Оценка сложности алгоритмов. Квадратичные сортировки

Denis Bakin

Оценка сложности алгоритмов

Зачем анализировать сложность

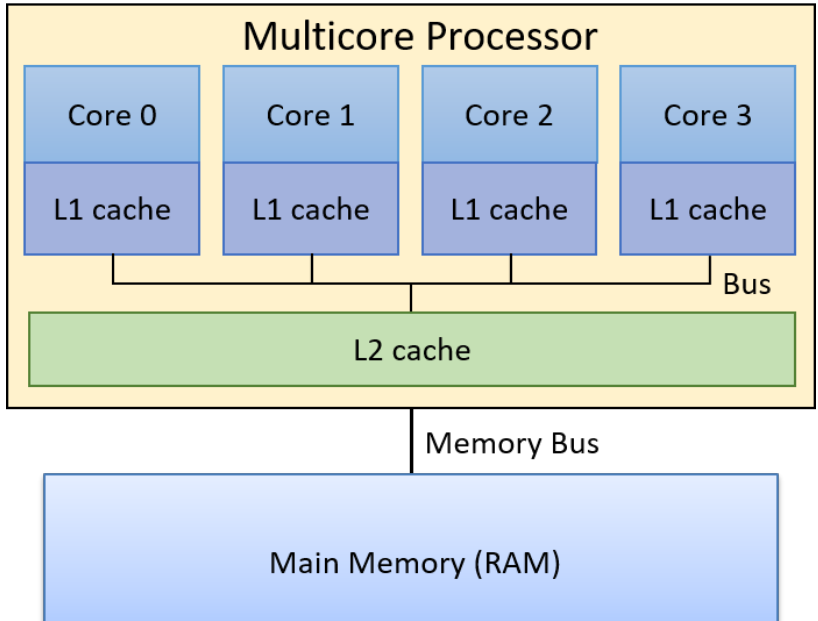
- Время в секундах зависит от машины и входных данных
- Удобнее считать количество элементарных операций (модель далее)
- Сравнение алгоритмов выполняется в терминах асимптотики (Big-O)

Упрощённая модель

Что мы считаем за одну операцию

- Арифметические операции (включая битовые)
- Обращение к памяти (считаем идеальную RAM)
- Доступ к элементу массива/вектора
- Модель упрощённая (память многослойна в реальности), но достаточна для наших рассуждений

Настоящая система памяти



Latency Numbers Every Programmer Should Know

```
Latency Comparison Numbers (~2012)
----------------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns                      14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns                      20x L2 cache, 200x
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us           ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD, 4X me
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datacenter roun
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

credit to Jeff Dean

O-нотация (Big-O)

Формальное определение

- Будем писать $f(x) = O(g(x)) \iff \exists C > 0 : \ f(x) < C \cdot g(x)$ при достаточно больших x
- O-нотация описывает скорость роста функции при $x \to +\infty$ — важен асимптотический тип роста, а не множители

Примеры O-нотации

Наглядно

- $f_1(n) = 1 + 2 + \cdots + n =$

Примеры O-нотации

Наглядно

- $f_1(n) = 1 + 2 + \cdots + n =$
- $f_1(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$

Примеры O-нотации

Наглядно

- $f_1(n) = 1 + 2 + \cdots + n =$
- $f_1(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$
- $f_2(n) = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} =$

Примеры O-нотации

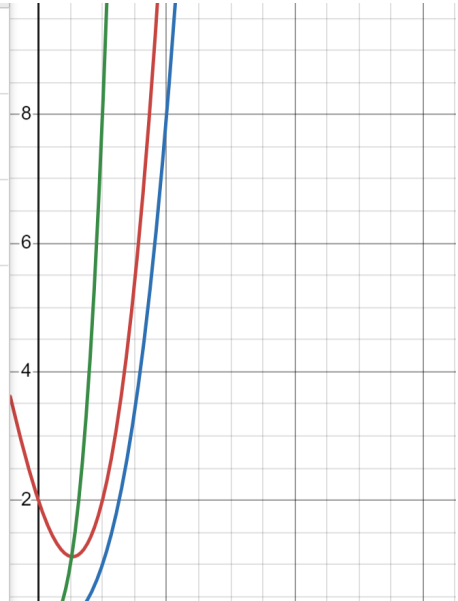
Наглядно

- $f_1(n) = 1 + 2 + \cdots + n =$
- $f_1(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$
- $f_2(n) = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} =$
- $f_2(n) = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = O(n^3)$

Примеры O-нотации

Наглядно

1. $x^3 + 2x^2 - 3x + 2$
2. x^3
3. $8x^3$
4.



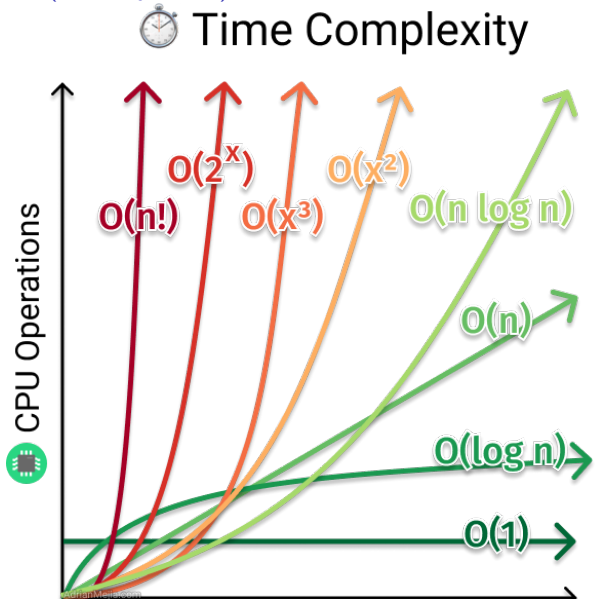
Частые классы сложности

От медленных к быстрым (рост функций)

- $O(1)$ — константа
- $O(\log n)$ — логарифм
- $O(\sqrt{n})$ — корень
- $O(n)$ — линейная
- $O(n \log n)$ — квазилинейная
- $O(n^2), O(n^3)$ — полиномиальные
- $O(2^n), O(n!)$ — экспоненциальные/факториальные

Частые классы сложности

От медленных к быстрым (рост функций)



Оценка сложности алгоритма

Дополнительные обозначения

- $\Omega(f(n))$ — нижняя оценка
- $\Theta(f(n))$ — точная оценка
- $O(f(n))$ — верхняя оценка
- Если $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$, то $f(n) = \Theta(g(n))$

Оценка сложности алгоритма

Критерий оптимальности

$$\begin{cases} T_{\text{algorithm}}(n) = O(f(n)) \\ \text{Problem}(n) = \Omega(f(n)) \end{cases}$$

То есть требуем, чтобы асимптотическая сложность алгоритма не хуже нижней оценки сложности задачи

Оценка сложности алгоритма

Способы поиска нижней оценки: алгоритм противника

Необходимо предъявить стратегию противника, которая:

Оценка сложности алгоритма

Способы поиска нижней оценки: алгоритм противника

Необходимо предъявить стратегию противника, которая:

- заставит алгоритм выполнить не менее $f(n)$ операций

Оценка сложности алгоритма

Способы поиска нижней оценки: алгоритм противника

Необходимо предъявить стратегию противника, которая:

- заставит алгоритм выполнить не менее $f(n)$ операций
- задает непротиворечивую задачу (при этом неизвестные игроку данные могут быть любыми)

Оценка сложности алгоритма

Способы поиска нижней оценки: алгоритм противника

Необходимо предъявить стратегию противника, которая:

- заставит алгоритм выполнить не менее $f(n)$ операций
- задает непротиворечивую задачу (при этом неизвестные игроку данные могут быть любыми)
- пример: поиск максимума в массиве из n чисел требует не менее $n - 1$ сравнения

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- каждую задачу можно представить в виде дерева решений
- листы дерева — возможные ответы
- высота дерева — количество шагов (операций) в худшем случае
- оптимальная сложность сортировки?
- $\Omega(n \log n)$

Оценка сложности алгоритма

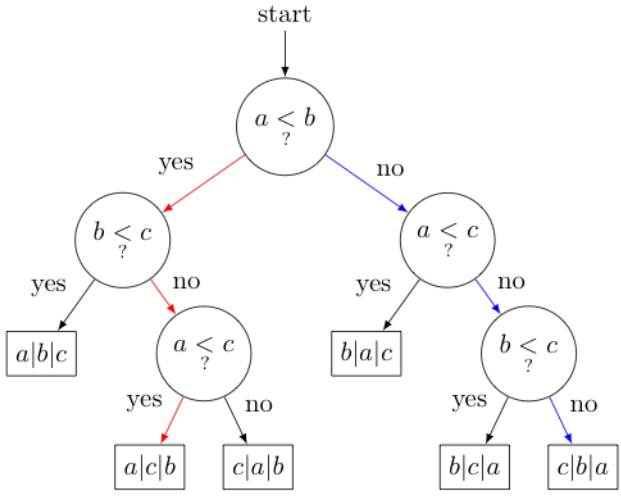


Figure 4: Пример решающего дерева

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- n элементов можно упорядочить $n!$ способами

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- n элементов можно упорядочить $n!$ способами
- в решающем дереве должно быть не менее $n!$ листьев

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- n элементов можно упорядочить $n!$ способами
- в решающем дереве должно быть не менее $n!$ листьев
- дерево с h высотой имеет не более 2^h листьев

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- n элементов можно упорядочить $n!$ способами
- в решающем дереве должно быть не менее $n!$ листьев
- дерево с h высотой имеет не более 2^h листьев
- $2^h \geq n!$

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- n элементов можно упорядочить $n!$ способами
- в решающем дереве должно быть не менее $n!$ листьев
- дерево с h высотой имеет не более 2^h листьев
- $2^h \geq n!$
- $h \geq \log_2(n!)$\$

Оценка сложности алгоритма

Способы поиска нижней оценки: оценки в решающем дереве

- n элементов можно упорядочить $n!$ способами
- в решающем дереве должно быть не менее $n!$ листьев
- дерево с h высотой имеет не более 2^h листьев
- $2^h \geq n!$
- $h \geq \log_2(n!)$\$
- $h = \Omega(n \log n)$

Пространственной сложности

Дана последовательность натуральных чисел $x_1, x_2, \ldots, x_n$. Стандартным отклонением называется величина

$$\sigma = \sqrt{\frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \ldots + (x_n - \mu)^2}{n - 1}},$$

где μ — среднее арифметическое чисел $x_1, x_2, \ldots, x_n$.

Определите стандартное отклонение для данной последовательности чисел.

- какова оценка сложности по времени?
- какова оценка сложности по памяти?

Сортировка: постановка задачи

- Даны n объектов, для которых определён только оператор «меньше» (<)
- Нужно переставить элементы так, чтобы
  - $\forall i < j \ : \ a_i < a_j$
- Формально: найти такую *перестановку* индексов p, что
  - $a[p_1] < a[p_2] < ... < a[p_n]$

Квадратичные сортировки

Общие свойства

- $O(n^2)$
- Полезны на небольших данных или как часть других алгоритмов
- Основные представители:
  - сортировка пузырьком
  - сортировка выбором
  - сортировка вставками

Сортировка пузырьком

Идея алгоритма

- Проходим по массиву и сравниваем соседние элементы

Сортировка пузырьком

Идея алгоритма

- Проходим по массиву и сравниваем соседние элементы
- $a_i > a_{i+1} \Rightarrow \mathrm{swap}(a_i, a_{i+1})$

Сортировка пузырьком

Идея алгоритма

- Проходим по массиву и сравниваем соседние элементы
- $a_i > a_{i+1} \Rightarrow \mathrm{swap}(a_i, a_{i+1})$
- После каждого прохода **наибольший** элемент «всплывает» в конец

Сортировка пузырьком

Идея алгоритма

- Проходим по массиву и сравниваем соседние элементы
- $a_i > a_{i+1} \Rightarrow \mathrm{swap}(a_i, a_{i+1})$
- После каждого прохода **наибольший** элемент «всплывает» в конец
- После `k` проходов последние `k` элементов гарантированно на своих местах

Пузырёк: пример работы

Массив: $[5, 3, 8, 4, 2]$

Шаг	Действие	Результат
1	Сравнить 5 и $3 \to$ поменять	$[3, 5, 8, 4, 2]$
2	Сравнить 5 и $8 \to$ ок	$[3, 5, 8, 4, 2]$
3	Сравнить 8 и $4 \to$ поменять	$[3, 5, 4, 8, 2]$
4	Сравнить 8 и $2 \to$ поменять	$[3, 5, 4, 2, 8]$
...	Повторять, пока массив отсортирован	$[2, 3, 4, 5, 8]$

Пузырёк: псевдокод

```
function bubble_sort(arr):
    n = длина(arr)
    for i in 0 .. n-1:
        for j in 0 .. n-i-2:
            if arr[j] > arr[j+1]:
                swap(arr[j], arr[j+1])
    return arr
```

- Сложность: $O(n^2)$ сравнений и перестановок

Сортировка выбором

Идея алгоритма

- На каждой итерации ищем **минимальный** элемент в неотсортированной части

Сортировка выбором

Идея алгоритма

- На каждой итерации ищем **минимальный** элемент в неотсортированной части
- Меняем его местами с первым элементом этой части

Сортировка выбором

Идея алгоритма

- На каждой итерации ищем **минимальный** элемент в неотсортированной части
- Меняем его местами с первым элементом этой части
- После `k` итераций первые `k` элементов уже на своих местах

Выбор: пример работы

Массив: $[5, 3, 8, 4, 2]$

Шаг	Действие	Результат
1	Найти минимум (2), поставить в начало	$[2, 3, 8, 4, 5]$
2	Найти минимум в хвосте (3), поставить на позицию 2	$[2, 3, 8, 4, 5]$
3	Найти минимум в хвосте (4), поставить на позицию 3	$[2, 3, 4, 8, 5]$
4	Найти минимум в хвосте (5), поставить на позицию 4	$[2, 3, 4, 5, 8]$

Выбор: псевдокод

```
function selection_sort(arr):
    n = длина(arr)
    for i in 0 .. n-1:
        min_idx = i
        for j in i+1 .. n-1:
            if arr[j] < arr[min_idx]:
                min_idx = j
        swap(arr[i], arr[min_idx])
    return arr
```

- Сложность: $O(n^2)$ сравнений
- Число перестановок — всего $O(n)$ (по одной на итерацию)

Сортировка вставками

Идея алгоритма

- Считаем, что первый элемент уже отсортирован

Сортировка вставками

Идея алгоритма

- Считаем, что первый элемент уже отсортирован
- Для каждого следующего элемента находим позицию в **отсортированной** части и вставляем его туда

Сортировка вставками

Идея алгоритма

- Считаем, что первый элемент уже отсортирован
- Для каждого следующего элемента находим позицию в **отсортированной** части и вставляем его туда
- Элементы, большие текущего, сдвигаем вправо

Вставки: пример работы

Массив: $[5, 3, 8, 4, 2]$

Шаг	Действие	Результат
1	$[5]$ уже отсортирован	$[5]$
2	Вставить 3	$[3, 5]$
3	Вставить 8	$[3, 5, 8]$
4	Вставить 4	$[3, 4, 5, 8]$
5	Вставить 2	$[2, 3, 4, 5, 8]$

Вставки: псевдокод

```
function insertion_sort(arr):
    n = длина(arr)
    for i in 1 .. n-1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key
    return arr
```

- Сложность: $O(n^2)$ в худшем случае
- Но если массив почти отсортирован — может работать почти за $O(n)$

Сравнение алгоритмов

Алгоритм	Сравнения
Пузырёк	$O(n^2)$
Выбор	$O(n^2)$
Вставки	$O(n^2)$

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$
- Бинарный метод: разбиваем степень по двоичному представлению $\to O(\log d)$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$
- Бинарный метод: разбиваем степень по двоичному представлению $\to O(\log d)$
- $3^{10} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$
- Бинарный метод: разбиваем степень по двоичному представлению $\to O(\log d)$
- $3^{10} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$
- $3^{10} = (3^2)^5 = 9^5$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$
- Бинарный метод: разбиваем степень по двоичному представлению $\to O(\log d)$
- $3^{10} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$
- $3^{10} = (3^2)^5 = 9^5$
- $9^5 = (9^2)^2 \cdot 9 = 81^2 \cdot 9$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$
- Бинарный метод: разбиваем степень по двоичному представлению $\to O(\log d)$
- $3^{10} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$
- $3^{10} = (3^2)^5 = 9^5$
- $9^5 = (9^2)^2 \cdot 9 = 81^2 \cdot 9$
- $81^2 = 6561$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- Наивное: a^d — d умножений $\to O(d)$
- Бинарный метод: разбиваем степень по двоичному представлению $\to O(\log d)$
- $3^{10} = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$
- $3^{10} = (3^2)^5 = 9^5$
- $9^5 = (9^2)^2 \cdot 9 = 81^2 \cdot 9$
- $81^2 = 6561$
- Итого: $3^{10} = 6561 \cdot 9 = 59049$

```
result := 1
while deg =/ 0:
    if deg is even:
        deg := deg / 2
        num := num * num
    else:
        deg := deg - 1
        result := result * num
```

Быстрое возведение в степень

Идея бинарного возведения (exponentiation by squaring)

- $O(\log d)$

```
result := 1
while deg =/ 0:
    if deg is odd:
        deg := deg - 1
        result := result * num

    deg := deg / 2
    num := num * num
```

Простые числа и делители — определения

Формулировки

- Делитель: a — делитель n, если $a \in \mathbb{N}$, $n \ \vdots \ a$ (т.е. $n \bmod a = 0$)
- Простое число: натуральное $p > 1$, у которого ровно два делителя: 1 и p

Лемма о парных делителях

- Пусть $n \vdots a$ и $b = n/a$ — парный делитель, $ab = n$

Лемма о парных делителях

- Пусть $n \vdots a$ и $b = n/a$ — парный делитель, $ab = n$
- Тогда не могут оба быть строго меньше $\sqrt{n}$, иначе $ab < n$ — противоречие

Лемма о парных делителях

- Пусть $n \vdots a$ и $b = n/a$ — парный делитель, $ab = n$
- Тогда не могут оба быть строго меньше $\sqrt{n}$, иначе $ab < n$ — противоречие
- И не могут оба быть больше $\sqrt{n}$, иначе $ab > n$ — тоже противоречие

Лемма о парных делителях

- Пусть $n \vdots a$ и $b = n/a$ — парный делитель, $ab = n$
- Тогда не могут оба быть строго меньше $\sqrt{n}$, иначе $ab < n$ — противоречие
- И не могут оба быть больше $\sqrt{n}$, иначе $ab > n$ — тоже противоречие
- Следовательно: для каждой пары один делитель $\leq \sqrt{n}$, другой $\geq \sqrt{n}$

Лемма о парных делителях

- Пусть $n \vdots a$ и $b = n/a$ — парный делитель, $ab = n$
- Тогда не могут оба быть строго меньше $\sqrt{n}$, иначе $ab < n$ — противоречие
- И не могут оба быть больше $\sqrt{n}$, иначе $ab > n$ — тоже противоречие
- Следовательно: для каждой пары один делитель $\leq \sqrt{n}$, другой $\geq \sqrt{n}$
- Значит, перебрав a от 1 до $\lfloor \sqrt{n} \rfloor$, мы найдём по одному представителю из каждой пары

Стратегии поиска делителей

Стратегии поиска делителей

- Перебрать все a от 1 до n (наивно) — очевидно $O(n)$

Стратегии поиска делителей

- Перебрать все a от 1 до n (наивно) — очевидно $O(n)$
- Перебрать a от 1 до $n/2$ (оптимизация: числа больше $n/2$ не делят n, кроме n самого) — $O(n)$, но с константой $\frac{1}{2}$

Стратегии поиска делителей

- Перебрать все a от 1 до n (наивно) — очевидно $O(n)$
- Перебрать a от 1 до $n/2$ (оптимизация: числа больше $n/2$ не делят n, кроме n самого) — $O(n)$, но с константой $\frac{1}{2}$
- Применить лемму: перебирать только до $\sqrt{n}$, добавляя парный n/a — $O(\sqrt{n})$

Пример (пошагово) для $n = 36$

Как ведёт себя каждый метод

- Перебор до n:
  - проверяются числа 1..36 $\rightarrow$ найдены делители: 1,2,3,4,6,9,12,18,36
- Перебор до $n/2$:
  - проверяются 1..18 $\rightarrow$ те же делители, кроме 36
- Перебор до $\sqrt{n}$ (здесь $\sqrt{36} = 6$):
  - $a = 1 \rightarrow$ пары: (1,36)
  - $a = 2 \rightarrow$ пары: (2,18)
  - $a = 3 \rightarrow$ пары: (3,12)
  - $a = 4 \rightarrow$ пары: (4,9)
  - $a = 5 \rightarrow$ не делит
  - $a = 6 \rightarrow$ парный: (6,6) — корень, учитывать один раз

Идея: перебор до $\sqrt{n}$ и добавление парного делителя

Пошаговое описание

- Для каждого a от 1 до $\lfloor\sqrt{n}\rfloor$:
  - Если $n \vdots a$, то $b = n/a$ — парный делитель
  - Если $a \neq b$, нужно учесть оба a и b
  - Если $a = b$ (точный квадрат) — учесть его один раз
- Количество итераций — $\lfloor\sqrt{n}\rfloor \to O(\sqrt{n})$