

# **Как понять, что реализация не подходит?**

плохо написал – иди переписывай!

# Алгоритм

Вход: значения

Вычисления: код

Выход: значение

# Качество кода?

## Оформление кода:

- Единообразное оформление: стандарт?

## Читабельность кода:

- Чтобы понять мог реализацию другой программист ?

Сложный код: – длинное нечитабельное и не структурированное полотно ?

**НО** на кого это влияет ?

--пользователь **не видит код**

--ошибки искать и другому разобраться сложно!

# Какой алгоритм плохой?

## Медленно работает

- Быстрая машина - это дорого, а большинство пользовательских и рабочих точно не дорогие
- Мобильные устройства не настолько производительны, как компьютеры
- Интернет может сбоить, быть медленным
- Может требоваться быстрый отклик или мы теряем пользователей и деньги



## Много жрет памяти

(чаще всего оперативную)

- Не хватает места другим приложениям
- Не хватает места другим исполняемым приложениям: работают медленно
- Данных может быть очень много и дублирование данных или попытка хранения в быстрой памяти - проблема



# Оценка сложности алгоритма

По **ПАМЯТИ**: сколько занимает памяти

- 1 переменная - 1 пункт сложности
- Массив размером  $n$  -  $n$  пунктов сложности
- Строка длиной  $n$  -  $n$  пунктов сложности

По **ВРЕМЕНИ**: сколько операций нужно выполнить

- 1 операция - 1 пункт сложности: присвоить/сравнить
- 1 итерация =  $k$  операций в итерации -  $k$  пунктов сложности
- Условие =  $t$  логических операций и действий -  $t$  пунктов сложности

# Примеры

## Память:

```
int i, j, n;
```

```
int A[100];
```

```
char B[n];
```

3 переменных	3 пункта
Массив 10 элементов	10 пунктов
Строка n символов	n пунктов

## Время

```
for i = 1 to n {
```

```
    j++;
```

```
    k=n;
```

```
}
```

```
while (A[i] + A[i-1] > 0) {
```

```
    A[i + 1] = A[i];
```

```
    i = i - 1;
```

```
}
```

```
if (i<0 && (i/2=0))
```

```
then j=n;
```

Операции	Цикл
1 (i = 1 )	n - итераций
{1	2 пункта внутри итерации
1}	Итого: 2n+1

Операции	Цикл
{3	n - итераций
2	7 пункта внутри итерации
2}	Итого: 2n+1

Условие	подробнее
4	i/2=0 - 2 операции

# Примеры

```
int i, j, n=100;
```

```
int A[n]=1;
```

```
char B[n];
```

```
for i = 1 to n do  
    j++;
```

```
i=n;
```

```
while (A[i] + A[i-1] > 0) {  
    A[i + 1] = A[i];  
    i = i - 1;  
    if (i<0 && (i/2=0)) then j=n;  
}
```

Память = М	Время = Т
+3	+1
+n=100	+n
+n=100	
	+1
	{+n=100}
	+1
	{+3
	+2
	+2
	4+1 = 5}=12
	Итераций цикла: n
n+n+3	12n+1+n+n+1
<b>M(n)=2n+3</b>	<b>T(n)=14n+2</b>

# Примеры

```
int n=10;
int A[n];
int key, i, j;
```

```
for i = 0 to n-1 do
    A[j] =i;
```

```
for i = 0 to n-1 do
    for j = 0 to n-1 do
        A[i]=A[i]+A[j];
```

Память = М	Время = Т
1	+1
+n=10	
+3	
	+1
	{+n=10}
	+1
	{+1
	{+2} – n итераций}
	n - итераций
	=n(2n+1)
1+n+3	n(2n+1)+1+n+1+1
<b>M(n)=n+4</b>	<b>T(n)=2n<sup>2</sup>+2n+3</b>



# Функции сложности

$M(n)$  – сложность по памяти

$T(n)$  – сложность по времени

- Зависят от размера входных данных  $n$
- Размерность мы можем получать во время исполнения программы
- Функция как и реализация может быть простой ( $x=2$  «прямая», аналогично код без циклом и без массивов  $M(n) = 3 / T(n) = 6$ )

# Функция сложности – функция!

$M(n)$  – сложность по памяти

$T(n)$  – сложность по времени

*Вспомним типовые функции:*

- $x^2$
- $\log_2 x$
- $x * \log_2 x$
- $x^3$
- $a^x$
- *и так далее*

# Функция сложности

$M(n)$  – сложность по памяти более предсказуема и чаще всего определена размером входных данных

$T(n)$  – сложность по времени зависит в большей степени от программиста и его творческого порыва!

Далее будем чаще говорить именно о **временной сложности - сложности алгоритма!**

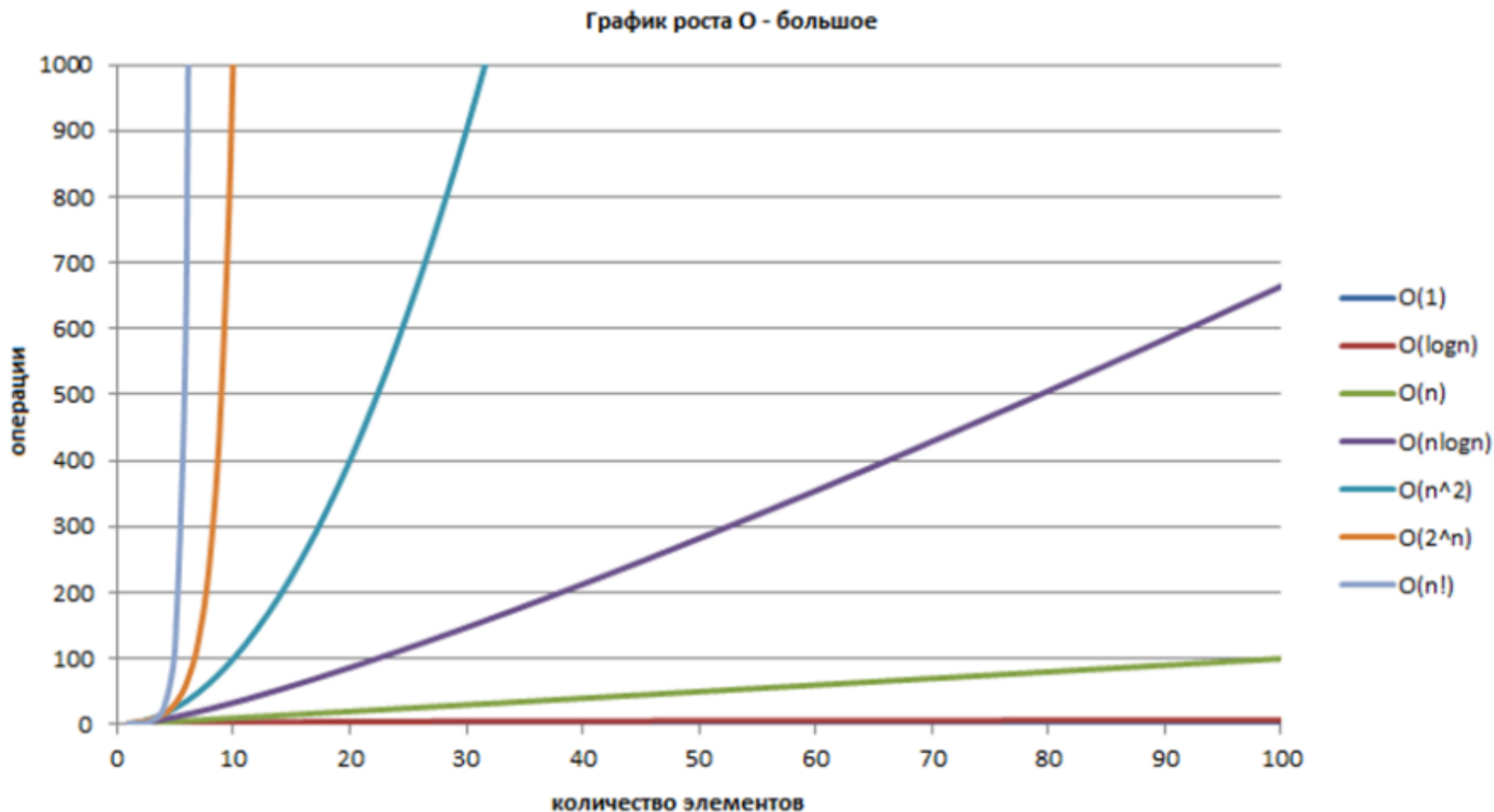
# Классификация алгоритмов

Обычно алгоритмы классифицируют в соответствии с их временной сложностью. Можно выделить следующие их типы:

1. **Постоянный** - сложность оценивается как  $O(1)$ .
2. **Логарифмический** - сложность оценивается как  $O(\log(n))$
3. **Линейный** - оценка равна  $O(n)$ .
4. **Квадратный** -  $O(n^2)$
5. **Кубический, полиномиальный** -  $O(n^3)$ ,  $O(n^m)$ .
6. **Экспоненциальный** -  $O(t^{p(n)})$ ,  $t$ - константа,  $p(n)$  - некоторая полиномиальная функция.
7. **Факториальный** -  $O(n!)$ . Обладает наибольшей временной сложностью среди всех известных типов.

# Классификация алгоритмов

Для того, чтобы визуально представить себе различную скорость роста функций, достаточно взглянуть на следующий график:



# Классификация алгоритмов

**Логарифмическая сложность** присуща алгоритмам, которые сводят большую задачу к набору меньших задач, уменьшая на каждом шаге размер задачи на постоянную величину. Например, двоичный поиск в массиве, когда на каждом шаге размер массива сокращается вдвое.

**Линейное время** выполнения свойственно тем алгоритмам, в которых осуществляется небольшая обработка каждого входного элемента.

**Оценка  $n \log(n)$**  возникает в тех случаях, когда алгоритм решает задачу, разбивая её на меньшие подзадачи и решая их независимо друг от друга, а затем объединяя решение.

# Классификация алгоритмов

**Квадратичное** время выполнения свойственно алгоритмам, обрабатывающим все пары элементов данных.

**Кубическое** время соответствует алгоритмам, которые обрабатывают все тройки элементов данных.

**Экспоненциальное и факториальное** время присуще алгоритмам, которые выполняют перебор всевозможных сочетаний элементов.

# Показательный пример

```
int n=10;  
int A[n];  
int key, i, j;
```

```
for i = 1 to n do  
    A[j] = i;  
  
for i = 1 to n do  
    for j = 1 to i do  
        A[i] = A[i] + A[j];
```

**$T(n)$**  – как описать?

*Распишем итерации  
первого цикла*

Итерация №	Операций
1	1
2	2
3	3
4	4
5	5
...	...
i=n	n



# Вычисление

```
for i = 1 to n do  
    for j = 1 to i do  
        A[i]=A[i]+A[j];
```

С помощью суммы арифметической прогрессии:

$$T(n) = (1+n)n/2 = \\ = (n^2+n)/2$$

- Внутренний цикл **for** зависит от внешнего
- Как мы видим (см таблицу ранее) число операций в итерации внутреннего цикла изменяется с шагом +1
- Описать число операций двух циклов можно с помощью суммы арифметической прогрессии

# Пример

```
int n=10;  
int A[N];  
int key, i, j;
```

//получаем массив A

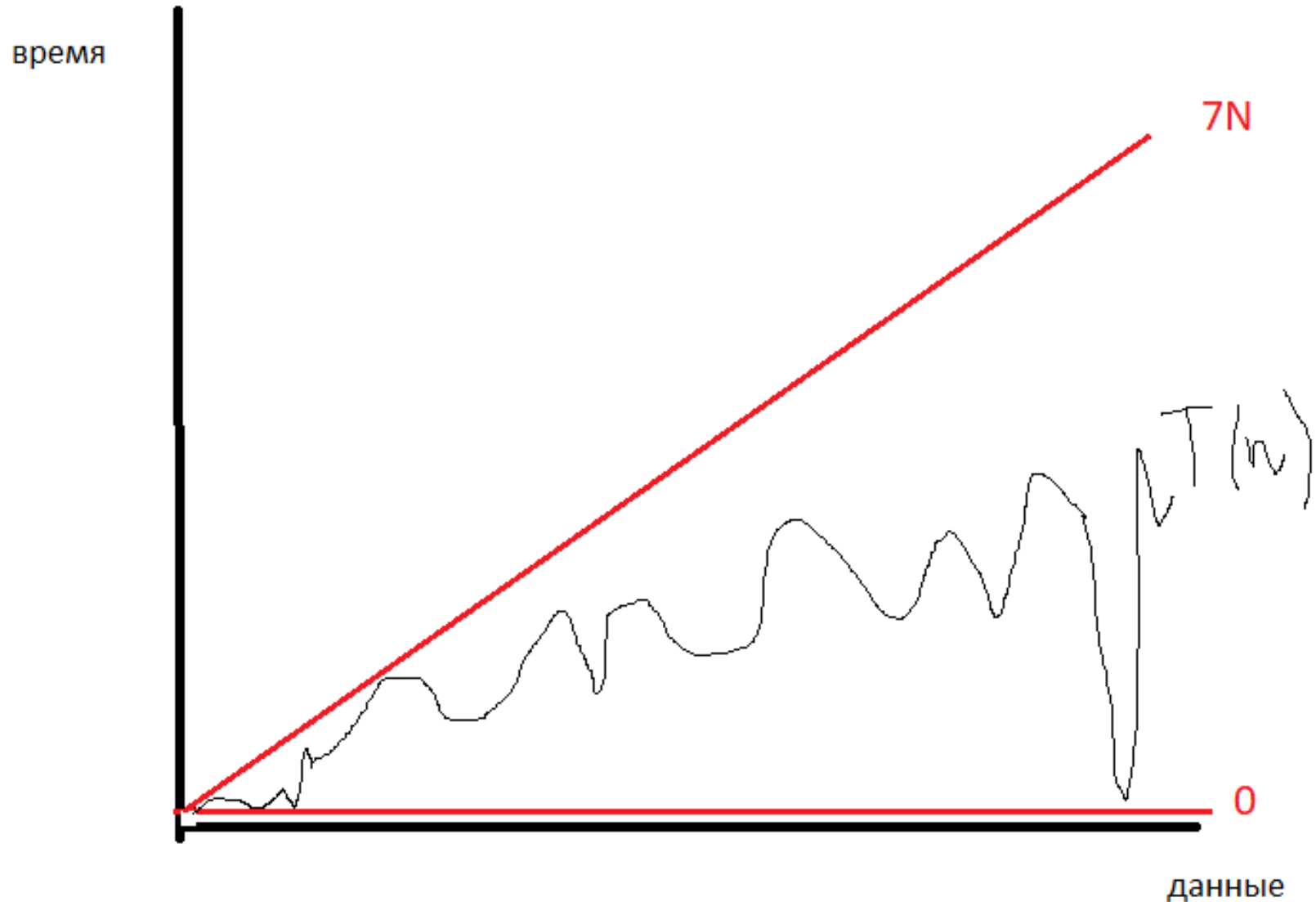
```
i=N;  
while (A[i] + A[i-1] > N){  
    A[i - 1] = A[i] ;  
    i = i - 1;  
}
```

- Не зная какой будет массив A, вывести функцию времени невозможно

Как тогда оценить сложность по времени?

- **Минимально:** цикл не выполнится ни разу, значит условие проверится 1 раз=3 операции
- **Максимально:** цикл выполнится N раз = 7N

# Иллюстрация оценки



Максимальная и минимальная оценки функции похожи на **асимптоты**

# Асимптотическая оценка сложности

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Чаще всего он представлен в виде O-нотации (*от нем. «Ordnung» — порядок*):  **$O(f(n))$** , где  **$f(n)$**  — формула, выражающая сложность алгоритма. В формуле может присутствовать переменная  **$n$** , представляющая размер входных данных.

$$T(n)=2n^2+2n+3$$

порядок роста  $n^2$  или  $O(n^2)$

$$T(n)=1290n^3+12n^2-3586$$

порядок роста  $n^3$  или  $O(n^3)$

$$T(n)=2n^2 \log n + 142n^2 - n$$

порядок роста  $n^2 \log n$  или  $O(n^2 \log n)$

$$T(n)=1290n^3 + 2n^2 \log n - 142n^2 - n + 386$$

порядок роста  $n^3$  или  $O(n^3)$

# Асимптотическая оценка сложности

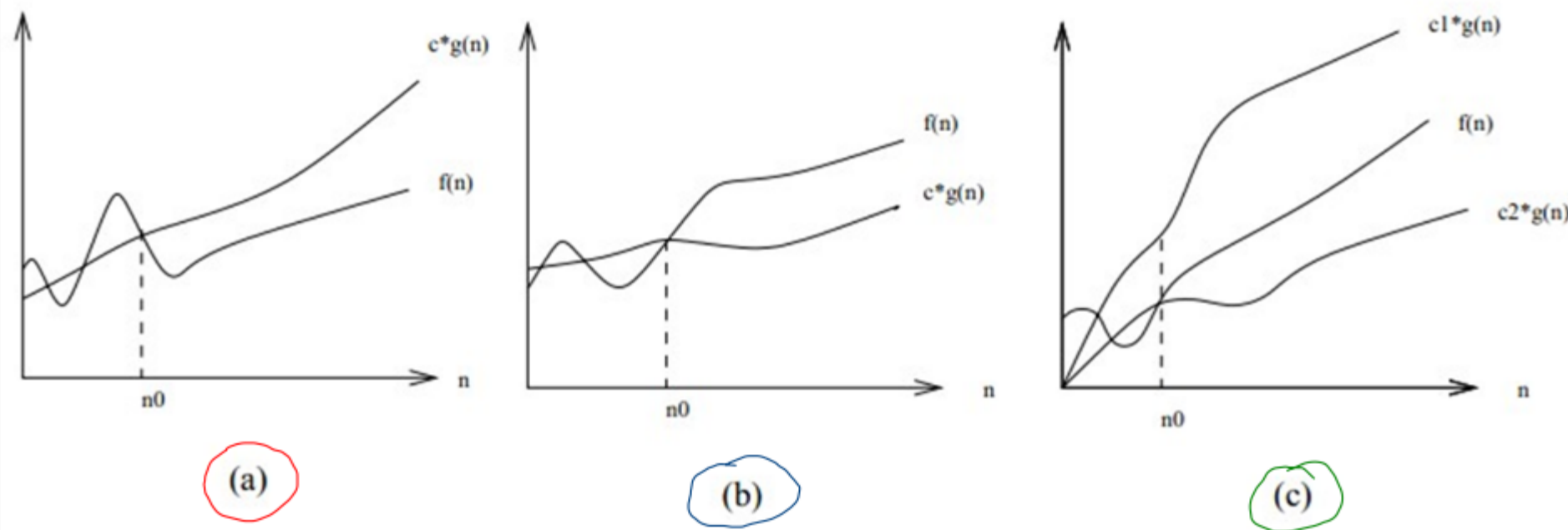


Figure 2.3: Illustrating the big (a)  $O$ , (b)  $\Omega$ , and (c)  $\Theta$  notations

$$(a) O(g(n)) = \left\{ T(n) : \exists c > 0, n_0 > 0 \right. \\ \left. \forall n > n_0 \quad 0 \leq T(n) \leq c \cdot g(n) \right\}$$

сверху

$$(b) \Omega(g(n)) = \left\{ T(n) : \exists c > 0, n_0 > 0 \right. \\ \left. \forall n > n_0 \quad 0 \leq c \cdot g(n) \leq T(n) \right\}$$

снизу

$$(c) \Theta(g(n)) = \left\{ T(n) : \exists c_1 > 0, c_2 > 0, n_0 > 0 \right. \\ \left. \forall n > n_0 \quad 0 \leq c_2 \cdot g(n) \leq T(n) \leq c_1 \cdot g(n) \right\}$$

точно

# Свойства O

$$T_1(n) = O(f_1(n))$$

$$T_2(n) = O(g_2(n))$$

C - константа!

1) Сложность суммы  $T_1 + T_2 = O(\max(f_1(n), g_2(n)))$   
большая из функций порядка роста

2) Сложность произведения  $T_1 \cdot T_2 = O(f_1(n) \cdot g_2(n))$   
произведение порядков

3) Умножение на константу  $C \cdot T_1 = O(f_1(n))$   
при произведении на константу её можно отбросить  
 $\Rightarrow$  не берем в расчёт

4) Сумма с константой  $T_1 + C = O(f_1(n))$   
при сумме с константой  $\rightarrow$  её можно отбросить

5) Теорема о связи  $\Theta, \Omega, O$

$$T(n) \sim \Theta(g(n)) \Leftrightarrow \begin{cases} T(n) \sim \Omega(g(n)) \\ T(n) \sim O(g(n)) \end{cases}$$

[точная оценка]  $\Leftrightarrow$  когда совпадает с оценкой  
снизу и сверху!

# Зачем оценивать алгоритмы?

Устройство 1:

$10^9$  операций/сек

Объем данных:  $N = 10^7$

Алгоритм:  $n^2$

Время:  $10^7 * 10^7 / 10^9$   
 $= 10^5$  сек **> суток**

Устройство 2:

$10^7$  операций/сек

Объем данных:  $N = 10^7$

Алгоритм:  $n * \log n$

Время:  $10^7 * \log 10^7 /$   
 $10^7 = 23,25$  сек **<**  
**МИНУТЫ**

# Зачем оценивать алгоритмы?

Время выполнения алгоритма с определённой сложностью в зависимости от размера входных данных при скорости  $10^6$  операций в секунду:

размер сложность	10	20	30	40	50	60
$n$	0,00001 сек.	0,00002 сек.	0,00003 сек.	0,00004 сек.	0,00005 сек.	0,00005 сек.
$n^2$	0,0001 сек.	0,0004 сек.	0,0009 сек.	0,0016 сек.	0,0025 сек.	0,0036 сек.
$n^3$	0,001 сек.	0,008 сек.	0,027 сек.	0,064 сек.	0,125 сек.	0,216 сек.
$n^5$	0,1 сек.	3,2 сек.	24,3 сек.	1,7 минут	5,2 минут	13 минут
$2^n$	0,0001 сек.	1 сек.	17,9 минут	12,7 дней	35,7 веков	366 веков
$3^n$	0,059 сек.	58 минут	6,5 лет	3855 веков	$2 \times 10^8$ веков	$1,3 \times 10^{13}$ веков



