое» введено немецким математиком Паулем Бахманом во втором томе его книги «Аналитическая теория чисел», вышедшем в 1894 году. Обозначение «**o** малое» впервые использовано другим немецким математиком, Эдмундом Ландау в 1909 году; с работами Ландау связана и популяризация обоих обозначений, в связи с чем их также называют символами Ландау. Обозначение пошло от немецкого слова «Ordnung» (порядок).

Фраза «сложность алгоритма есть **O**($f(n)$)» означает, что с увеличением параметра n, характеризующего количество входной информации алгоритма, время работы алгоритма будет возрастать не быстрее, чем некоторая константа, умноженная на $f(n)$.

Фраза «функция $f(x)$ является **o** малым от функции $g(x)$ в окрестности точки p» означает, что с приближением x к p $f(x)$ уменьшается быстрее, чем $g(x)$, то есть отношение $|f(x)|/|g(x)|$ стремится к нулю.

Обычно выражение «f является **O** большим от g» записывается с помощью равенства $f(x) = \mathbf{O}(g(x))$.

Обычно выражение «f является **o** малым от g» записывается с помощью равенства $f(x) = \mathbf{o}(g(x))$.

Такие обозначения очень удобны, но требуют некоторой осторожности при использовании. Дело в том, что $f(x) = \mathbf{O}(g(x))$ и $f(x) = \mathbf{o}(g(x))$ не являются равенствами в обычном смысле, представляют собой несимметричные отношения.

В частности, можно писать

$f(x) = \mathbf{O}(g(x))$ или $f(x) = \mathbf{o}(g(x))$,

но выражения

$\mathbf{O}(g(x)) = f(x)$ или $\mathbf{o}(g(x)) = f(x)$

бессмысленны.

Пример. При $x \to 0$ верно, что $\mathbf{O}(x^2) = \mathbf{o}(x)$,

но неверно, что $\mathbf{o}(x) = \mathbf{O}(x^2)$.

Замечание. При любом x верно, что $\mathbf{o}(x) = \mathbf{O}(x)$, то есть бесконечно малая величина является ограниченной, но неверно, что ограниченная величина является бесконечно малой: $\mathbf{O}(x) = \mathbf{o}(x)$.

Вместо знака равенства методологически правильнее было бы употребить знаки принадлежности и включения, понимая **O**( ) и **o**( ) как обозначения для множества функций, то есть используя запись в форме:

$x^3 + x^2 \in \mathbf{O}(x^3)$

или

$\mathbf{O}(x^2) \subset \mathbf{o}(x)$

вместо соответственно

$x^3 + x^2 = \mathbf{O}(x^3)$

и

$\mathbf{O}(x^2) = \mathbf{o}(x)$.

Однако на практике такая запись встречается крайне редко, в основном, в простейших случаях.

При использовании данных обозначений должно быть явно оговорено (или очевидно из контекста), о каких окрестностях (одно- или двусторонних; содержащих целые, вещественные, комплексные или другие числа) и о каких допустимых множествах функций идет речь (поскольку такие же обозначения употребляются и применительно к функциям многих переменных, к функциям комплексной переменной, к матрицам и др.).

Целые книги написаны по теории алгоритмов, но нам, простым смертным, которым не хочется влезать в дебри математики, но всё же любопытно узнать, как сравнивают алгоритмы по эффективности между собой, достаточно знать, что такое асимптотические нотации в общих чертах.

Допустим, у нас есть простейший алгоритм линейного поиска (то есть алгоритм, который перебирает последовательно все элементы массива друг за другом, в ходе этого запоминает индекс того элемента, значение которого равно искомому значению, а завершает работу, когда все элементы массива пройдены. То есть для искомого значения ***x*** массива **A** с количеством элементов ***n*** мы вправе записать следующий алгоритм:

1) присваиваем значение «не найдено» переменной «Ответ»;

2) для каждого индекса **i** (от 1 до *n*): если A[i] = *x*, то присваиваем переменной «Ответ» значение **i**;

3) возвращаем значение переменной «Ответ».

Нотация асимптотического роста

Попробуем подсчитать примерное время выполнения алгоритма в зависимости от n.
Допустим,
t1 — время на выполнение шага 1 (выполняется 1 раз),
t3 — время на выполнение шага 3 (один раз),
t2 — время на проверку условия в шаге 2 (n+1 раз),
t2′ — время на инкремент в шаге 2 (выполняется n раз),
t3 — время на проверку условия A[i] = x (выполняется n раз) и
t3′ — время на присваивание переменной «Ответ» = **i** (выполняется от 0 до n раз в зависимости от содержимого массива, в котором ищем значение).
Итак, в худшем случае имеем:
t1 + t2*(n+1) + t2′*n + t3*n + t3′ * n + t3,
а в лучшем:
t1 + t2 * (n+1) + t2′ * n + t2A *n + t2A′ * 0 + t3.
Сгруппируем слагаемые относительно переменного фактора n (все остальные величины — константы):
(t2 + t2′ + t3) * n + (t1 + t2 + t3) в лучшем случае и
(t2 + t2′ + t3 + t3′) * n + (t1 + t2 + t3) в худшем.
В обоих случаях у нас получается многочлен типа a*n + b, где a и b — константы.
Мы видим, что при увеличении n все меньше значения будет играть b.

Нотация асимптотического роста

Не вдаваясь в определение асимптотической нотации, покажем, что это такое. У функции, зависящей от количества шагов алгоритма, приведенной к виду многочлена, убираем все младшие члены и коэффициент при старшем члене. То есть a**n* + b превращается просто в *n*. Это называется тета (**θ**) функцией или нотацией. У нас и в лучшем, и в худшем случае тета будет зависеть только от *n*: **θ**(*n*).

Вообще асимптотическая нотация для «худшего» случая называется **O**-нотацией, а для «лучшего» случая — омега (**Ω**)-нотацией. В нашем случае они совпадают. То есть при **O**-нотации **O**(*n*) мы показываем, что время выполнения алгоритма не может быть больше, чем *n*, помноженный на коэффициент. Младшими членами, как уже показано, при увеличении количества шагов алгоритма мы можем пренебречь. Например, если у нас функция, описывающая быстродействие алгоритма, вроде такой: t1 * n^2 + t2**n* + t3, то тета-нотация будет такой: **θ**(n^2).

В целом, алгоритм с **O**(1) лучше, чем **O**(*n*), который лучше, чем алгоритм с **O**(*n**log *n*) и т.д.

Нотация асимптотического роста

Эти обозначения в лаконичном виде представлены в таблице:

Обозначение	Граница	Рост
(Тета) Θ	Нижняя и верхняя границы, точная оценка	Равно
(О - большое) O	Верхняя граница, точная оценка неизвестна	Меньше или равно
(о - малое) o	Верхняя граница, не точная оценка	Меньше
(Омега - большое) Ω	Нижняя граница, точная оценка неизвестна	Больше или равно
(Омега - малое) ω	Нижняя граница, не точная оценка	Больше

В следующей таблице представлены и другие подобные обозначения.

Для функций $f(n)$ и $g(n)$ при $n \to n_0$ используются следующие обозначения:

Обозначение	Интуитивное объяснение	Определение
$f(n) \in O(g(n))$	f ограничена сверху функцией g (с точностью до постоянного множителя) асимптотически	$\exists(C > 0), U : \forall(n \in U)\ \lvert f(n)\rvert \leq C\lvert g(n)\rvert$
$f(n) \in \Omega(g(n))$	f ограничена снизу функцией g (с точностью до постоянного множителя) асимптотически	$\exists(C > 0), U : \forall(n \in U)\ C\lvert g(n)\rvert \leq \lvert f(n)\rvert$

А теперь сопоставим сложность алгоритма с эффективностью его выполнения. Результат такого сопоставления сведем в таблицу. В этой таблице если алгоритм имеет сложность, указанную в левом столбце, то его эффективность – в правом столбце:

Алгоритм	Эффективность
o(n)	< n
O(n)	≤ n
Θ(n)	= n
Ω(n)	≥ n
ω(n)	> n

И в заключение этой темы предлагаю рассмотреть числовой пример, который, я уверена, вас удивит, если не поразит.

n (размер задачи)	O(n)	$O(2^n)$
50	1 сек	1 сек
51	1,02 сек	2 сек
60	1,2 сек	17 мин
70	1,4 сек	12 суток
80	1,6 сек	34 года
90	1,8 сек	~35 тыс. лет

АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ:

машинно-ориентированные | процедурно-ориентированные

проблемно-ориентированные

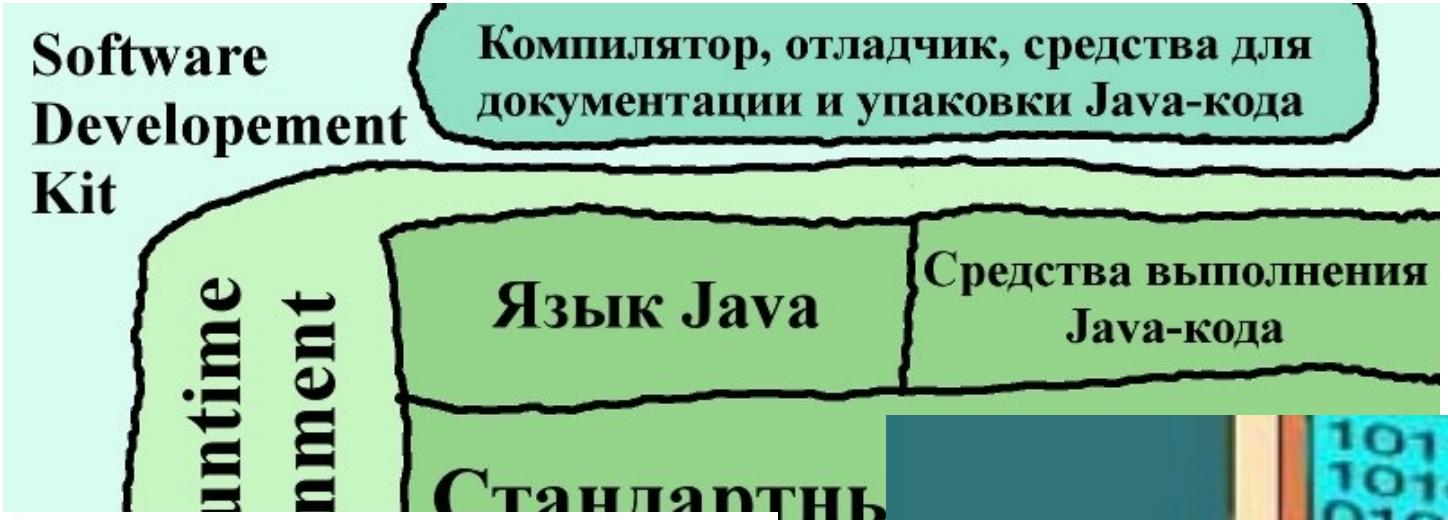
Машинно-ориентированные языки программирования низкого уровня — программирование на них наиболее трудоемко, но позволяет создавать оптимальные программы, максимально учитывающие функционально-структурные особенности конкретного компьютера.

Процедурно-ориентированные и **проблемно-ориентированные** языки относятся к языкам высокого уровня, использующим макрокоманды. Макрокоманда при трансляции генерирует много машинных команд.

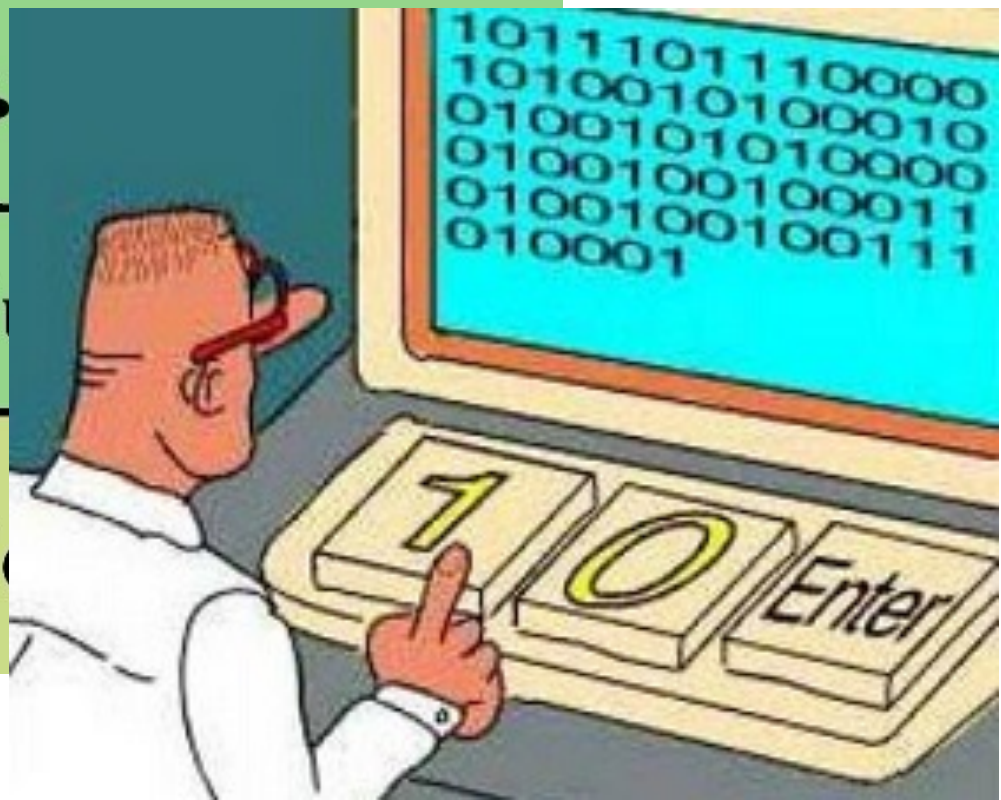
ТРАНСЛЯТОРЫ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

КОМПИЛЯТОРЫ при трансляции переводят на машинный язык сразу всю программу и затем хранят ее в памяти машины в двоичных кодах.

ИНТЕРПРЕТАТОРЫ каждый раз при исполнении программы заново преобразуют в машинные коды каждую макрокоманду и передают ее для непосредственного выполнения компьютеру.


Software
Developement
Kit
Компилятор, отладчик, средства для документации и упаковки Java-кода
Язык Java
Средства выполнения Java-кода


Код на языке C#
Компилятор C#
Код на языке IL
JIT-компилятор
* запуск на исполнение
Машинный код


1
0
Enter

Команда машинной программы (иначе, **машинная команда**) — это элементарная инструкция машине, выполняемая ею автоматически без каких-либо дополнительных указаний и пояснений. Машинная команда состоит из двух частей: операционной и адресной.

Операционная часть команды (КОП — код операции) — это группа разрядов в команде, предназначенная для представления кода операции машины.

Адресная часть команды (Адреса) — это группа разрядов в команде, в которых записываются коды адреса (адресов) ячеек памяти машины, предназначенных для оперативного хранения информации, или иных объектов, задействованных при выполнении команды.

Адреса команды делятся на:

одно-, двух- и трехадресные

КОП	a1	a2	a3

a1 — адрес ячейки (регистра), куда следует поместить число, полученное в результате выполнения операции.

КОП	a1	a2

a1 — это обычно адрес ячейки (регистра), где хранится первое из чисел, участвующих в операции.

КОП	a1

a1 в зависимости от модификации может обозначать либо адрес ячейки (регистра), либо адрес ячейки (регистра) куда следует поместить число.

безадресные содержат только код операции, а информация должна быть заранее помещена в определенные регистры машины.

Виды машинных команд по выполняемым операциям:

- операции пересылки информации внутри компьютера
- арифметические операции над информацией
- логические операции над информацией
- операции над строками (текстовой информацией)
- операции обращения к внешним устройствам компьютера
- операции передачи управления
- обслуживающие и вспомогательные операции

Операции передачи управления (или иначе — ветвления программы), которые служат для изменения естественного порядка выполнения команд.

Виды передачи управления: БЕЗУСЛОВНАЯ И УСЛОВНАЯ

Операции безусловной передачи управления всегда приводят к выполнению после данной команды не следующей по порядку, а той, адрес которой в явном или неявном виде указан в адресной части команды.

Условная операция тоже вызывают передачу управления по адресу, указанному в адресной части команды, но только в том случае, если выполняется некоторое заранее оговоренное для этой команды условие.

Команды безусловной передачи управления:

Команда передачи управления, которая просто передает управление по заданному адресу и больше никаких действий не выполняет

Команда передачи управления (ее часто называют командой вызова процедуры или подпрограммы), которая, кроме передачи управления процедуре, еще и запоминает в специальной стековой памяти адрес следующей команды (адрес возврата из процедуры)

Безадресная команда передачи управления (команда возврата из процедуры) возвращающая управление по запомненному адресу возврата

АДРЕСАЦИЯ РЕГИСТРОВ И ЯЧЕЕК ПАМЯТИ

Адресация операндов в командах программы:

- Непосредственная
- Прямая
- Косвенная
- Ассоциативная
- Неявная

Непосредственная адресация заключается в указании в команде самого значения операнда, а не его адреса.

Прямая адресация состоит в указании в команде непосредственно абсолютного или исполнительного адреса операнда.

Косвенная адресация имеет в виду указание в команде регистра(ов) или ячейки памяти, в которых находятся абсолютный, исполнительный адрес операнда или их составляющие.

Ассоциативная адресация — указание в команде не адреса, а идентифицирующего содержательного признака операнда, подлежащего выборке.

Неявная адресация — адреса операнда в команде не указано, но он подразумевается кодом операции.

Абсолютный (Аинд) адрес формируется как сумма адресов исполнительного (Аисп) и сегментного (Асегм):

$$Aabc = Асегм + Аисп;$$

$$Асегм = 16 \cdot A'сегм;$$

$$Аисп = [Асмещ] + [Абаз] + [Аинд].$$

При адресации данных могут использоваться все составляющие адреса:

$$Aabc = Асегм + [Асмещ] + [Абаз] + [Аинд].$$

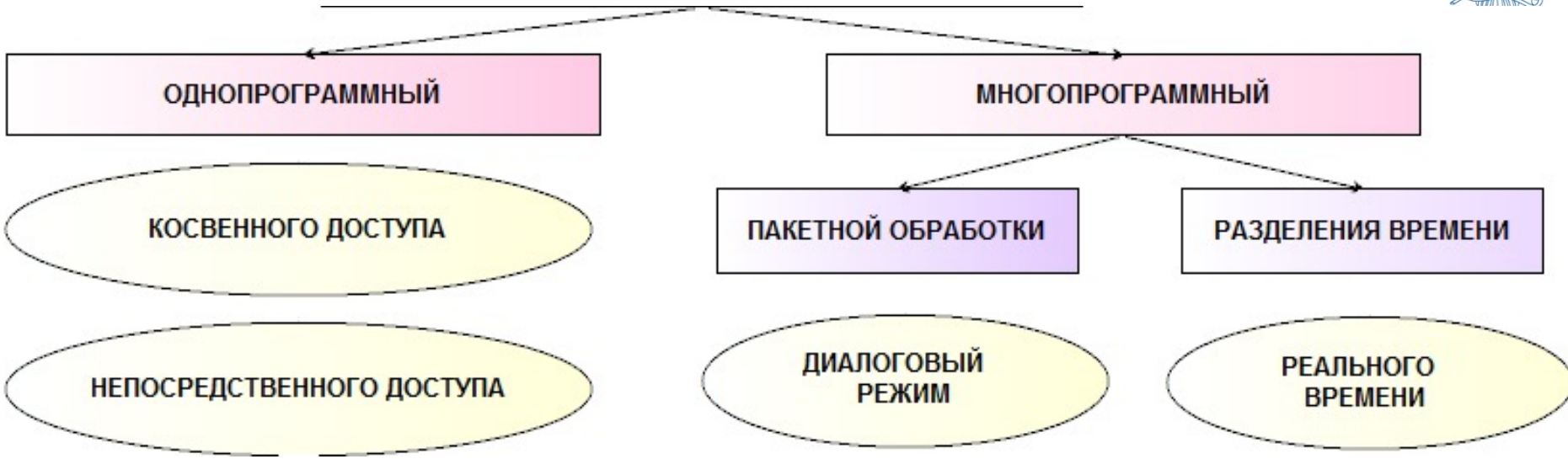
При адресации команд программы могут использоваться только две составляющие адреса:

$$Aabc = Асегм + [Асмещ] = 16 \cdot A'сегм + Асмещ .$$

ОСНОВНЫЕ ЭТАПЫ ПОДГОТОВКИ И РЕШЕНИЯ ЗАДАЧИ НА КОМПЬЮТЕРЕ

1. Формулировка и формализованная постановка задачи.
2. Выбор математической модели и метода решения задачи.
3. Разработка алгоритма решения задачи, то есть последовательности процедур, которые необходимо выполнить для решения задачи.
4. Составление программы решения задачи, то есть запись алгоритма решения задачи на языке, понятном машине.
5. Ввод программы в компьютер и ее отладка.
6. Ввод исходных данных и решение задачи на компьютере.
7. Анализ полученных результатов и выводы по результатам решения.

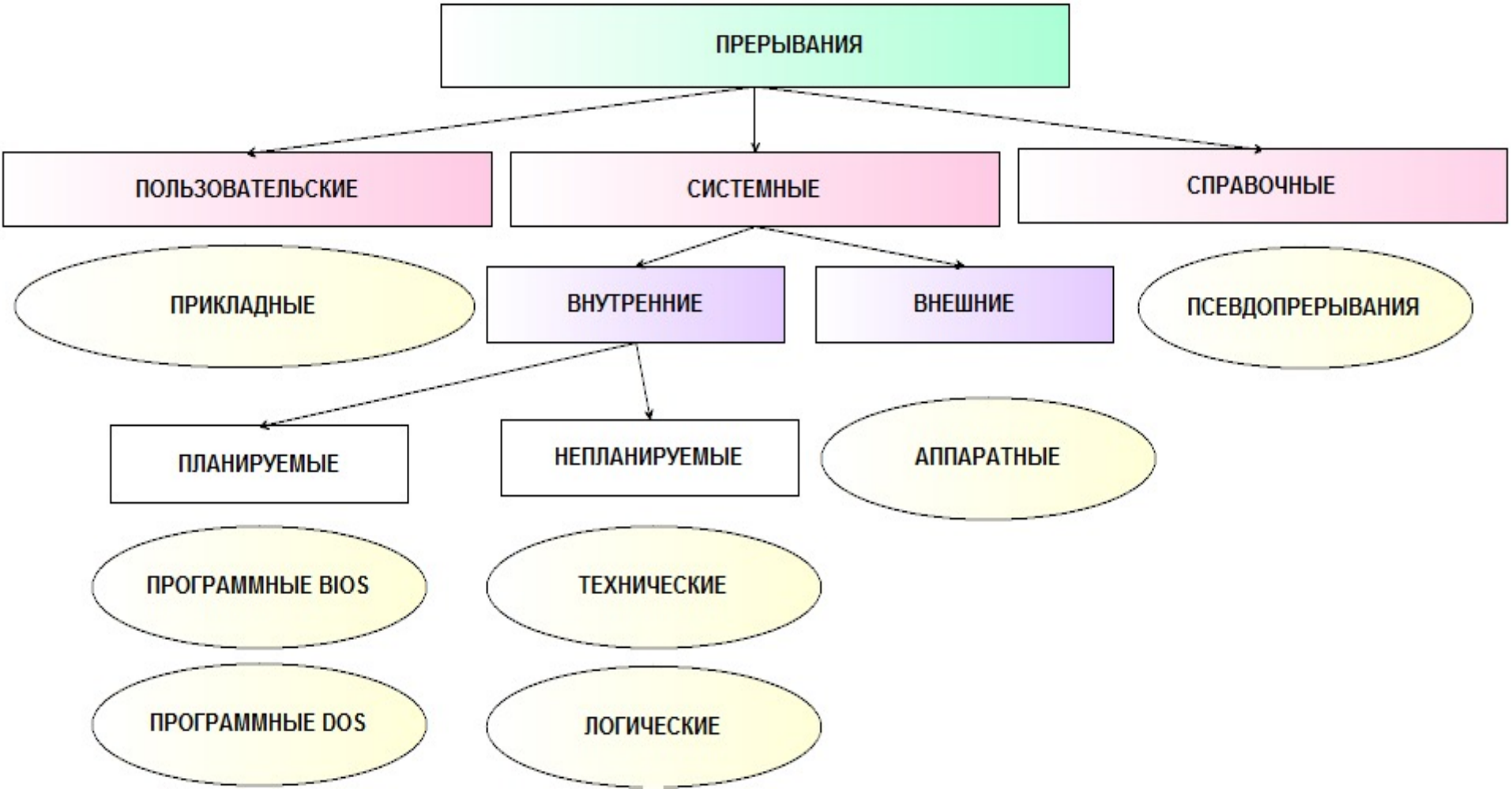
РЕЖИМЫ РАБОТЫ КОМПЬЮТЕРА



Однопрограммный режим использования компьютера самый простой, применяется во всех поколениях компьютеров. Из современных машин этот режим чаще всего используется в персональных компьютерах.

Многопрограммный режим обеспечивает лучшее расходование ресурсов компьютера, но несколько ущемляет интересы пользователя. Для реализации этого режима необходимо прежде всего разделение ресурсов машины в пространстве и во времени.

СИСТЕМА ПРЕРЫВАНИЙ КОМПЬЮТЕРА



ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программное обеспечение

Программное обеспечение (англ. software) — это совокупность программ, обеспечивающих функционирование компьютеров и решение задач предметных областей.

ВИДЫ ПО:

Системное ПО - управляет работой компьютера и выполняет различные вспомогательные функции;

Прикладное ПО - предназначено для решения конкретных задач пользователя в предметных областях;

Инструментальное ПО - это средства создания программного обеспечения и информационных систем (системы программирования, инструментальные среды и системы моделирования).

Системное программное обеспечение предназначено для организации эффективной работы компьютера и пользователя, а также эффективного выполнения прикладных программ.

Операционная система	Сервисные программы

Операционная система (ОС) — это комплекс программ, предназначенных для управления загрузкой, запуском и выполнением других пользовательских программ, а также для планирования и управления вычислительными ресурсами компьютера, т.е. управления ее работой с момента включения до момента выключения питания.

Операционная система (ОС) – это комплекс программ, которые обеспечивают возможность рационального использования оборудования компьютера удобным для пользователя образом. Фундаментальным понятием для изучения работы операционных систем является понятие **процессов** как основных динамических объектов, над которыми системы выполняют определенные действия.

На сегодняшний день, операционная система — это первый и основной набор программ, загружающийся в компьютер. Помимо вышеуказанных функций ОС может осуществлять и другие, например, предоставление общего пользовательского интерфейса.

Операционная система - это виртуальная машина

Операционная система - это менеджер ресурсов

Операционная система - это защитник пользователей и программ

Операционная система - это постоянно функционирующее ядро

Понятие **процесса** для операционной системы является ключевым.

Процесс – это:

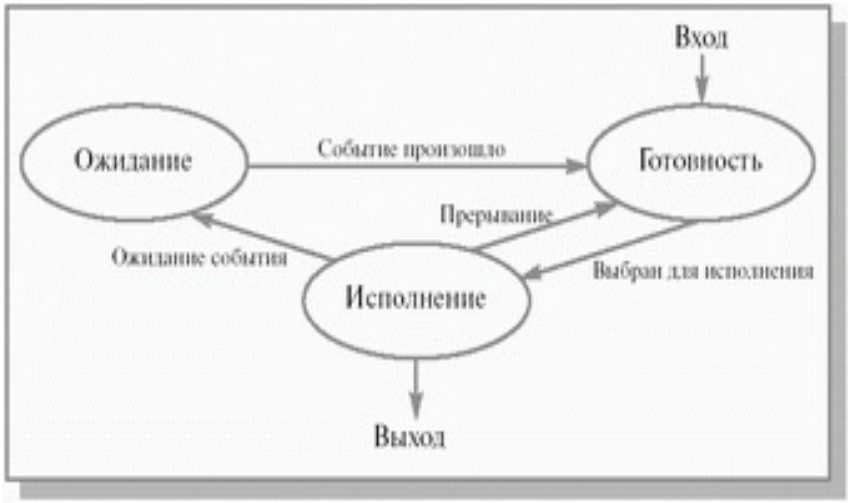
- последовательность исполняющихся команд;
- соответствующие ресурсы (выделенная для выполнения память или адресное пространства, файлы, устройства ввода-вывода и т.д.;
- текущее состояние – состояние программного счетчика и регистра флагов;
- состояние регистра, стеков, рабочих переменных.

Понятие **процесса** характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы.

Процесс может находиться как минимум в двух состояниях: процесс исполняется и процесс не исполняется.

Это очень грубая модель, она не учитывает, в частности, того, что процесс, выбранный для исполнения, может все еще ждать события, из-за которого он был приостановлен, и реально к выполнению не готов.

Для того чтобы избежать такой ситуации, разобьем состояние «Процесс не исполняется» на два новых состояния: «Готовность» и «Ожидание»

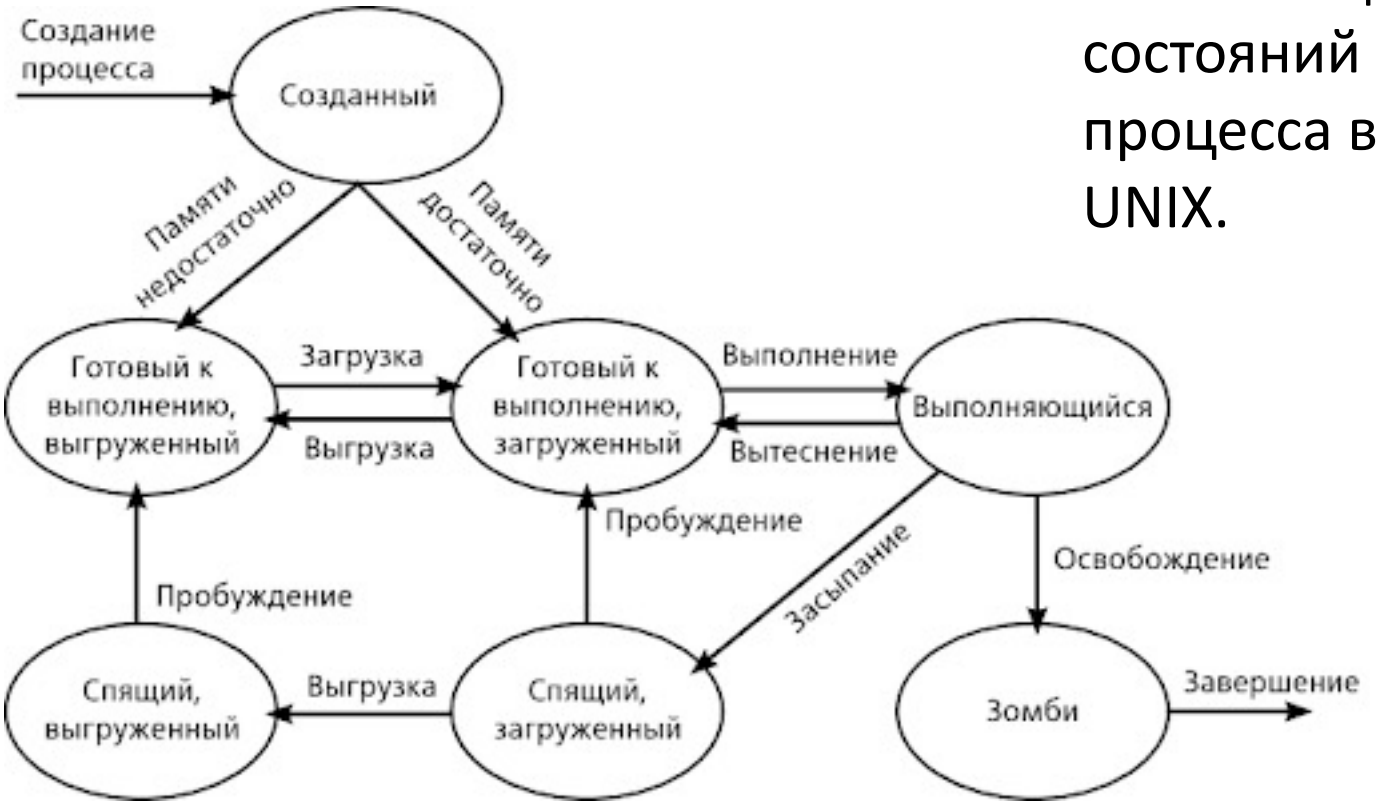


Эта модель хорошо описывает поведение процессов во время их существования, но она не акцентирует внимания на появлении процесса в системе и его исчезновении.

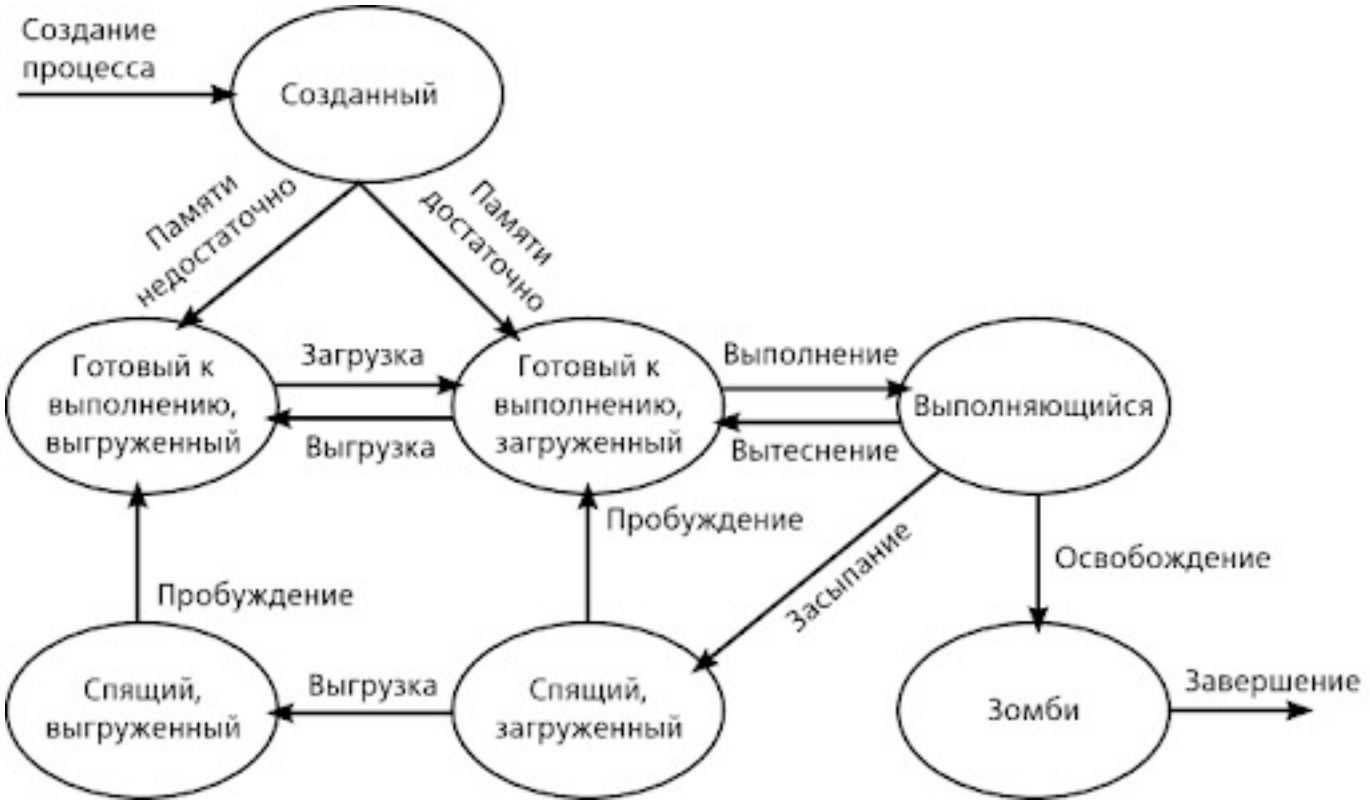
Для полноты картины нам необходимо ввести еще два состояния процессов: «Рождение» и «Закончил исполнение».

Это наиболее полная модель, описанная в учебниках.

А это наиболее полная модель состояний процесса в ОС UNIX.



Состояния процесса



Большинство этих состояний совпадает с классическим набором состояний процессов в многозадачных операционных системах. Для операционной системы UNIX характерно особое состояние процесса — *зомби*. Процесс получает это состояние, если он завершился раньше, чем этого ожидал его родительский процесс. В UNIX перевод процессов в состояние зомби служит для корректного завершения группы процессов, освобождения ресурсов и т.п.

Итак, операционная система (ОС) — это комплекс программ, предназначенных для управления загрузкой, запуском и выполнением других пользовательских программ, а также для планирования и управления вычислительными ресурсами ЭВМ, т.е. управления ее работой с момента включения до момента выключения питания.

ОСНОВНЫЕ ФУНКЦИИ ОС:

- Поддержка диалога с пользователем
- Планирование и организация процесса обработки программ
- Запуск программ на выполнение
- Поддержка работы периферийных устройств (внешние устройства)
- Ввод-вывод и управление данными
- Распределение ресурсов (оперативной и кэш памяти, процессора, внешних устройств)
- Выполнение вспомогательных операций обслуживания
- Передача информации между различными внутренними устройствами

1. Модуль, управляющий файловой системой
2. Модуль, расшифровывающий и выполняющий команды (командный процессор)
3. Драйверы периферийных устройств

Дополнительным элементом современных операционных систем (надстройкой) являются модули, обеспечивающие пользовательский интерфейс.

Классификация ОС по признакам

По числу параллельно решаемых на компьютере задач

По числу одновременно работающих пользователей

По типу интерфейса

По типу аппаратуры

По числу разрядов адресной шины компьютеров

Основная задача **файловой системы** — обеспечение взаимодействия программ и физических устройств ввода/вывода (различных накопителей). Она также определяет структуру хранения файлов и каталогов на диске, правила задания имен файлов, допустимые атрибуты файлов, права доступа и др.

Файл — это поименованная последовательность любых данных, стандартная структура которой обеспечивает ее размещение в памяти машины (компьютера).

АТРИБУТЫ ФАЙЛА:

R (Read-Only) — «только для чтения»

H (Hidden) — «скрытый файл»

A (Archive) — «неархивированный файл»

Надежность ПО – это вероятность безотказной работы ПО на основе заданных условий в течении определенного промежутка времени.

Основные задачи при проектировании ПО:

1. Выявление и исключение ошибок на этапе разработки ПО;

2. Проектирование отказоустойчивого ПО, которая выявляет ошибки программы за счет ошибок аппаратной части и восстанавливает нормальное состояние ПО.

Критерии оценки ПО:

Функциональные в соответствии с поставленной задачей

Конструктивные полностью характеризующие ПО

Основные виды контроля:

Визуальный	Статический	Динамический

Визуальный контроль – это проверка программы без использования компьютера. *Сквозной контроль программы* – насколько корректно написана программа.

Статический контроль – проверка программы на правдоподобия текста без применения инструментальных средств (целостность, живучесть, завершенность, работоспособность).

Динамический контроль – проверка правильности программы и ее выполнение на компьютере.

Литература

1. Введение в анализ сложности алгоритмов (часть 1). [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/196560/
2. Введение в анализ сложности алгоритмов (часть 2). [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/195482/
3. «О» большое и «о» малое. [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/«О» большое и «о» малое
4. Как сравнивают быстродействие алгоритмов. [Электронный ресурс]. Режим доступа: http://langtoday.com/?p=343
5. ГОСТ 19.701-90 - http://cert.obninsk.ru/gost/282/282.html
6. https://sites.google.com/site/anisimovkhv/learning/pris/lecture/tema8/tema8_2
7. Бройдо В.Л. Вычислительные системы, сети и телекоммуникации: Учеб. пособие для вузов.-2-е изд. - СПб.:Питер, 2015. - 702 с.
8. Стефан Р. Дэвис. C++ Для чайников [Электронный ресурс] http://www.proklondike.com/books/cpp/cplus_dlja_chainikov.html
9. Магда Ю.С. Программирование и отладка C/C++ приложений для микроконтроллеров ARM. - М.: ДМК Пресс, 2015. - 168 с.