Лекция 1. Анализ эффективности алгоритмов

E-mail: lazycoyote11@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2026 г.

Литература

- **[DSABook] Курносов М. Г., Берлизов Д. М. Алгоритмы и структуры обработки информации (учебное пособие). – Новосибирск: Параллель, 2019. – 211 с.**
- **[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.

- 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.
- Скиена С. С. Алгоритмы. Руководство по разработке. — 2-е изд. — СПб: БХВ, 2011 — 720 с.
- Макконнелл Дж. Основы современных алгоритмов. — 2-е изд. — М.: Техносфера, 2004. — 368 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.

Решения задачи на компьютере

1. Постановка задачи
2. Разработка алгоритма решения задачи
3. Доказательство корректности алгоритма и анализ его эффективности
4. Реализация алгоритма на языке программирования.
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)

- Запись алгоритма на формальном языке исполнителя называется программой
- Алгоритм **корректен** (*correct*), если он для любых корректных значений входных данных выдает корректные выходные данные

Свойства алгоритма

1. **Дискретность** – алгоритм представляется как последовательность инструкций исполнителя. Каждая инструкция выполняется только после того, как закончилось выполнение предыдущей команды.
2. **Конечность** (результативность, финитность) – алгоритм должен заканчиваться после выполнения конечного числа инструкций.
3. **Детерминированность** – каждый шаг алгоритма должен быть однозначно определен – записан на формальном языке исполнителя. Детерминированность обеспечивает совпадение результатов, получаемых при многократном выполнении алгоритма, на одном и том же наборе входных данных.
4. **Массовость** – алгоритм решения задачи должен быть применим для некоторого класса задач, различающихся лишь значениями входных данных.

Поиск максимального элемента

- Вход: последовательность из n чисел $(a_0, a_1, a_2, \ldots, a_{n-1})$
- Выход: индекс i элемента a_i, имеющего наибольшее значение

```
int max(int *a, int n)
{
    int maxi = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[maxi])
            maxi = i;
    }
    return maxi;
}
```

$a[0:8]$, $n = 9$

i	0	1	2	3	4	5	6	7	8
a_i	8	45	7	52	12	76	22	4	15

Поиск максимального элемента

$maxi = 0$

$i = 1$

if 45 > 8

$maxi = 1$

```
int max(int *a, int n)
{
    int maxi = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[maxi])
            maxi = i;
    }
    return maxi;
}
```

i	0	1	2	3	4	5	6	7	8
a_i	8	45	7	52	12	76	22	4	15

maxi *i*

Поиск максимального элемента

$maxi = 1$

$i = 2$

if 7 > 45

$maxi = 1$

```
int max(int *a, int n)
{
    int maxi = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[maxi])
            maxi = i;
    }
    return maxi;
}
```

i	0	1	2	3	4	5	6	7	8
a_i	8	45	7	52	12	76	22	4	15

maxi *i*

Поиск максимального элемента

maxi = 1

i = 3

if 52 > 45

maxi = 3

```
int max(int *a, int n)
{
    int maxi = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[maxi])
            maxi = i;
    }
    return maxi;
}
```

i	0	1	2	3	4	5	6	7	8
a_i	8	45	7	52	12	76	22	4	15

maxi *i*

Поиск максимального элемента

return 5

```
int max(int *a, int n)
{
    int maxi = 0;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[maxi])
            maxi = i;
    }
    return maxi;
}
```

i	0	1	2	3	4	5	6	7	8
a_i	8	45	7	52	12	76	22	4	15

maxi *i*

Показатели эффективности алгоритма

- **Количество операций** – временная эффективность (*time efficiency*), показывает насколько быстро работает алгоритм
- **Объем потребляемой памяти** – пространственная эффективность (*space efficiency*), отражает максимальное количество памяти, требуемой для выполнения алгоритма

Показатели эффективности позволяют:

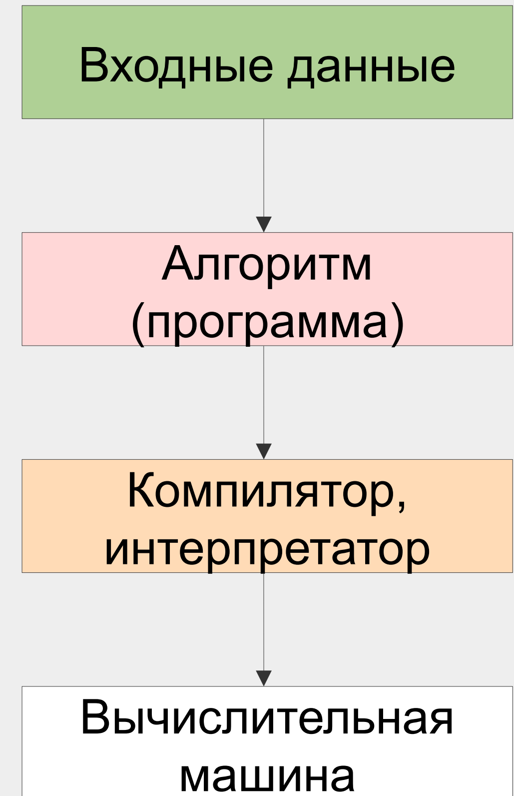
- Оценить потребности алгоритма в вычислительных ресурсах: процессорном времени, памяти, пропускной способности сети
- Сравнивать алгоритмы между собой

Существуют также другие показатели, которые имеет смысл рассматривать, если они значительно влияют на процесс выполнения задачи

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

1. У каждого алгоритма есть параметры, определяющие размер его входных данных
2. Поиск наименьшего элемента в массиве: n — количество элементов в массиве
3. Алгоритм умножения двух матриц: количества строк m и столбцов n в матрицах
4. Сравнение двух строк: s_1, s_2 — длины первой и второй строк
5. 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

От каких параметров зависит время работы алгоритма ***sum***?

```
int sum(int *a, int n)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        s = s + a[i];
    }
    return s;
}
```

Суммирование элементов массива

От каких параметров зависит время работы алгоритма ***sum***?

– От количества элементов *n* в массиве

```
int sum(int *a, int n)
{
    int s = 0;
    for (int i = 0; i < n; i++) {
        s = s + a[i];
    }
    return s;
}
```

Суммирование элементов массива

Выразим число T(n) операций, выполняемых алгоритмом

```
int sum(int *a, int n)
{
    int s = 0;          // Запись в память
    /* Проверка условия окончания, переход, i++ */
    for (int i = 0; i < n; i++) {
        s = s + a[i]; // 2 чтения, +, запись
    }
    return s;           // Возврат значения
}
```

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

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

Линейный поиск элемента в массиве

```c
int linear_search(int *a, int n, int x)
{
    for (int i = 0; i < n; i++) {
        if (a[i] == x)
            return i;
    }
    return -1;
}
```

- Количество операций алгоритма *linear_search* 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

```
int linear_search(int *a, int n, int x)
{
    for (int i = 0; i < n; i++) {
        if (a[i] == x)
            return i;
    }
    return -1;
}
```

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

LinearSearch(*a*[], **22**):

i	0	1	2	3	4
a_i	22	33	13	21	15

$T_{Best}(n) = 3$

или

$T_{Best}(n) = 1$

Линейный поиск элемента в массиве

```
int linear_search(int *a, int n, int x)
{
    for (int i = 0; i < n; i++) {
        if (a[i] == x)
            return i;
    }
    return -1;
}
```

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

LinearSearch(a[], **15***):*

i	0	1	2	3	4
a_i	22	33	13	21	15

$$T_{Worst}(n) = 2n + 1$$

или

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

Линейный поиск элемента в массиве

```
int linear_search(int *a, int n, int x)
{
    for (int i = 0; i < n; i++) {
        if (a[i] == x)
            return i;
    }
    return -1;
}
```

- **Средний случай** (*average case*) — это «средний» экземпляр задачи, набор «усредненных» входных данных
- В среднем случае оцениваются математическое ожидание количества операций, выполняемых алгоритмом
- Не всегда очевидно, какие входные данные считать «усредненными» для задачи

LinearSearch(*a*[], **x**):

i	0	1	2	3	4
a_i	22	33	13	21	15

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:

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

$$= \frac{p}{n}(1 + 2 + \ldots + i \ldots + n)$$

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 **[DSABook, 1.3]**
- Если искомый элемент присутствует в массиве (*p* = 1), то в среднем требуется выполнить (*n* + 1) / 2 операции сравнения для его нахождения

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

$$=\frac{p}{n}\left(\frac{n^2+n}{2}\right)+(1-p)n=$$

$$=p\left(\frac{n+1}{2}\right)+(1-p)n$$

Какой случай рассматривать?

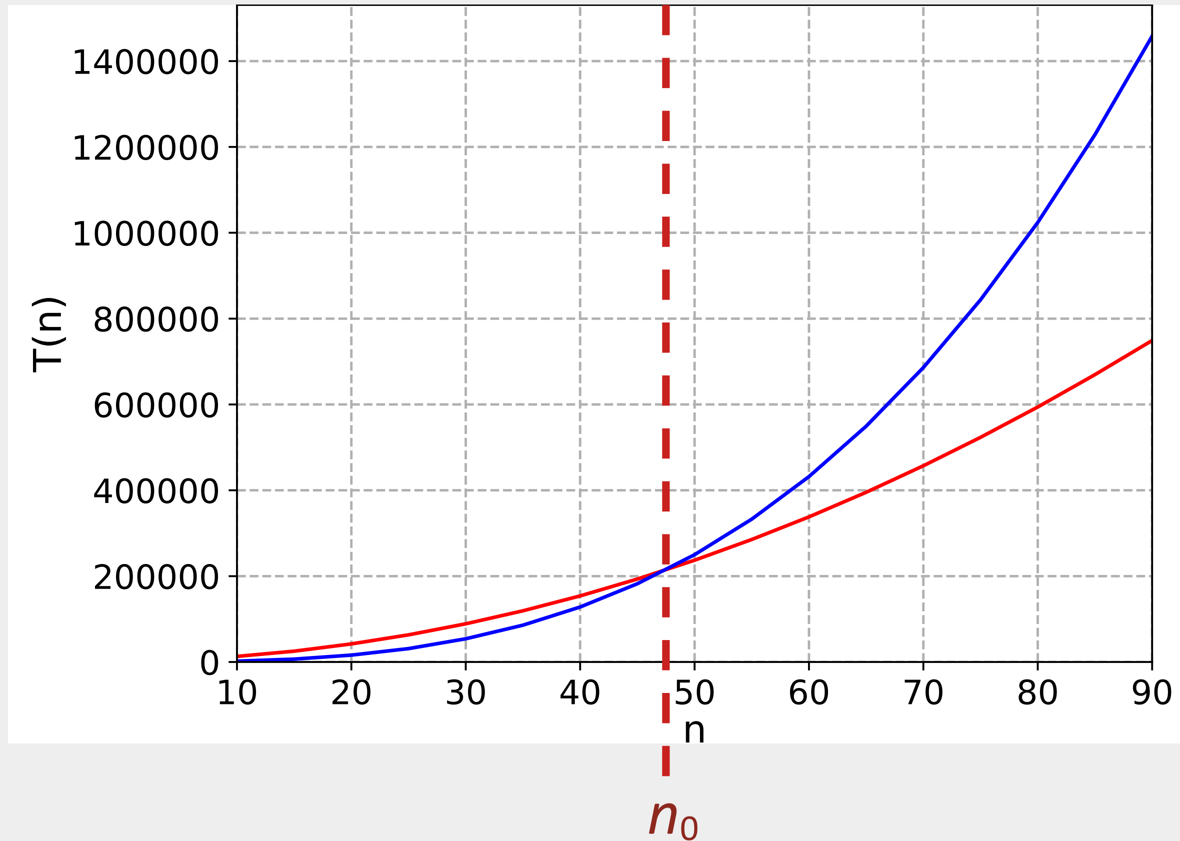
- При анализе алгоритмов мы будем уделять основное внимание времени работы алгоритмов в худшем случае — максимальному времени работы на всех наборах входных данных
- При возможности, будем строить оценки эффективности для среднего случая
- Понятия количество операций алгоритма и время выполнения алгоритма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

Анализ сложности алгоритмов

- Известно количество операций, выполняемых двумя алгоритмами, которые решают 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 функции $T_1(n)$ и $T_2(n)$, и на основе n_0 отдавать предпочтение тому или иному алгоритму

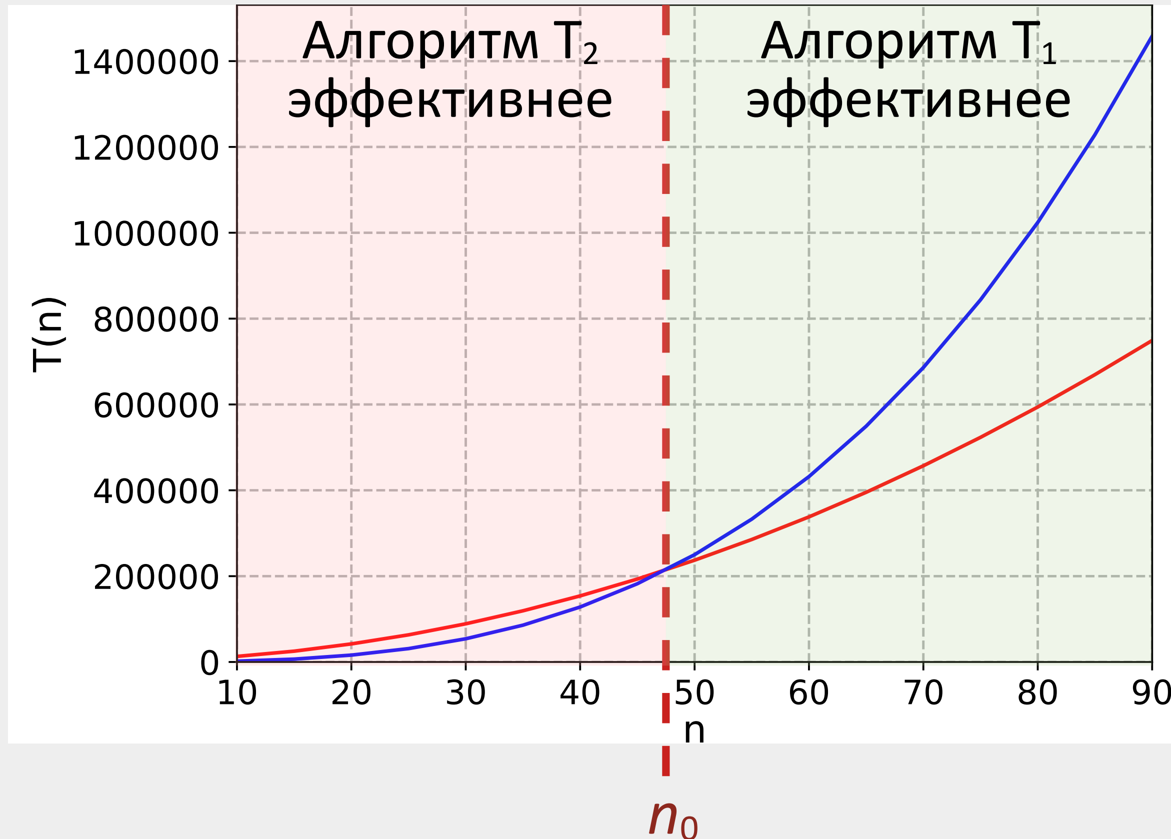


$T_2(n) = 2n^3 + 3$

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

Анализ сложности алгоритмов

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

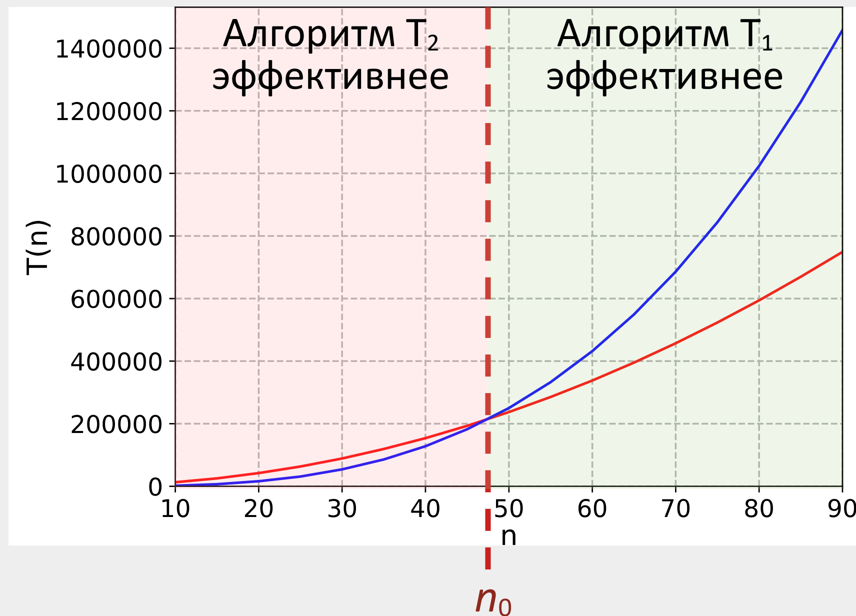


$T_2(n) = 2n^3 + 3$

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

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 данных — при $n \to \infty$



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

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

Скорость роста функций

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

$\log_2 n$	n	$n\log_2 n$	n^2	2^n	$n!$
0	1	0	1	2	1
1	2	2	4	4	2
1.6	3	5	9	8	6
2.0	4	8	16	16	24
2.3	5	12	25	32	120
2.6	6	16	36	64	720
2.8	7	20	49	128	5040
3.0	8	24	64	256	40320
3.2	9	29	81	512	362880
3.3	10	33	100	1024	3628800
10	1000	9966	$1*10^6$		
19.9	$1*10^6$	$\sim 20*10^6$	$1*10^{12}$		

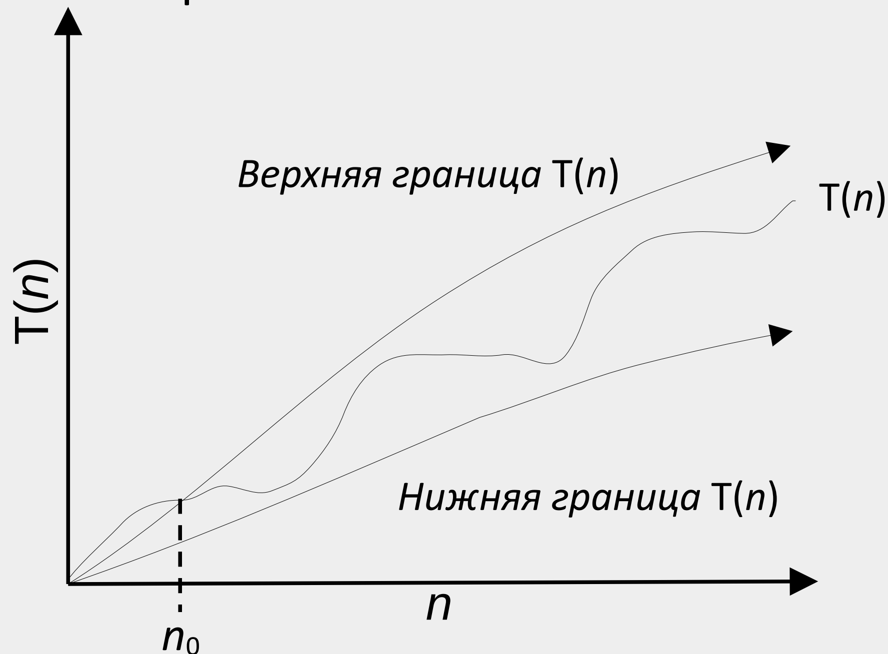
Пусть процессор выполняет одну операцию за 0.000000001 сек. (тактовая частота 1 GHz), тогда для перевода в секунды поделить на 10^9

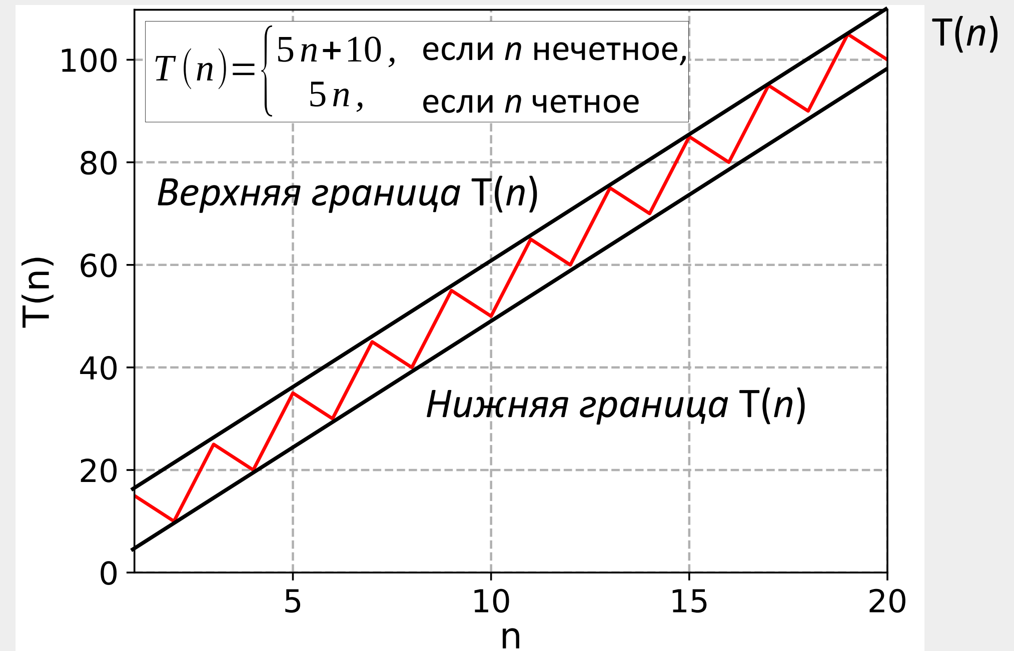
Анализ сложности алгоритмов

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

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

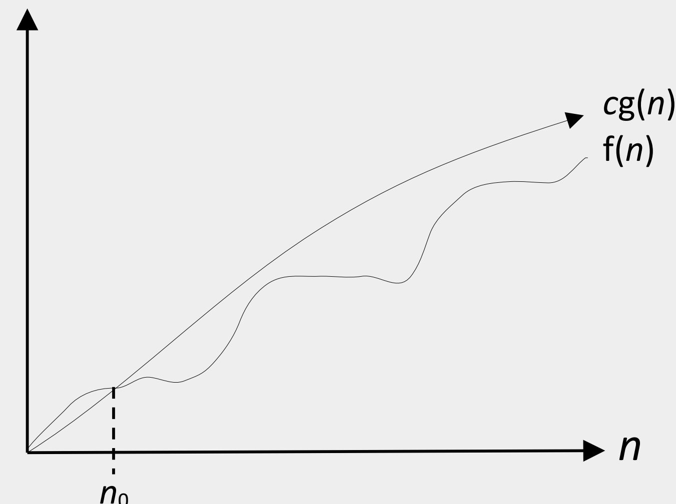
- В **теории вычислительной сложности алгоритмов** (*computational complexity theory*) для указания границ функции T(*n*) используют **асимптотические обозначения**: O (о большое), Ω (омега большое), Θ (тета большое), а также o (о малое), ω (омега малое)
- Далее будем считать, что областью определения функции f(*n*) и g(*n*), которые выражают число операций алгоритма, является множество неотрицательных целых чисел: $n \in \{0, 1, 2, \ldots\}$
- 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)):

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

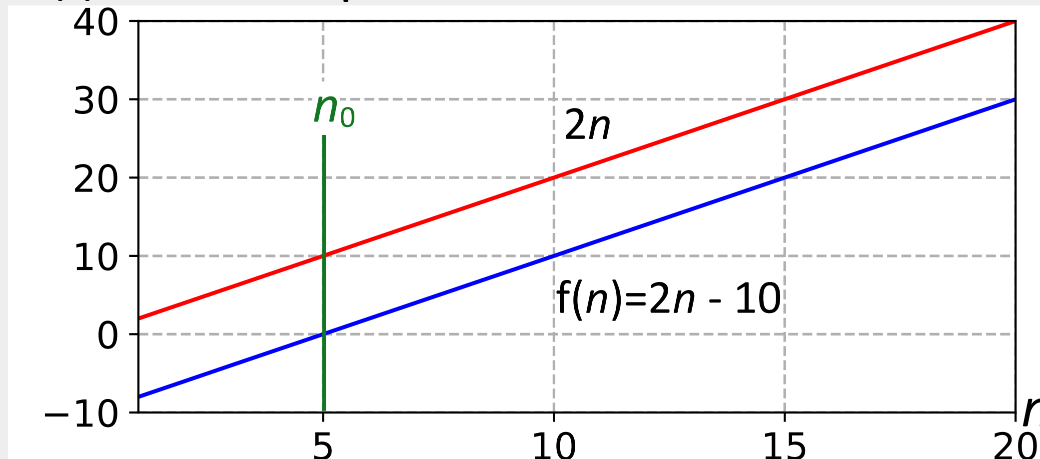
- Существуют константы $c > 0$ и n_0 {0, 1, 2, …} такие, что f(n) < $c \cdot$ g(n) для всех $n \geq n_0$
- Функция f(n) ограничена сверху функцией g(n) с точностью до постоянного множителя
- Читается как «f от n есть о большое от g от n»
- Используется, чтобы показать, что время работы не быстрее, чем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



Асимптотически 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 $n_0 \in \{0, 1, 2, \ldots\}$
- Доказательство: возьмём $c = 2$ и $n_0 = 5$
- Эти значения обеспечивают выполнение неравенства $0 \le 2n - 10 \le 2n$ для любых $n \ge 5$
- Прямая $2n$ проходит выше прямой $2n - 10$



O-обозначение (о большое)

- $3n^2 + 100n + 8 = O(n^2)$
- Для доказательства возьмем $c = 4$, $n_0 = 101$, при любых $n \geq 101$ справедливо неравенство:
- $0 \leq 3n^2 + 100n + 8 \leq 4n^2$

- $3n^2 + 100n + 8 = O(n^3)$
- Возьмем $c = 1$, $n_0 = 12$, для любых $n \geq 12$ справедливо $0 \leq 3n^2 + 100n + 8 \leq n^3$

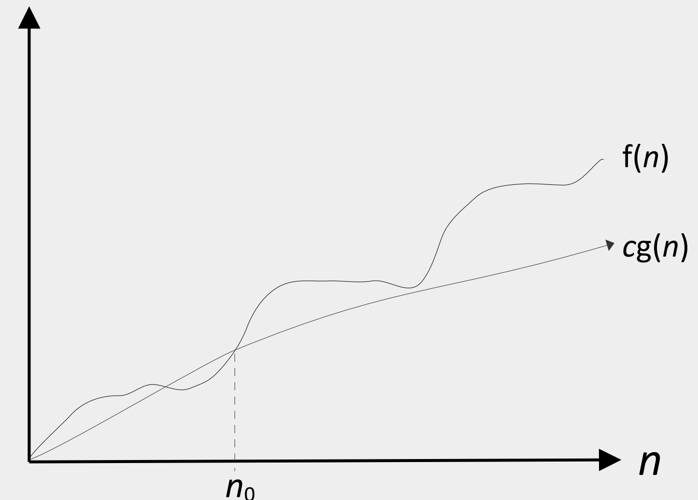
- $0.000001n^3 \neq O(n^2)$
- Не существует констант $c > 0$ и n_0, которые обеспечивают выполнение неравенств из определения O. Так для любых $c > 0$ и $n \geq c / 0.000001$ имеет место неравенство $0.000001n^3 > cn^2$

Ω-обозначение (омега большое)

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

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

- 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*»



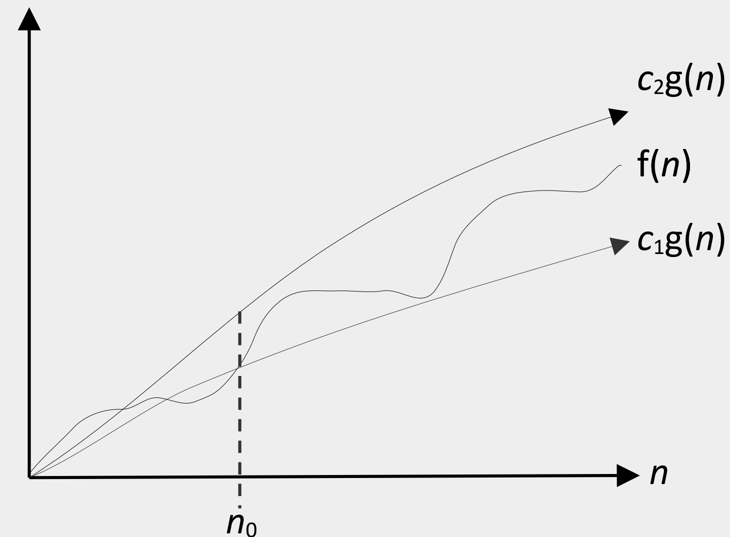
Асимптотическая нижняя граница (asymptotic lower bound) для функции f(n)

Θ-обозначение (тета большое)

- Пусть f(n) — это количество операций, выполняемых алгоритмом
- Θ-обозначение f(n) = Θ(g(n)):

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

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



Асимптотически точная оценка (asymptotic tight bound) для функции f(n)

Пример

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

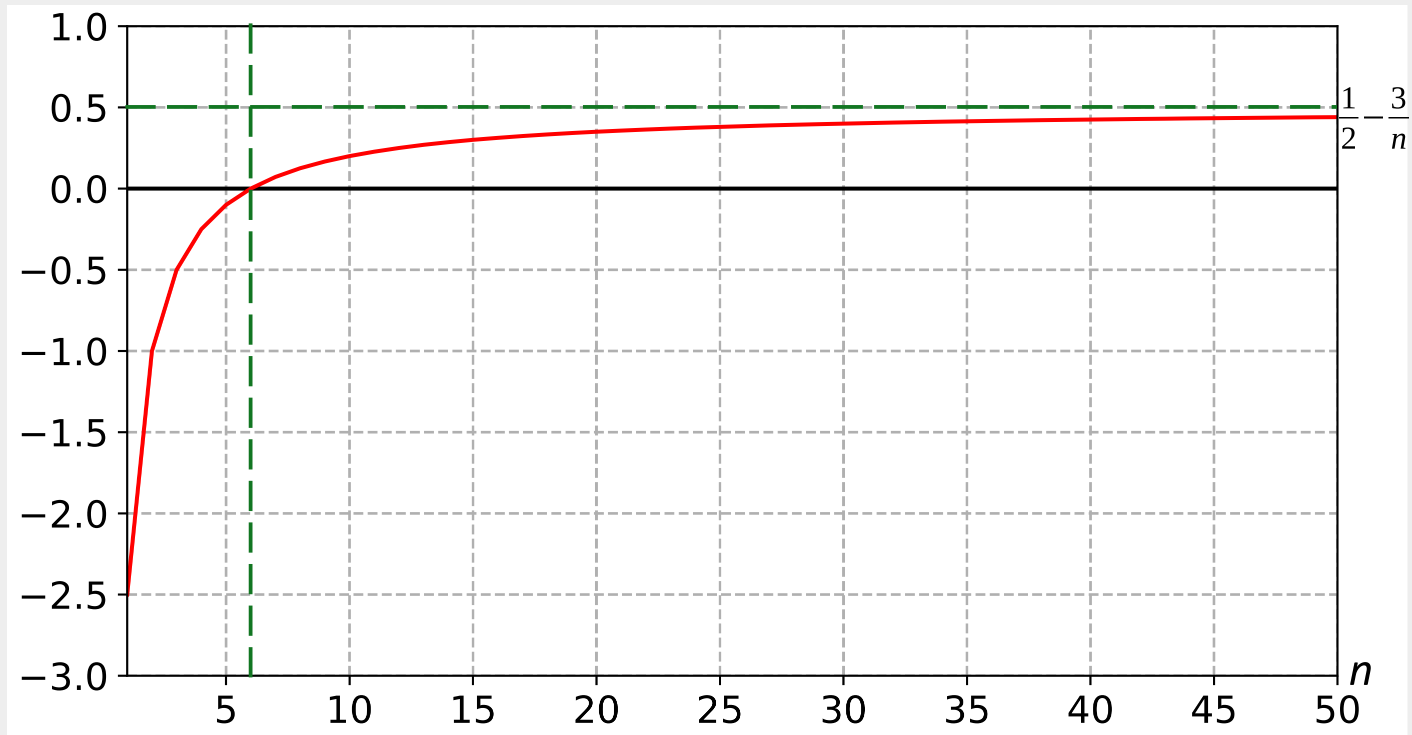
$$\exists c_1>0, c_2>0, n_0\geq 0:$$
$$0\leq c_1 n^2\leq \frac{1}{2}n^2-3n\leq c_2 n^2, \forall n\geq n_0$$

- Необходимо найти c_1, c_2, n_0

Пример

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

$$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2, \forall n \ge n_0$$



Свойства O, Ω, Θ

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

   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. $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

   Пример: $O(n^3) \cdot O(n) = O(n^4)$

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)?

```
int bubble_sort(int* arr, int n)
{
    int swapped = 1;
    while (swapped) {
        swapped = 0;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                int tmp = arr[i];
                arr[i] = arr[i – 1];
                arr[i – 1] = tmp;
                swapped = 1;
            }
        }
    }
}
```

Пространственная эффективность

- Какова «сложность по памяти» алгоритма сортировки методом «пузырька»?
- Сколько ячеек памяти требуется алгоритму (не учитывая входной массив)?
  - Переменные *i, swapped, tmp* занимают 3 ячейки памяти
  - $T(n) = 3 = O(1)$
  - Константная сложность по памяти

```
int bubble_sort(int* arr, int n)
{
    int swapped = 1;
    while (swapped) {
        swapped = 0;
        for (int i = 1; i < n; i++) {
            if (arr[i - 1] > arr[i]) {
                int tmp = arr[i];
                arr[i] = arr[i – 1];
                arr[i – 1] = tmp;
                swapped = 1;
            }
        }
    }
}
```

Дальнейшее чтение

- **[DSABook]** http://dsabook.mkurnosov.net [Глава 1]
- **[CLRS]** Глава 3. «Рост функций»
- **[Aho]** 1.4 «Время выполнения функций»