

Структуры и алгоритмы обработки данных

Лекция 1.

Рысин М.Л.

Литература по алгоритмизации:

1. **Кнут Д.** Искусство программирования. Тома 1-4, 1976-2013.
2. **Вирт Н.** Алгоритмы + структуры данных = программы, 1985.
3. **Лафоре Р.** Структуры данных и алгоритмы в Java. 2-е изд., 2013.
4. **Макконнелл Дж.** Основы современных алгоритмов. 2-е изд., 2004.
5. **Седжвик Р., Уэйн К.** Алгоритмы на Java. 4-е изд., 2013.
6. **Скиена С.** Алгоритмы. Руководство по разработке, 2011.
7. **Стивенс Р.** Алгоритмы. Теория и практическое применение (C#), 2016.
8. **Хайнеман Д. и др.** Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017.

Литература по C++:

1. **Страуструп Б.** Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
2. **Прата С.** Язык программирования C++. Лекции и упражнения. - 6-е изд., 2012.
3. **Шилдт Г.** Полный справочник по C++. 4-е изд., 2006.
4. **Хортон А.** Visual C++ 2010. Полный курс, 2011.
5. **Седжвик Р.** Фундаментальные алгоритмы на C++, 2001-2002
6. **Павловская Т.А.** C/C++. Программирование на языке высокого уровня, 2003.

Интернет-ресурсы (общего назначения):

- **Национальный открытый университет «ИНТУИТ»** [Электронный ресурс] URL: <http://www.intuit.ru/> (дата обращения 11.09.2016).
- **Хабр** [Электронный ресурс]. URL: <http://habr.ru/> (дата обращения 01.07.2014).
- **Tproger** [Электронный ресурс] URL: <https://tproger.ru/> (дата обращения 07.09.2018).
- **CodeNet – всё для программиста** [Электронный ресурс]. URL: <http://www.codenet.ru/> (дата обращения 07.09.2018).
- **MIT OpenCourseWare** [Электронный ресурс]. URL: <http://ocw.mit.edu> (дата обращения 01.07.2014).

1. Алгоритмы: вводные понятия.

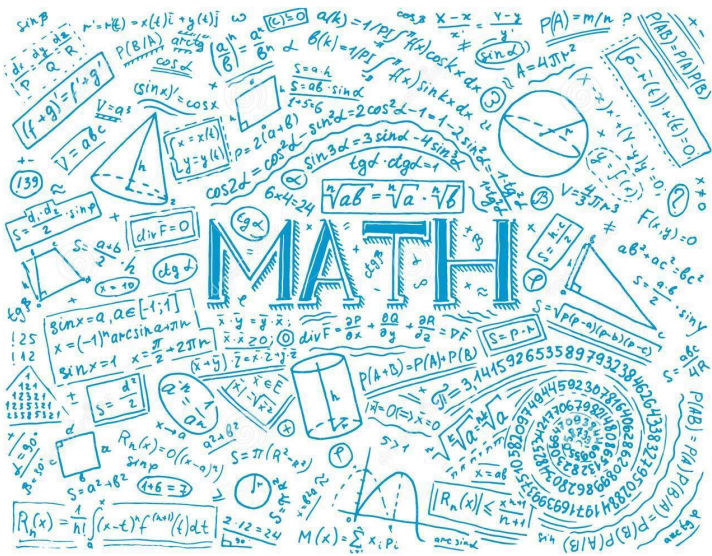
Алгоритм (лат. algorithmi) –



- Это набор инструкций, описывающих порядок действий **исполнителя**, для достижения определённого результата
(неформальное определение)
- Базисное понятие в математике:
- **Вычисления**
(вычислительная задача) – это обработка числовой информации по

Алгоритм вычислений

- Алгоритм решения **вычислительной задачи** – это корректно определённая **вычислительная процедура**, на вход которой подаётся значение (набор значений), и результатом выполнения которой является **выходное значение** (набор значений)
- Алгоритм **корректен**, если для каждого ввода результатом его работы



Исполнитель –



- Это абстрактная или реальная (техническая или биологическая) **система**, способная выполнить действия, предписываемые алгоритмом
- **Неформальный** (знает конечную цель А.) и **формальный**

Характеристики:

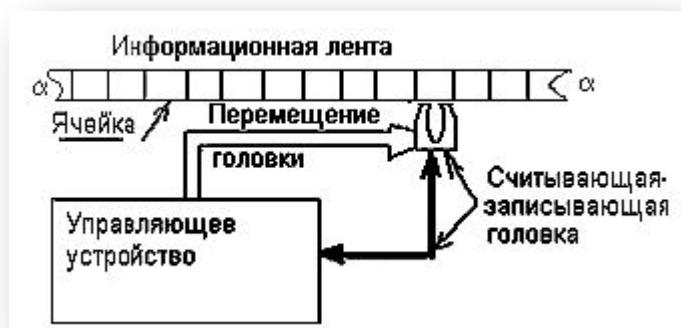
- **Среда** (обстановка) – место действия
- Система **команд**:
 - Должны быть заданы **условия применимости** (состояние среды)
 - Описаны **результаты** выполнения
- Набор **действий**
- **Отказы** (недопустимое для выполнения команды состояние среды).

Теория алгоритмов –



- Наука на стыке математики и информатики об общих свойствах и закономерностях **алгоритмов** и разнообразных формальных моделях их представления
- **Теоретическая основа вычислительных наук**
- Задачи:
 - **Формализация алгоритма** (модели вычислений) →
 - **Формализация задач**
 - Алгоритмическая **неразрешимость**
 - Уровни **сложности** (классификация, анализ.

Способы формализации алгоритма



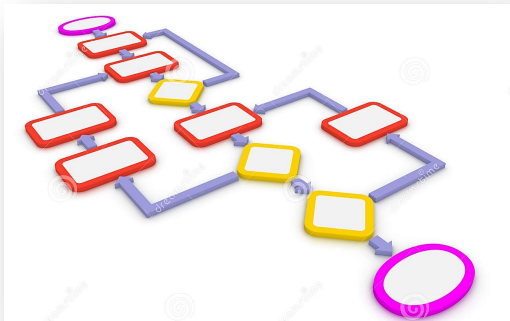
- Теория автоматов:
 - машина Тьюринга,
 - машина Поста;
- **Рекурсивные функции** Гёделя — Эрбрана — Клини
- **Нормальный алгоритм** Маркова
- **λ -исчисление** Чёрча.

Виды алгоритмов



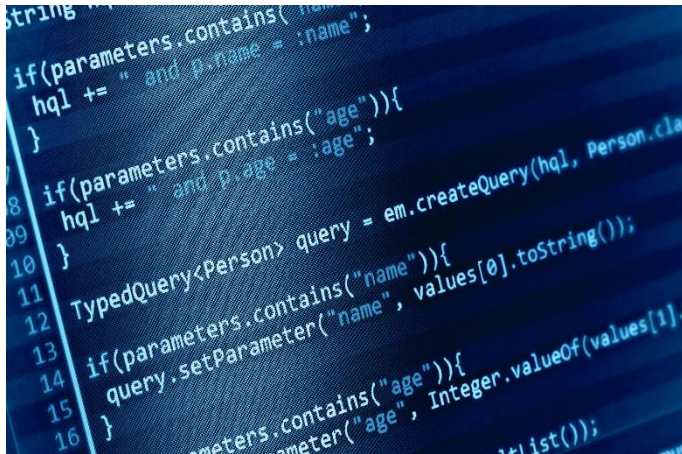
- **Детерминированные** (жёсткие, механические) – единственная и достоверная последовательность инструкций, приводящая к однозначному результату
- **Гибкие:**
 - **Вероятностные** (стохастические):
 - Используют **случайные величины** (ГСЧ),
 - Несколько путей решения, приводящими к **высоко вероятному достижению результата**;
 - **Эвристические** – используют различные разумные соображения без строгих обоснований.

Свойства алгоритма:



- **Дискретность** – разбиение на конечное количество отдельных шагов
- **Понятность** – включает только команды из набора допустимых команд исполнителя
- **Детерминированность** (определённость) – каждый следующий шаг однозначно определяется состоянием системы – один и тот же ответ для одних и тех же исходных данных
- **Результативность** – всегда приводит к получению определённого результата
- **Массовость** – применимость к множеству наборов начальных данных
- **Завершаемость** (конечность) – результат за конечное время (число шагов).

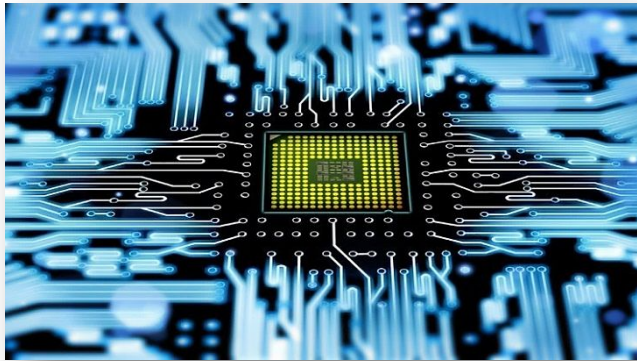
Способы записи алгоритма



```
string hql = "select p from Person p";
if(parameters.contains("name")){
    hql += " and p.name = :name";
}
if(parameters.contains("age")){
    hql += " and p.age = :age";
}
TypedQuery<Person> query = em.createQuery(hql, Person.class);
if(parameters.contains("name")){
    query.setParameter("name", values[0].toString());
}
if(parameters.contains("age")){
    query.setParameter("age", Integer.valueOf(values[1].toString()));
}
return query.getResultList();
```

- **Словесный** (на естественном языке)
- **Формульный**
- **Табличный** (для реляционных задач)
- **Графический** (блок-схемы)
- **Операторный** – из конечного набора допустимых команд исполнителя.

Компьютерная программа –



- Это алгоритм решения **вычислительной задачи** компьютером
- **Исполнитель**
- **Машинная команда:**
 - КОп (обяз. часть)
 - Адресная часть

Команда 1:	01011000	0110	00000000000100000000
	операция	операнд 1	операнд 2
Команда 2:	01011010	0110	00000000000100000100
	операция	операнд 1	операнд 2
Команда 3:	01010000	0110	00000000000100001000
	операция	операнд 1	операнд 2

BB 11 01 B9 0D 00 B4 0E 8A 07
43 CD 10 E2 F9 CD 20 48 65 6C
6C 6F 2C 20 57 6F 72 6C 64 21

- **Скрипт.**

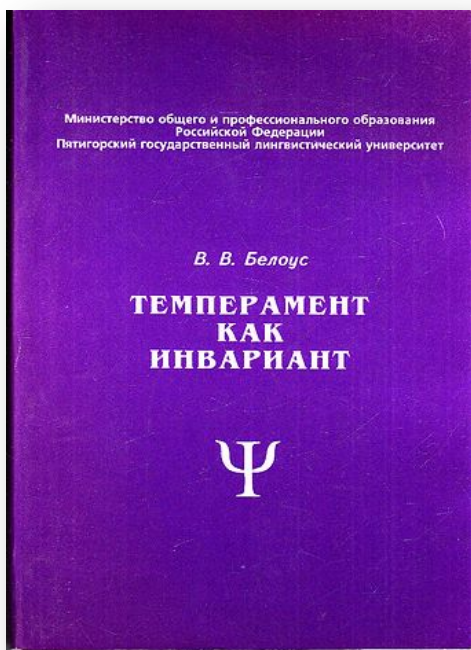
Язык программирования –



- Это набор допустимых **операторов, синтаксические и семантические** правила их использования для создания компьютерных программ
- Уровневая классификация:
 - **ЯВУ**
 - **Ассемблеры** - машиноориентированные
 - Язык двоичных машинных кодов (**нативный код**)
- **Трансляция:**
 - **Интерпретация**
 - **Компиляция.**

2. Корректность алгоритма.

Инвариант



- Алгоритм **корректен**, если для каждого ввода результатом его работы является **корректный вывод**
- Методы оценки корректности – на принципах **математической индукции** (путём рассуждений)
- **Инвариант** – это свойство некоторого класса (множества) мат.объектов, остающееся **неизменным** при определённого вида преобразованиях.

Инвариант цикла –

```
int j = 9;
for(int i=0; i<10; i++)
    j--;
```

Примеры

инвариантов:

а) $i + j == 9$

б) $i \geq 0 \ \&\& \ i \leq 10$

- Свойство, сохраняемое циклом – это **логическое выражение** (предикат), **истинное** непосредственно **перед** и сразу **после** каждой итерации цикла, зависящее от переменных, изменяющихся в теле цикла
- Инвариант цикла \neq условие цикла
- Инвариант может быть использован для **доказательства корректности** циклического алгоритма без необходимости его непосредственного выполнения (**верификация**)
- Чтобы убедиться, что **оптимизированный цикл** остался корректным, достаточно доказать, что **инвариант цикла не нарушен** и условие завершения цикла **достижимо**.

Доказательство корректности цикла инвариантом

1. Доказывается, что выражение инварианта истинно перед началом цикла (**инициализация**).
2. Доказывается, что выражение инварианта сохраняет свою истинность после выполнения тела цикла (**сохранение**). Так, по индукции, доказывается, что по завершении цикла инвариант будет выполняться.
3. Доказывается, что при истинности инварианта после завершения цикла (**завершение**) переменные примут те значения, которые и требуется получить (что определяется из выражения инварианта и конечных значениях переменных в условии цикла).
4. Доказывается (возможно, без применения инварианта), что цикл завершится, то есть условие завершения рано или поздно будет выполнено.
 - Истинность утверждений на этих этапах однозначно свидетельствует о том, что цикл выполнится за конечное время и даст желаемый результат.

Схема проверки инварианта цикла

```
// Инвариант цикла должен быть истинным здесь – при инициализации
while ( Условие_выполнения_цикла ) {
    // начало тела цикла
    ...
    // конец тела цикла
    // Инвариант цикла должен быть истинным здесь (после итерации)
}
// Инвариант цикла должен быть истинным здесь – по завершении цикла
}
```

Пример – алгоритм поиска минимума в массиве

```
Min ← A[1]
for i ← 2 to n do
  if A[i] < Min then
    Min ← A[i]
  endif
od
```

Формулировка
инварианта:

- После выполнения каждого шага цикла в переменной Min записан минимум из первых i элементов $[1, i)$ массива.

Область неопределённости

```
Min ← A[1]
for i ← 2 to n do
  if A[i] < Min then
    Min ← A[i]
  endif
od
```

- Область изменения параметров задачи $[1, n)$ можно разделить на две части:
 - **исследованную область**, для которой найден Min в $[1, i)$;
 - **область неопределенности** $[i+1, n)$.
- Необходимо составлять цикл так, чтобы **на каждой итерации область неопределенности сокращалась**
- В начале первой итерации исследованная область представляла собой единственную точку 1, а область неопределенности составляла $[2, n)$
- На втором шаге область неопределенности сократилась до $[3, n)$, на третьем – до $[4, n)$ и т.д., пока не превратится в пустое

Пример – алгоритм суммирования элементов массива

```
Sum ← 0  
for i ← 1 to n do  
  Sum ← Sum + A[i]  
od
```

- После каждого шага цикла при любом i к переменной Sum добавляется элемент массива $A[i]$
- После окончания очередного шага цикла **в Sum накоплена сумма всех элементов массива с номерами от 1 до i**
- Вывод: после завершения цикла ($i=n$), в Sum будет записана сумма всех элементов массива.

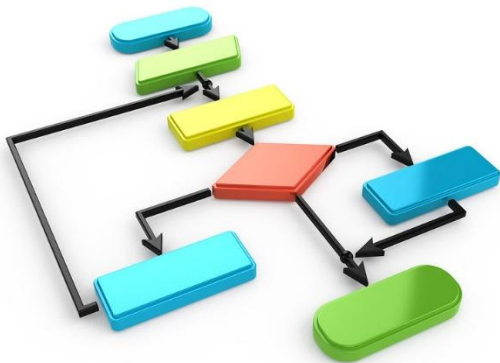
Пример – сортировка массива пузырьком

```
for i←1 to n-1 do
  for j←n-1 downto i do
    if A[j] > A[j+1] then
      c←A[j]; A[j] ← A[j+1]; A[j + 1] ←c;
    endif
  od
od
```

- На каждом шаге **внешнего цикла** на свое место «всплывает» один элемент массива
- Поэтому **инвариант внешнего цикла**: «После выполнения i -го шага цикла первые i элементов массива отсортированы и установлены на свои места»
- Во **внутреннем цикле** очередной «лёгкий» элемент поднимается вверх к началу массива
- Перед первым шагом внутреннего цикла элемент, который будет стоять на i -м месте в отсортированном массиве, может находиться в любой ячейке от $A[i]$ до $A[n]$
- После каждого шага его «зона нахождения» сужается на одну позицию
- **Инвариант внутреннего цикла**: «Элемент на i -м месте в отсортированном массиве может находиться в любой ячейке от $A[i]$ до $A[j]$ »
- Когда в конце этого цикла $j = i$, элемент $A[i]$ встаёт на своё место.

3. Анализ эффективности алгоритма

Анализ алгоритма



- Позволяет предсказать **требуемые** для его выполнения **ресурсы** (время работы процессора, память и пр.)
- На основе анализа нескольких алгоритмов можно выбрать **наиболее эффективный**. →

Эффективность алгоритма



- Критерии – **скорость** (время) и **расход памяти** (или других ресурсов – диска, трафик в сети и пр.)
- Алгоритм A1 эффективнее алгоритма A2, если алгоритм A1 выполняется за меньшее время и (или) требует меньше компьютерных ресурсов
- Составляющие эффективности:
 - Время – мера **системной эффективности**
 - Расход памяти – мера **эффективности пространства**
 - Количество команд относительно количества обрабатываемых данных – мера **вычислительной эффективности**.

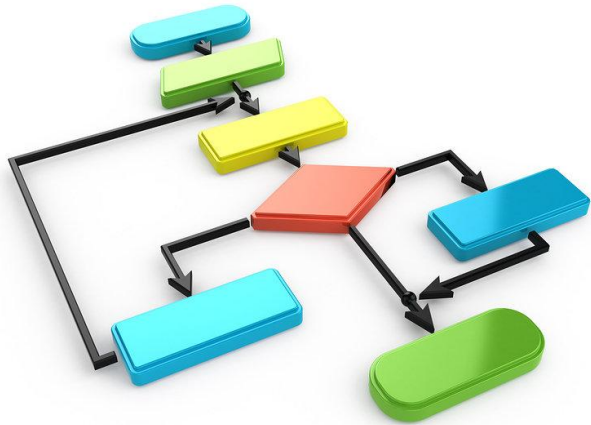
Сложность алгоритма



Сложность как характеристика связана с эффективностью:

- **Эффективный** алгоритм требует **приемлемое** время исполнения и разумную ресурсоемкость
- **Сложность** возрастает при **увеличении** времени исполнения алгоритма и (или) задействованных ресурсов
- Т.о. для одной и той же задачи **более сложный** алгоритм из нескольких характеризуется **меньшей эффективностью**.

Вычислительная сложность



- Составляющие:

- **Временная сложность** - отражает временные затраты на реализацию алгоритма
- **Емкостная сложность** – отражает объём требующейся алгоритму памяти

- Подходы к оценке:

- **Эмпирический анализ** (экспериментальный, практический):
 - Практический метод →
 - Теоретический метод
- **Асимптотический анализ.**

Практический метод (1/2)

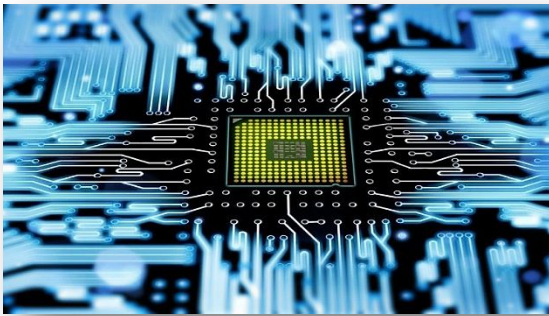
```
#include <ctime>
    const int n=100000000;
    double sum(int *x, int n);

int main(){
int x[n];
srand(time(0));
    for(int i=0;i<n;i++)
    x[i]=rand();
    sum(x,n);
    int t2=clock()/CLOCKS_PER_SEC;
    cout<<"при n="<<n<<"t= "<<t2<<"\n";
}
double sum(int *x, int n){
    int s=0;
    for(int i=0;i<n;i++){
        s=s+x[i];
    }
    return s;
}
```

Характеризуется **измеримыми параметрами:**

- **Временная сложность – во временных единицах** (микро-, милли-, секундах) или **количестве тактов процессора**
- **Емкостная сложность – в битах** (байтах и производных единицах), **минимальных аппаратных требованиях** и пр.

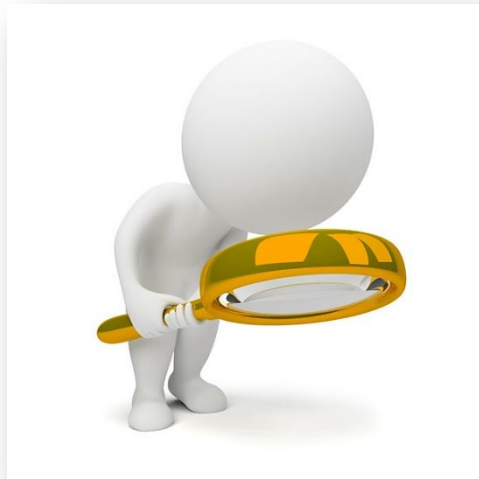
Практический метод (2/2)



Факторы, влияющие на оценку:

- Особенности **аппаратно-программной платформы**:
 - Характеристики **оборудования** (тактовая частота, объем ОЗУ и сверхоперативной памяти, размер файла подкачки)
 - Архитектура **программной среды** (многозадачность, алгоритм работы планировщика задач, особенности ОС)
- **Язык программирования** (транслятор)
- Квалификация (**опыт**) программиста
- В результате – практическая оценка **не является абсолютным показателем эффективности** (сложности)

Теоретический подход (1/2)



- Характеризует алгоритм **без привязки** к конкретному оборудованию, ПО и средствам реализации
- **Временная сложность** – в количестве операций, тактах работы машины Тьюринга и пр.
- **Емкостная сложность** определяется объёмом данных (входных, промежуточных, выходных), числом задействованных ячеек на ленте машины Тьюринга и пр.

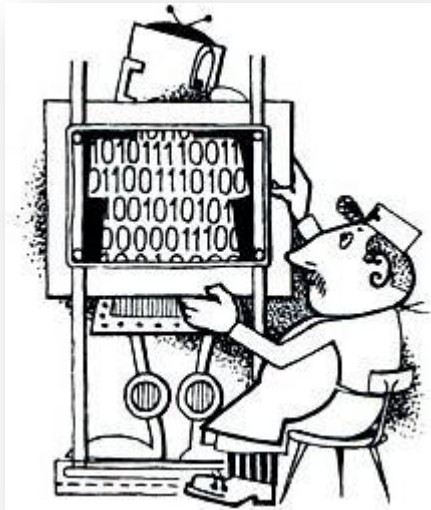
Теоретический подход (2/2)



Факторы, влияющие на оценку эффективности (сложности):

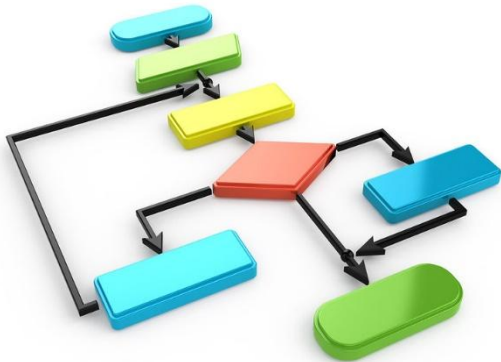
- **Объём входных данных (размер входа, размерность задачи)** – например, количество элементов в массиве на сортировку или длина строки и пр.
- **Метод решения** – например, тот или иной алгоритм сортировки

Модель вычислительной машины



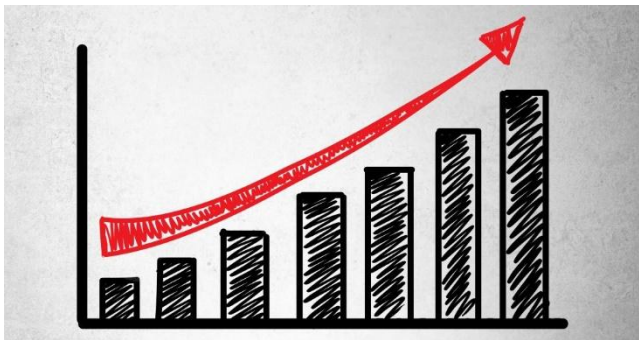
- Идеализированная
одноядерная
однопроцессорная
машина с памятью с
произвольным доступом
(RAM)
- **Команды** –
арифметические,
перемещения данных,
управляющие
- Каждая команда
выполняется за
определённое
фиксированное время.
→

Время работы алгоритма



- Тогда **время работы алгоритма** складывается из элементарных операций (**шагов**), которые необходимо выполнить
- Время выполнения различных строк **псевдокода** может отличаться, но пусть одна и та же i -я строка выполняется за константное время c_i . \rightarrow

Функция роста



- Время работы – это функция от объёма входных данных
- Пусть n – объём входных данных для некоторого алгоритма
- Тогда $T(n)$ – функция роста, показывает рост времени при увеличении входных данных
- **Скорость роста** (порядок роста) – функция более высокого порядка, главный член формулы $T(n)$.

Лучший, средний и худший случаи

- Пусть рассматривается алгоритм **проверки наличия числа** в некотором массиве
- Если этот массив упорядочен по возрастанию, то проверяем до первого элемента, который равен или больше искомого. В этом случае $T(n) < n$ (**лучший и средний случаи**)
- Однако в **худшем случае** (когда искомый элемент – последний в неотсортированном массиве) нужно просмотреть **все элементы**, и тогда $T(n) = n$.

9	3	7	5	1		1	3	7	5	9
---	---	---	---	---	--	---	---	---	---	---

Правила определения количества операторов в одной инструкции алгоритма



1. В строке алгоритма расположена одна простая команда – количество равно **1**.
2. Учитывается каждая команда в блоке команд.
3. Оператор цикла, в котором количество итераций зависит от n , оценивается через количество выполняемых сравнений в условии цикла: **$n+1$** .
4. Если тело цикла выполняется n раз, тогда количество операций в теле цикла после выполнения всех итераций = количество операторов тела цикла **$*n$** .

Пример 1. Среднее арифметическое всех положительных чисел массива A[n]

Оператор	Количество оператора	выполнений
Sum←0	1	
count ←0	1	
For i←1 to n do	n+1	
If(A[i]>0)	n	
sum←sum+A[i]	n	
count←count+1	n	
endIf		
od		
If (count≠0)	1	
return sum/count	1	
Else		
return -1	1	
elseif		

- $T(n)=1+1+(n+1)+n+n+n+1+1=4n+5$
- **Порядок роста:**
 - 4 и 5 – константы, рост будет определяться значением переменной n
 - Константы при определении порядка роста в выражении игнорируются
- Т.о. получаем **линейную зависимость** количества операций от количества элементов n.

Пример 2. Алгоритм с вложенным циклом (оба цикла зависят от n)

Оператор	Количество выполнений оператора
max ← A[1,1]	1
For i ← 1 to n do	n+1
For j ← 1 to m do	n*(m+1)
If(A[I,j]>max)	n*m
max ← A[i,j]	n*m
endIf	
od	
od	
return max	1

- Дана матрица размером $n*m$. Найти максимальный элемент.
- **Худший случай** – все числа положительные:
 - $T(n) = 1+(n+1)+3n*m+n+1 = 3nm+2n+3$, т.е. сложность зависит от $n*m$; если $m=n$, то получим n^2 .
- **Лучший случай** – положительных нет совсем:
 - Операторы под if не выполняются, и сложность вычислялась бы так же, только без одного слагаемого $n*m$: $T(n) = 1+(n+1)+2n*m+n+1 = 2nm+2n+3$, т.е. сложность также $n*m$.
- **Средний случай** (есть какое-то число положительных):
 - Тогда $2nm+2n+3 \leq T(n) \leq 3nm+2n+3$, и $2nm \leq T(n) \leq 3nm$.