

Лекция 1

Анализ эффективности алгоритмов

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Весенний семестр, 2016

Литература

- **[CLRS]** Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К. **Алгоритмы: построение и анализ.** – 3-е изд. – М.: Вильямс, 2013
- **[Levitin]** Левитин А.В. **Алгоритмы: введение в разработку и анализ.** – М.: Вильямс, 2006. – 576 с.
- **[Aho]** Ахо А.В., Хопкрофт Д., Ульман Д.Д. **Структуры данных и алгоритмы.** – М.: Вильямс, 2001. – 384 с.
- **[DSABook]** Курносов М.Г. Введение в структуры и алгоритмы обработки данных. - Новосибирск: Автограф, 2015. - 179 с.
- Дасгупта С., Пападимитриу Х., Вазирани У. **Алгоритмы.** - М.: МЦНМО, 2014. - 320 с.
- Кормен Т.Х. **Алгоритмы: Вводный курс.** - М.: Вильямс, 2014. - 208 с.
- Седжвик Р. **Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск.** – К.: ДиаСофт, 2001. – 688 с.
- Макконнелл Дж. **Основы современных алгоритмов.** – 2-е изд. – М.: Техносфера, 2004. – 368 с.
- Скиена С.С. **Алгоритмы. Руководство по разработке.** – 2-е изд. – СПб: БХВ, 2011 – 720 с.
- Керниган Б.В., Пайк Р. **Практика программирования.** – СПб.: Невский Диалект, 2001. – 381 с.
- Кнут Д. **Искусство программирования.** Том {1, 3}, 3-е изд. - М.: Вильямс, 2010.



Решение задачи на компьютере

1. **Постановка задачи** (problem statement) – точная формулировка условий задачи с описанием её *входных* (input) и *выходных* данных (output)
2. **Разработка алгоритма** решения задачи (algorithm design)
3. **Доказательство корректности алгоритма** и анализ его эффективности
4. **Реализация алгоритма** на языке программирования (implementation)
5. **Выполнение программы** для получения требуемого результата

Понятие алгоритма

- **Алгоритм (algorithm)** – это конечная последовательность инструкций *исполнителю*, в результате выполнения которых обеспечивается получение из входных данных требуемого выходного результата (решение *задачи*)
- **Уточнения**
 - ❑ Алгоритм должен описываться на формальном языке исполнителя, исключающем неоднозначность толкования предписаний
 - ❑ Множество инструкций исполнителя конечно
 - ❑ Запись алгоритма на формальном языке называется *программой* (program)

Свойства алгоритмов

- **Дискретность** – алгоритм представляется как последовательность инструкций исполнителя. Каждая инструкция выполняется только после того, как закончилось выполнение предыдущего шага
- **Конечность (результативность, финитность)** – алгоритм должен заканчиваться после выполнения конечного числа инструкций
- **Массовость** – алгоритм решения задачи должен быть применим для некоторого класса задач, различающихся лишь значениями входных данных
- **Детерминированность (определенность)** – каждый шаг алгоритма должен быть точно определен – записан на формальном языке исполнителя. Детерминированность обеспечивает *одинаковость результата, получаемого при многократном выполнении алгоритма, на одном и том же наборе входных данных*

Задача поиска максимального элемента

- **вход:** последовательность из n чисел (a_1, a_2, \dots, a_n)
- **выход:** номер i элемента a_i , имеющего наибольшее значение:

$$a_i \geq a_j, \quad \forall j \in \{1, 2, \dots, n\}.$$

Массив $a[0:24]$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
180	155	193	176	159	169	172	195	201	160	167	180	177	174	179	187	166	193	183	175	169	170	157	199	187

Задача поиска максимального элемента

- **вход:** последовательность из n чисел (a_1, a_2, \dots, a_n)
- **выход:** номер i элемента a_i , имеющего наибольшее значение:

$$a_i \geq a_j, \quad \forall j \in \{1, 2, \dots, n\}.$$

Массив $a[0:24]$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
180	155	193	176	159	169	172	195	201	160	167	180	177	174	179	187	166	193	183	175	169	170	157	199	187

↑
 $i = 8$

Алгоритм линейного поиска

Массив $a[0:24]$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
180	155	193	176	159	169	172	195	201	160	167	180	177	174	179	187	166	193	183	175	169	170	157	199	187

```
function Max(a[1..n])  
    maxi = 1  
    for i = 2 to n do  
        if a[i] > a[maxi] then  
            maxi = i  
        end for  
    return maxi  
end function
```


Алгоритм линейного поиска

Массив $a[0:24]$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
180	155	193	176	159	169	172	195	201	160	167	180	177	174	179	187	166	193	183	175	169	170	157	199	187

```
function Max(a[1..n])  
    maxi = 1  
    for i = 2 to n do  
        if a[i] > a[maxi] then  
            maxi = i  
        end for  
    return maxi  
end function
```

Свойства алгоритма

- Дискретность
- Конечность
- Массовость
- Детерминированность

Показатели эффективности алгоритмов

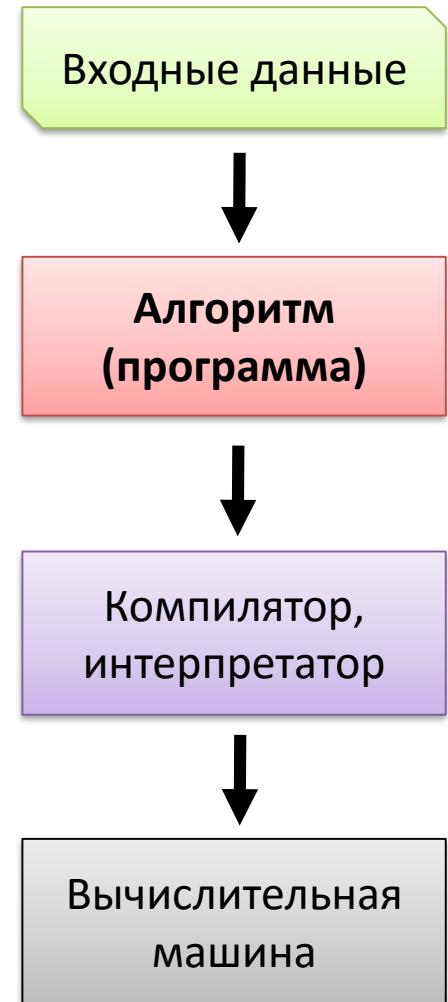
- **Количество выполняемых операций –**
временная эффективность (time efficiency),
показывает на сколько быстро работает алгоритм
- **Объем потребляемой памяти –**
пространственная эффективность (space efficiency),
отражает максимальное количество памяти требуемой
для выполнения алгоритма
- Показатели эффективности позволяют:
 - ❑ Оценивать потребности алгоритма в вычислительных ресурсах:
процессорном времени, памяти, пропускной способности сети
 - ❑ Сравнивать алгоритмы между собой

Анализ времени выполнения алгоритмов

- Что влияет на время выполнения алгоритма (программы)?

1. Размер входных данных
2. Качество реализации алгоритма на языке программирования
3. Качество скомпилированного кода
4. Производительность вычислительной машины

- Для большинства алгоритмов количество выполняемых ими операций напрямую зависит от *размера входных данных*
- Например, в алгоритме поиска наибольшего элемента время выполнения определяется не значениями в массиве, а его длиной n



Размер входных данных алгоритма

- У каждого алгоритма есть параметры, определяющие размер его входных данных
- Поиск наименьшего элемента в массиве:
 n — количество элементов в массиве
- Алгоритм умножения двух матриц:
количества строк m и столбцов n в матрицах
- Сравнения двух строк:
 s_1, s_2 — длина первой и второй строк
- Поиск кратчайшего пути в графе между парой вершин:
 n, m — количество вершин и ребер в графе

Количество операций алгоритма

- Количество операций алгоритма можно выразить как функцию от размера его входных данных: $T(n)$, $T(s_1, s_2)$, $T(n, m)$
- В качестве исполнителя будем использовать модель однопроцессорной вычислительной машины с произвольным доступом к памяти (Random Access Machine – RAM)
 - Машина обладает неограниченной памятью
 - Для выполнения арифметических и логических операций (+, −, *, /, %) требуется один временной шаг – такт процессора
 - Обращение к оперативной памяти для чтения или записи занимает один временной шаг
 - Выполнение условного перехода (*if-then-else*) требует вычисления логического выражения и выполнения одной из ветвей *if-then-else*
 - Выполнение цикла (*for, while, do*) подразумевает выполнение всех его итераций

Суммирование элементов массива

```
function SumArray(a[1..n])  
    sum = 0  
    for i = 1 to n do  
        sum = sum + a[i]  
    end for  
    return sum  
end function
```

- Время работы алгоритма *SumArray* зависит только от размера n массива
- Выразим число $T(n)$ операций, выполняемых алгоритмом

Суммирование элементов массива

```
function SumArray(a[1..n])  
    sum = 0                // Запись в память  
    for i = 1 to n do      // i <= n, jle, i++  
        sum = sum + a[i]   // 2 чтения, +, запись  
    end for  
    return sum             // Возврат значение 1 оп.  
end function
```

- Выразим число $T(n)$ операций, выполняемых алгоритмом

$$T(n) = 1 + 4n + 1 = 4n + 2$$

- Можно ограничиться подсчётом только *базовых операций* – наиболее важных операций, от которых зависит время выполнения алгоритма
- В алгоритме *SumArray* – это операция «+»

Линейный поиск элемента в массиве

```
function LinearSearch(a[1..n], x)
    for i = 1 to n do
        if a[i] = x then
            return i
        end if
    end for
    return -1
end function
```

- Количество операций алгоритма *LinearSearch* может существенно отличаться для одного и того же размера n входных данных
- Базовая операция алгоритма – это сравнение « $a[i] = x$ »
- Рассмотрим три возможных случая:
 - ☐ Лучший случай (best case)
 - ☐ Худший случай (worst case)
 - ☐ Средний случай (average case)

Линейный поиск элемента в массиве

```
function LinearSearch(a[1..n], x)
  for i = 1 to n do
    if a[i] = x then
      return i
  end for
  return -1
end function
```

LinearSearch(a[], 6, 45)

1	2	3	4	5	6
45	21	93	4	22	67

- **Лучший случай** (best case) – это экземпляр задачи (набор входных данных), на котором алгоритм выполняет *наименьшее* число операций
- Для *LinearSearch* – это входной массив, первый элемент которого содержит искомое значение x (одно сравнение)

$$T_{Best}(n) = 1$$

Линейный поиск элемента в массиве

```
function LinearSearch(a[1..n], x)
  for i = 1 to n do
    if a[i] = x then
      return i
  end for
  return -1
end function
```

LinearSearch(a[], 6, 67)

1	2	3	4	5	6
45	21	93	4	22	67

- **Худший случай** (worst case) – это экземпляр задачи, на котором алгоритм выполняет *наибольшее* число операций
- Для *LinearSearch* – это массив, в котором отсутствует искомый элемент или он расположен в последней ячейке (n сравнений)

$$T_{Worst}(n) = n$$

Линейный поиск элемента в массиве

```
function LinearSearch(a[1..n], x)
  for i = 1 to n do
    if a[i] = x then
      return i
  end for
  return -1
end function
```

LinearSearch(a[], 6, x)

1	2	3	4	5	6
45	21	93	4	22	67

- **Средний случай** (average case) – это “средний” экземпляр задачи, набор “усреднённых” входных данных
- В среднем случае оценивается математическое ожидание количества операций, выполняемых алгоритмом
- Не всегда очевидно, какие входные данные считать «усреднёнными» для задачи

Линейный поиск элемента в массиве

- Обозначим через $p \in [0, 1]$ вероятность присутствия искомого элемента x в массиве
- Будем считать, что искомый элемент с одинаковой вероятностью p / n может находиться в любой из n ячеек массива
- В общем случае, если элемент x расположен в ячейке i , то это требует выполнения i сравнений
- Запишем математическое ожидание (среднее значение) числа операций, выполняемых алгоритмом

$$T_{Average}(n) = 1 \frac{p}{n} + 2 \frac{p}{n} + \dots + i \frac{p}{n} + \dots + n \frac{p}{n}$$

Линейный поиск элемента в массиве

- В нашей оценке мы должны учесть и тот факт, что искомое значение x с вероятностью $1 - p$ может отсутствовать в массиве
- Тогда формула примет следующий вид

$$\begin{aligned} T_{Average}(n) &= \frac{p}{n} [1 + 2 + \dots + n] + (1 - p)n = \\ &= \frac{p}{n} \left[\frac{n^2 + n}{2} \right] + (1 - p)n \\ &= p \left(\frac{n + 1}{2} \right) + (1 - p)n \end{aligned}$$

- Вывод: если искомый элемент присутствует в массиве ($p = 1$), то в среднем требуется выполнить $(n + 1) / 2$ операции сравнения для его нахождения

Какой случай рассматривать?

- При анализе алгоритмов мы будем уделять основное внимание времени работы алгоритмов в *худшем случае* – максимальному времени работы на всех наборах входных данных
- При возможности, будем строить оценки эффективности для среднего случая
- Понятия *количество операций алгоритма* и *время выполнения алгоритма* (execution time) мы будем использовать как синонимичные

Анализ сложности алгоритмов

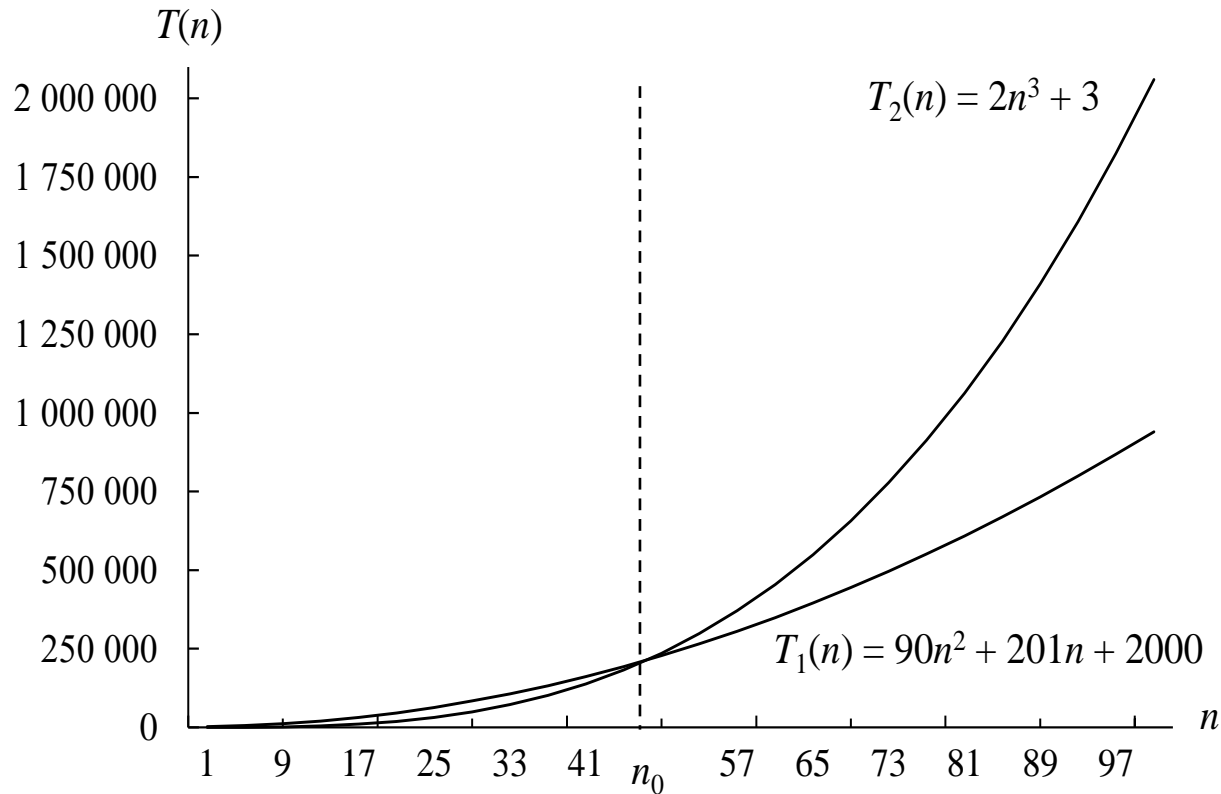
- Известно количество операций, выполняемых двумя алгоритмами, решающими одну задачу:

- $T_1(n) = 90n^2 + 201n + 2000$

- $T_2(n) = 2n^3 + 3$

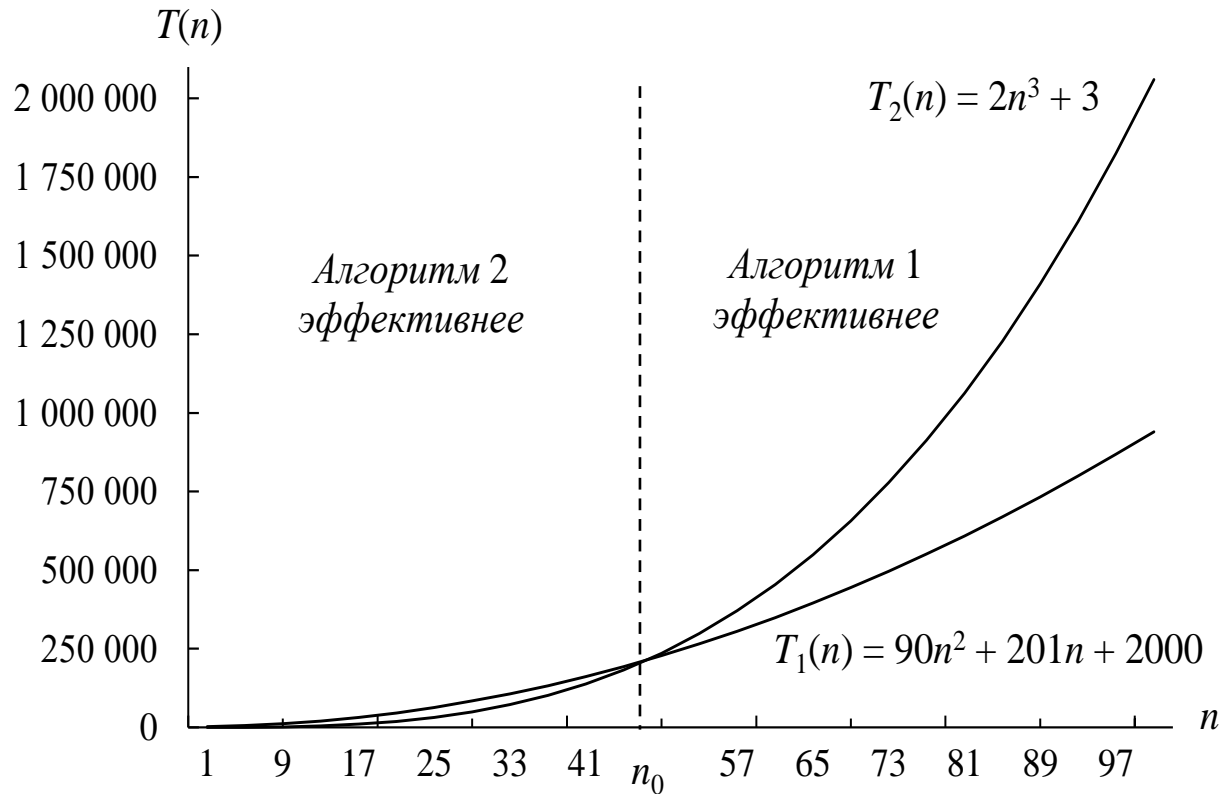
- Какой из алгоритмов предпочтительнее использовать на практике?

Анализ сложности алгоритмов



Мы можем найти такое значение n_0 при котором происходит пересечение функций $T_1(n)$ и $T_2(n)$, и на основе n_0 отдавать предпочтение тому или иному алгоритму

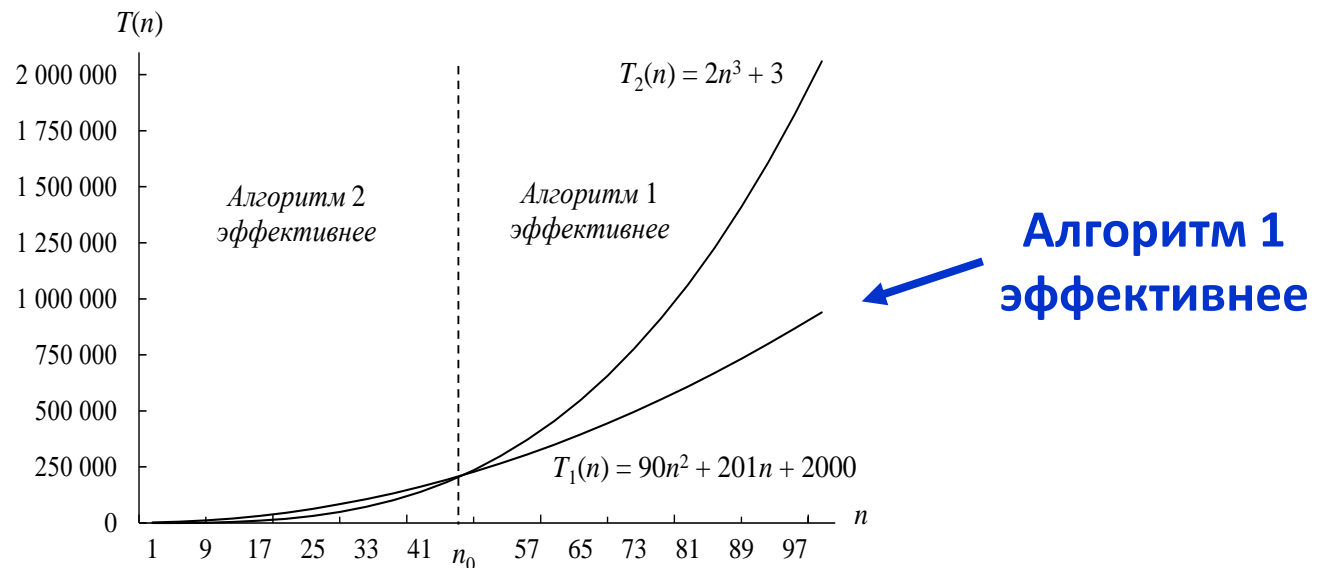
Анализ сложности алгоритмов



Мы можем найти такое значение n_0 при котором происходит пересечение функций $T_1(n)$ и $T_2(n)$, и на основе n_0 отдавать предпочтение тому или иному алгоритму

Анализ сложности алгоритмов

- Если известно, что на вход будут поступать данные небольших размеров, то вопрос о выборе эффективного алгоритма не является первостепенным – можно использовать самый “простой” алгоритм
- **Вопросы, связанные с эффективностью алгоритмов, приобретают смысл при больших размерах входных данных – при $n \rightarrow \infty$**



Скорость роста функций

- Скорость роста (rate of growth) или **порядка роста** (order of growth) функций $T(n)$ определяется её старшим, доминирующим членом

$\lg n$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n	$n!$
0	0	1	0	1	2	1
0.3	1	2	2	4	4	2
0.5	1.6	3	5	9	8	6
0.6	2.0	4	8	16	16	24
0.7	2.3	5	12	25	32	120
0.78	2.6	6	16	36	64	720
0.85	2.8	7	20	49	128	5 040
0.90	3	8	24	64	256	40 320
0.95	3.2	9	29	81	512	362 880
1	3.3	10	33	100	1024	3 628 800
3	10	1 000	9 966	1 000 000		
4	13.3	10 000	132 877	100 000 000		
5	16.6	100 000	1 660 964	10 000 000 000		
6	19.9	1 000 000	19 931 569	1 000 000 000 000		

Скорость роста функций

Пусть процессор выполняет одну операцию за 0.000000001 сек.
(тактовая частота 1 GHz)

$\lg n$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n	$n!$
0	0	1	0	1	2	1
0.3	1	2	2	4	4	2
0.5	1.6	3	5	9	8	6
0.6	2.0	4	8	16	16	24
0.7	2.3	5	12	25	32	120
0.78	2.6	6	16	36	64	720
0.85	2.8	7	20	49	128	5 040
0.90	3	8	24	64	256	40 320
0.95	3.2	9	29	81	512	362 880
1	3.3	10		100	1024	3 628 800
3	10	1 000		1 000 000		
4	13.3	10 000		100 000 000		
	6.6	100 000	1 660 964	10 000 000 000		
	19.9	1 000 000	19 931 569	1 000 000 000 000		

1000 с =
16 мин 40 с

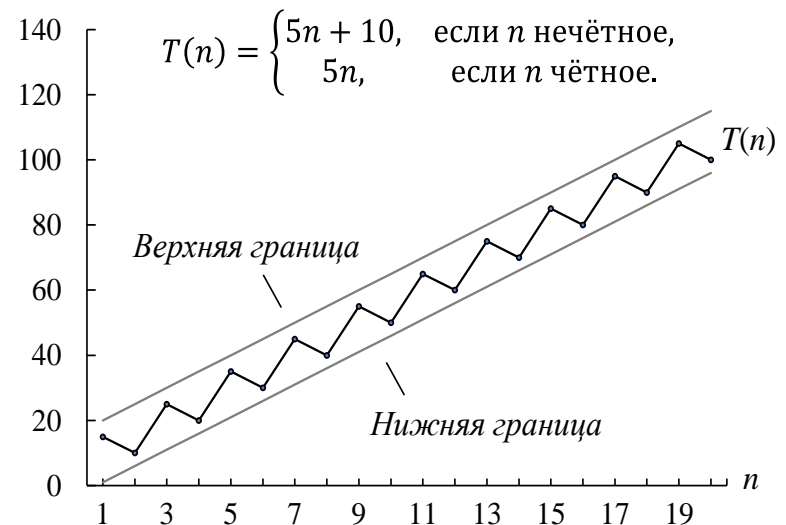
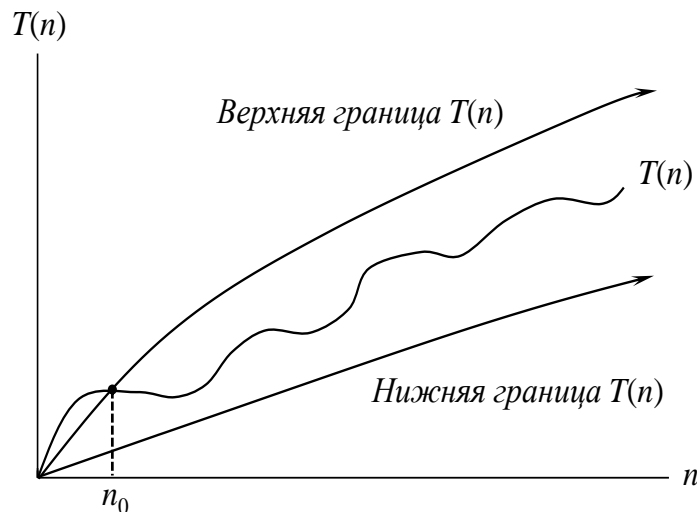
0,001 с

Анализ сложности алгоритмов

- Нам требуется математический аппарат, дающий ответ на вопрос какая из функций растёт быстрее при $n \rightarrow \infty$
 - $T_1(n) = 90n^2 + 201n + 2000$
 - $T_2(n) = 2n^3 + 3$
- Ответы на эти вопросы даёт **асимптотический анализ** (asymptotic analysis), который позволяет оценивать скорость роста функций $T(n)$ при стремлении размера входных данных к бесконечности (при $n \rightarrow \infty$)

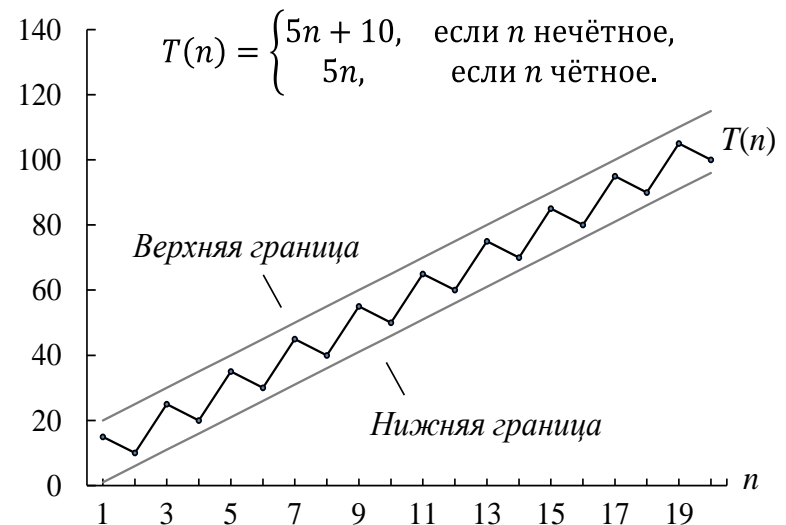
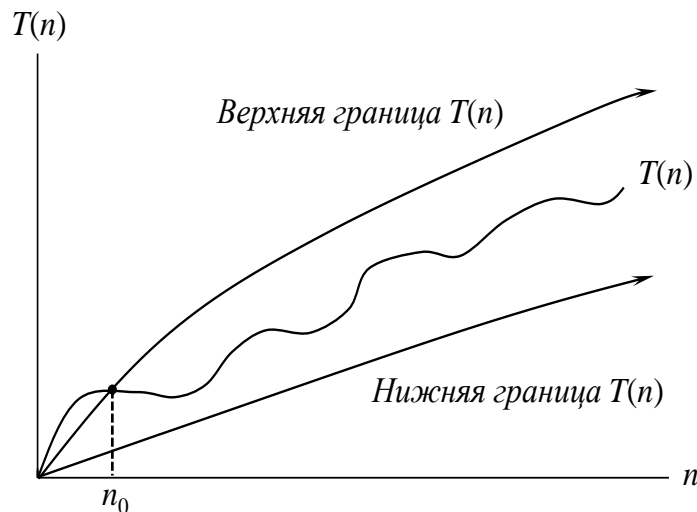
Асимптотические обозначения

- Как правило, функция времени $T(n)$ выполнения алгоритма имеет большое количество локальных экстремумов – неровный график с выпуклостями и впадинами
- Проще работать с верхней и нижней оценками (границами) времени выполнения алгоритма



Асимптотические обозначения

- В теории вычислительной сложности алгоритмов (computational complexity theory) для указания границ функций $T(n)$ используют **асимптотические обозначения**: O (о большое), Ω (омега большое), Θ (тета большое), а также o (о малое), ω (омега малое)



Асимптотические обозначения

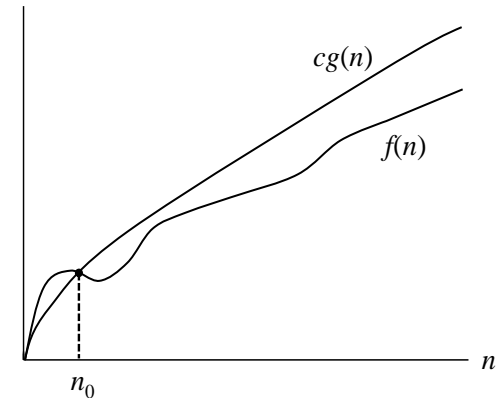
- Далее будем считать, что областью определения функции $f(n)$ и $g(n)$, которые выражают число операций алгоритма, является множество неотрицательных целых чисел:
 $n \in \{0, 1, 2, \dots\}$
- Функции $f(n)$ и $g(n)$ являются *асимптотически неотрицательными* – при больших значениях n они принимают значения большие или равные нулю

O-обозначение (о большое)

- Пусть $f(n)$ – это количество операций выполняемых алгоритмом
- O-обозначение $f(n) = O(g(n))$

$$f(n) \in O(g(n)) = \left\{ \begin{array}{l} \exists c > 0, n_0 \geq 0: \\ 0 \leq f(n) \leq cg(n), \forall n \geq n_0 \end{array} \right\}$$

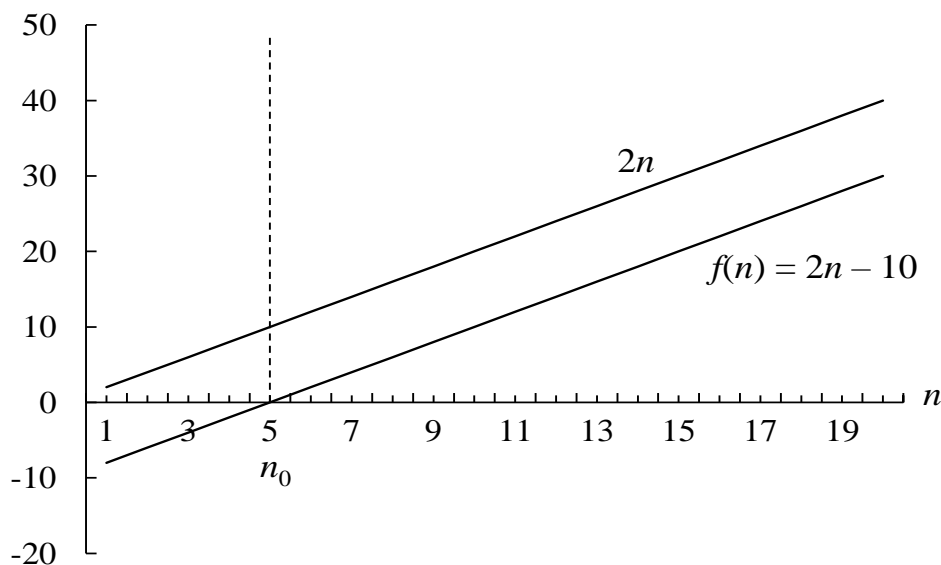
- Существуют константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$ такие, что $f(n) \leq c \cdot g(n)$ для всех $n \geq n_0$
- Функция $f(n)$ **ограничена сверху** функцией $g(n)$ с точность до постоянного множителя
- Читается как “ f от n есть о большое от g от n ”
- Используется чтобы показать, что время работы не быстрее чем функция $g(n)$



**Асимптотическая верхняя граница (asymptotic upper bound)
для функции $f(n)$**

O-обозначение (о большое)

- Докажем, что $2n - 10 = O(n)$
- Для этого требуется найти константы $c > 0$ и $n_0 \in \{0, 1, 2, \dots\}$
- **Доказательство:** возьмём $c = 2$, $n_0 = 5$
- Эти значения обеспечивают выполнение неравенства $0 \leq 2n - 10 \leq 2n$ для любых $n \geq 5$
- Прямая $2n$ проходит выше прямой $2n - 10$



O-обозначение (о большое)

- $3n^2 + 100n + 8 = O(n^2)$

Для доказательства возьмём $c = 4$, $n_0 = 101$, при любых $n \geq 101$ справедливо неравенство $0 \leq 3n^2 + 100n + 8 \leq 4n^2$

- $3n^2 + 100n + 8 = O(n^3)$

Возьмём $c = 1$, $n_0 = 12$, для любых $n \geq 12$ справедливо $0 \leq 3n^2 + 100n + 8 \leq n^3$

- $3 \cdot 0.000001n^3 \neq O(n^2)$

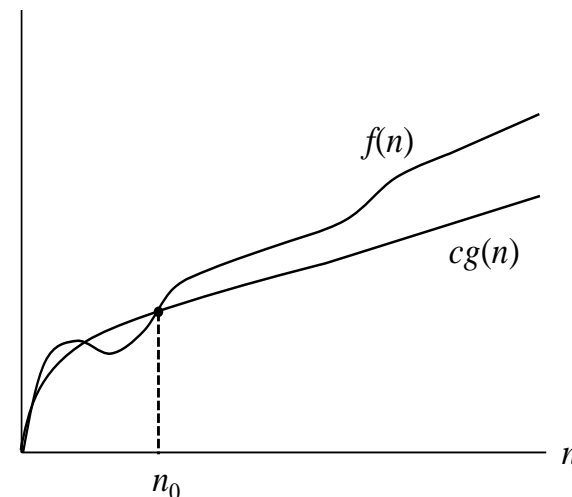
Так не существует констант $c > 0$ и n_0 , которые обеспечивают выполнения неравенств из определения O . Так для любых $c > 0$ и $n \geq c / 0.000001$ имеет место неравенство $0.000001n^3 > cn^2$

Ω-обозначение (омега большое)

- Пусть $f(n)$ – это количество операций выполняемых алгоритмом
- Ω-обозначение $f(n) = \Omega(g(n))$

$$f(n) \in \Omega(g(n)) = \left\{ \begin{array}{l} \exists c > 0, n_0 \geq 0: \\ 0 \leq cg(n) \leq f(n), \forall n \geq n_0 \end{array} \right\}$$

- Функция $f(n)$ **ограничена снизу** функцией $g(n)$ с точность до постоянного множителя
- Используется чтобы показать, что функция (время работы алгоритма) растет не медленнее чем функция $g(n)$
- Читается как
“ f от n есть омега большое от g от n ”



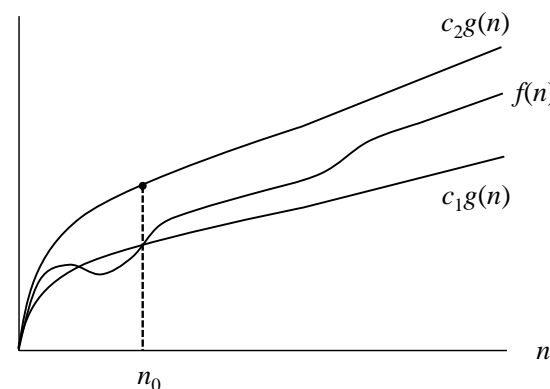
Асимптотическая нижняя граница (asymptotic lower bound) для функции $f(n)$

Θ-обозначения (тета большое)

- Пусть $f(n)$ – это количество операций выполняемых алгоритмом
- Θ-обозначение $f(n) = \Theta(g(n))$

$$f(n) \in \Theta(g(n)) = \left\{ \begin{array}{l} \exists c_1 > 0, c_2 > 0, n_0 \geq 0: \\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \\ \forall n \geq n_0 \end{array} \right\}$$

- Функция $f(n)$ **ограничена
снизу и сверху** функцией $g(n)$
с точность до постоянного множителя



Пример

- Пусть $f(n) = \frac{1}{2}n^2 - 3n$ (количество операций в алгоритме)
- **Докажем**, что $f(n) = \Theta(n^2)$
- **Доказательство** следует из определения Θ -обозначения:

$$\exists c_1 > 0, c_2 > 0, n_0 \geq 0:$$

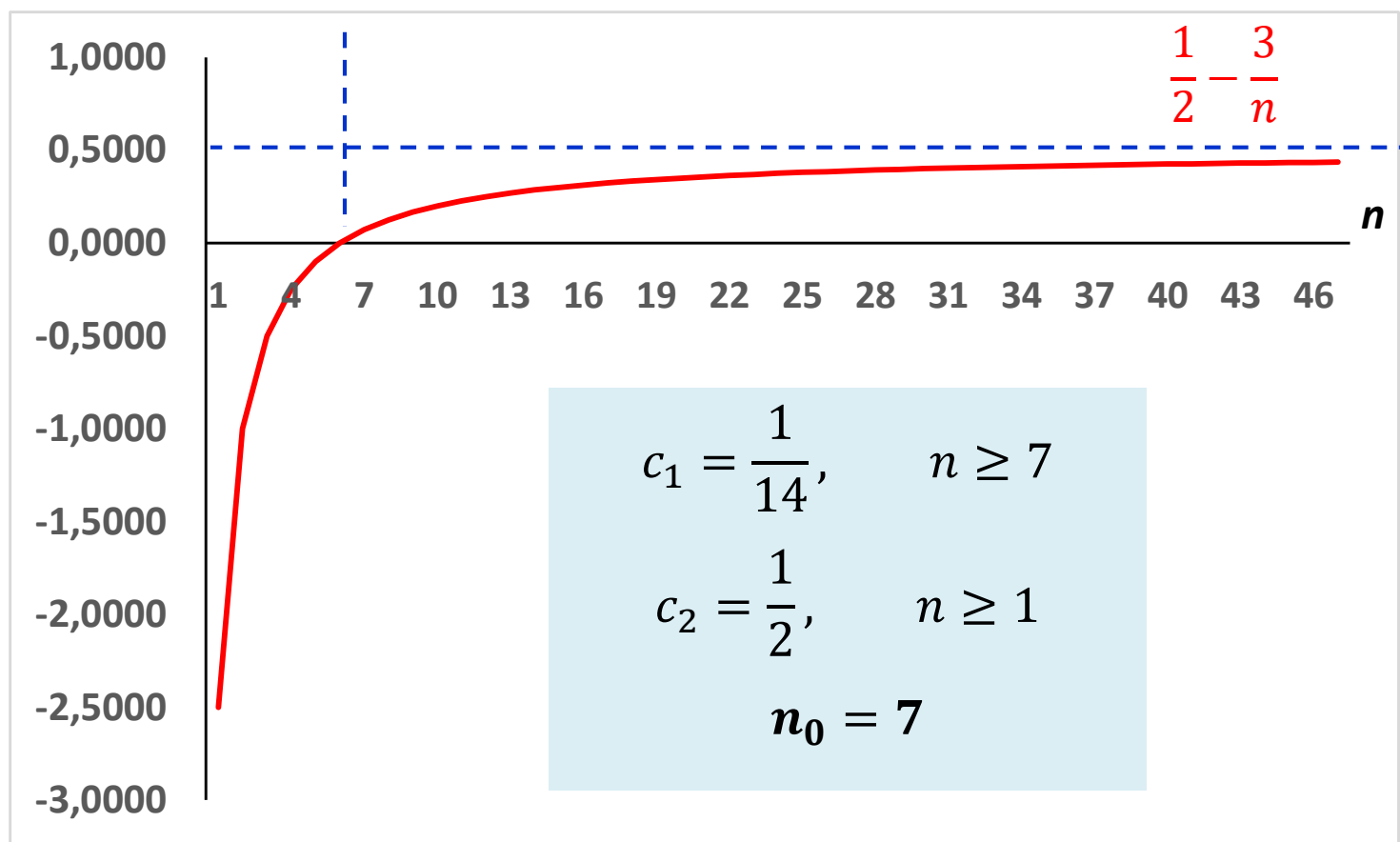
$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \quad \forall n \geq n_0$$

- **Необходимо найти c_1, c_2, n_0**

Пример

- Необходимо найти $c_1 > 0, c_2 > 0, n_0 > 0$:

$$0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2, \quad \forall n \geq n_0$$



Свойства O , Θ , Ω

1. $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

Пример: $n^3 + n^2 + n + 1 = O(n^3)$

2. $O(c \cdot f(n)) = O(f(n))$

Пример: $O(4n^3) = O(n^3)$

3. $O(f(n)) * O(g(n)) = O(f(n) * g(n))$

Пример: $O(n^3) * O(n) = O(n^4)$

Основные классы сложности

Класс сложности	Название
$O(1)$	Константная сложность
$O(\log(n))$	Логарифмическая сложность
$O(n)$	Линейная сложность
$O(n \log n)$	Линейно-логарифмическая сложность
$O(n^2)$	Квадратичная сложность
$O(n^3)$	Кубическая сложность
$O(2^n)$	Экспоненциальная сложность
$O(n!)$	Факториальная сложность

Пространственная эффективность

- Какова “сложность по памяти” алгоритма сортировки методом “пузырька”?
- Сколько ячеек памяти требуется алгоритму (не учитывая входной массив)?

```
function BubbleSort(v[1..n])  
  for i = 1 to n - 1 do  
    for j = n to i + 1 do  
      if v[j - 1] > v[j] then  
        temp = v[j - 1]  
        v[j - 1] = v[j]  
        v[j] = temp  
      end if  
    end for  
  end for  
end function
```

Пространственная эффективность

- Какова “сложность по памяти” алгоритма сортировки методом “пузырька”?
- Сколько ячеек памяти требуется алгоритму (не учитывая входной массив)?

```
function BubbleSort(v[1..n])  
  for i = 1 to n - 1 do  
    for j = n to i + 1 do  
      if v[j - 1] > v[j] then  
        temp = v[j - 1]  
        v[j - 1] = v[j]  
        v[j] = temp  
      end if  
    end for  
  end for  
end function
```

- Переменные *i, j, temp* занимают 3 ячейки памяти
- $T(n) = 3 = O(1)$
- **Константная сложность по памяти**

Домашнее чтение

- **Обязательное чтение**

- ☐ **[DSABook]** <http://dsabook.mkurnosov.net/> **[Глава 1]**
- ☐ **[CLRS]** “Глава 3. Рост функций”
- ☐ **[Aho]** “1.4 Время выполнения программ”

- **Дополнительное чтение**

- ☐ **[Levitin]** “Основы анализа эффективности алгоритмов”