Лекция 1. Асимптотика. Введение.

Алгоритм

Алгоритм - это некоторая последовательность действий, которые необходимо совершить для достижения нужного результата.

Яичница с зеленым луком и помидорами черри

Chef-daw

Свойства алгоритма

- **Конечность** (результативность) алгоритма означает, что за конечное число шагов должен быть получен результат;
- **Дискретность** алгоритма означает, что алгоритм должен быть разбит на последовательность выполняемых шагов;
- **Понятность** алгоритма означает, что алгоритм должен содержать только те команды, которые входят в набор команд, который может выполнить конкретный исполнитель;
- **Точность** алгоритма означает, что каждая команда должна пониматься однозначно;
- **Массовость** алгоритма означает, что однажды составленный алгоритм должен подходить для решения подобных задач с разными исходными данными.
- **Детерминированность** (определенность). Алгоритм обладает свойством детерминированности, если для одних и тех же наборов исходных данных он будет выдавать один и тот же результат, т.е. результат однозначно определяется исходными данными.

Источник: https://learnpascal.ru/vvedenie-v-paskal/algoritm.html


2009...
NEWS!
WOW!
50 miles away...

Сложность алгоритма

- Временная сложность (в худшем случае) — это функция от размера входных данных, равная максимальному количеству элементарных операций, проделываемых алгоритмом для решения экземпляра задачи указанного размера.

- Пространственная (емкостная) сложность относится к объему используемой памяти, необходимой для выполнения алгоритма.

В чем измерять временную сложность?

- В секундах?
- В котиках?

В чем измерять временную сложность?

- В секундах?
- В котиках?
- В элементарных операциях!

Модель RAM (Random Access Machine)

Каждое вычислительное устройство имеет свои особенности, которые могут влиять на длительность вычисления. Обычно при разработке алгоритма не берутся во внимание такие детали, как размер кэша процессора или тип многозадачности, реализуемый операционной системой. Анализ алгоритмов проводят на модели абстрактного вычислителя, называемого *машиной с произвольным доступом к памяти (RAM).*

Модель состоит из памяти и процессора, которые работают следующим образом: несмотря на то, что такая модель далека от реального компьютера, она замечательно подходит для анализа алгоритмов. После того, как алгоритм будет реализован для конкретной ЭВМ, вы можете заняться профилированием и низкоуровневой оптимизацией, но это будет уже оптимизация кода, а не алгоритма.

Модель RAM (Random Access Machine)

- память состоит из ячеек, каждая из которых имеет адрес и может хранить один элемент данных;
- каждое обращение к памяти занимает одну единицу времени, независимо от номера адресуемой ячейки;
- количество памяти достаточно для выполнения любого алгоритма;
- процессор выполняет любую элементарную операцию (основные логические и арифметические операции, чтение из памяти, запись в память, вызов подпрограммы и т.п.) за один временной шаг;
- циклы и функции не считаются элементарными операциями.

Пример – посчитать число элементарных операций

```
import sys
min_value = sys.maxsize #максимально
возможное значение типа int
max_value = -sys.maxsize - 1 #минимально
возможное значение типа int

for x in array:
    if x < min_value:
        min_value = x
    if x > max_value:
        max_value = x
```

```
import sys
min_value = sys.maxsize #максимально
возможное значение типа int
max_value = -sys.maxsize - 1 #минимально
возможное значение типа int

for x in array:
    if x < min_value:
        min_value = x
for x in array:
    if x > max_value:
        max_value = x
```

Упрощения

- Оцениваем самую быстрорастущую часть («худшее») слагаемого;
- Игнорируем коэффициенты

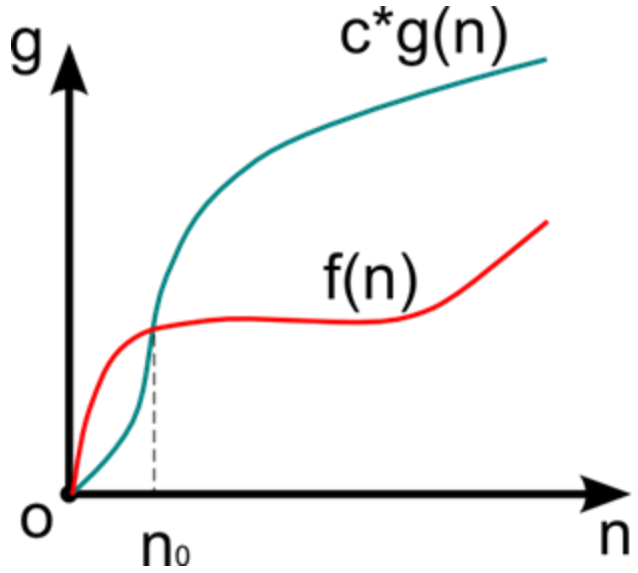
Получаем асимптотическую сложность!

Big-O

Оценка худшего случая

$$f(n) \in O(g(n))$$

$$\exists(C > 0), n_0 : \forall(n > n_0)\ |f(n)| \leq |Cg(n)| \quad \text{или}$$
$$\exists(C > 0), n_0 : \forall(n > n_0)\ f(n) \leq Cg(n)$$



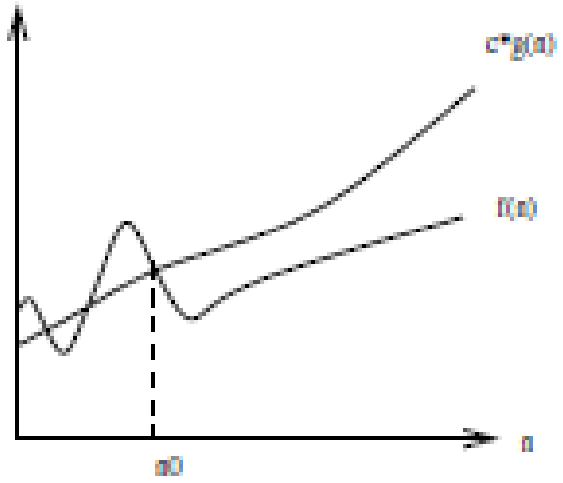
Оценка O (O большое)

Оценка O требует, чтобы функция $f(n)$ не превышала значения функции $g(n)$ при $n>n_0$ с точностью до положительного постоянного множителя, а именно: $f(n) = O(g(n))$, если

$$\exists c > 0, n_0 > 0 : \forall n > n_0 \quad 0 \le f(n) \le cg(n)$$

$O(g(n))$ обозначает класс функций, таких, что все они растут не быстрее, чем функция $g(n)$ с точностью до положительного постоянного множителя. Говорят, что функция $g(n)$ мажорирует функцию $f(n)$

Paul Bachmann, 1894 > Edmund Landau, 1909 > Ordnung



Оценка O (O большое)

Для функций

$$f_1(n) = 12, \quad f_2(n) = 5n + 23, \quad f_3(n) = n \ln n, \quad f_4(n) = 7n^2 + 12n - 34$$

будет справедливой оценка O(n^2).
Следует указывать наиболее «близкую» мажорирующую функцию. Например, для функции $f(n) = 12\ n^3$ справедлива оценка O(2^n), но практически она малопригодна

Оценка Ω (Омега)

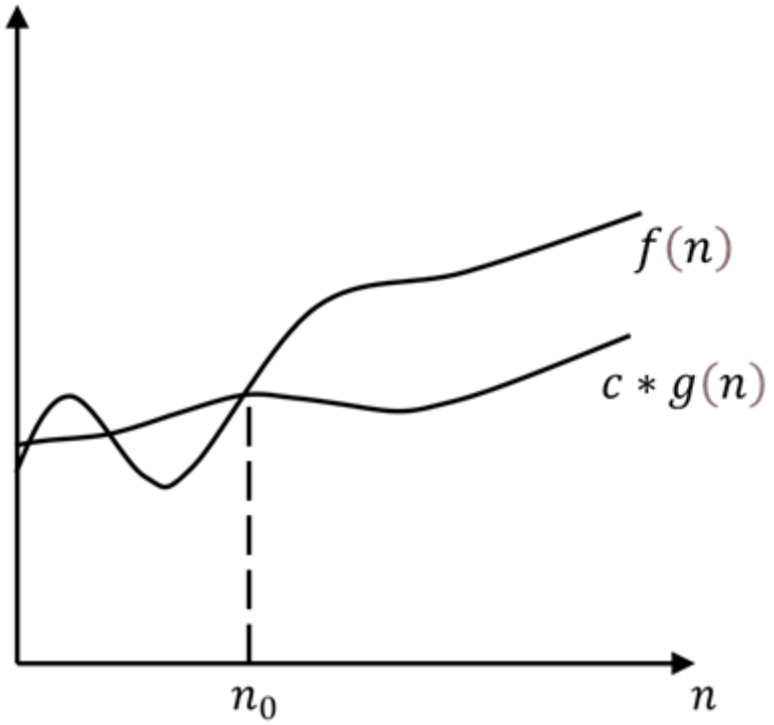
Оценка Ω является оценкой снизу – определяет класс функций, которые растут не медленнее, чем функция $g(n)$ с точностью до положительного постоянного множителя.

$f(n) = \Omega(g(n))$, если

$$\exists C > 0, n_0 > 0 : \forall n > n_0 \qquad 0 \leq Cg(n) \leq f(n)$$

Например, $\Omega(n\ ln(n))$ обозначает класс функций, которые растут не медленнее, чем функция $g(n) = n \ln(n)$.

В этот класс попадают, например, все полиномы со степенью больше единицы



Оценка Θ (Тета)

Функция $f(n) = \Theta(g(n))$, если

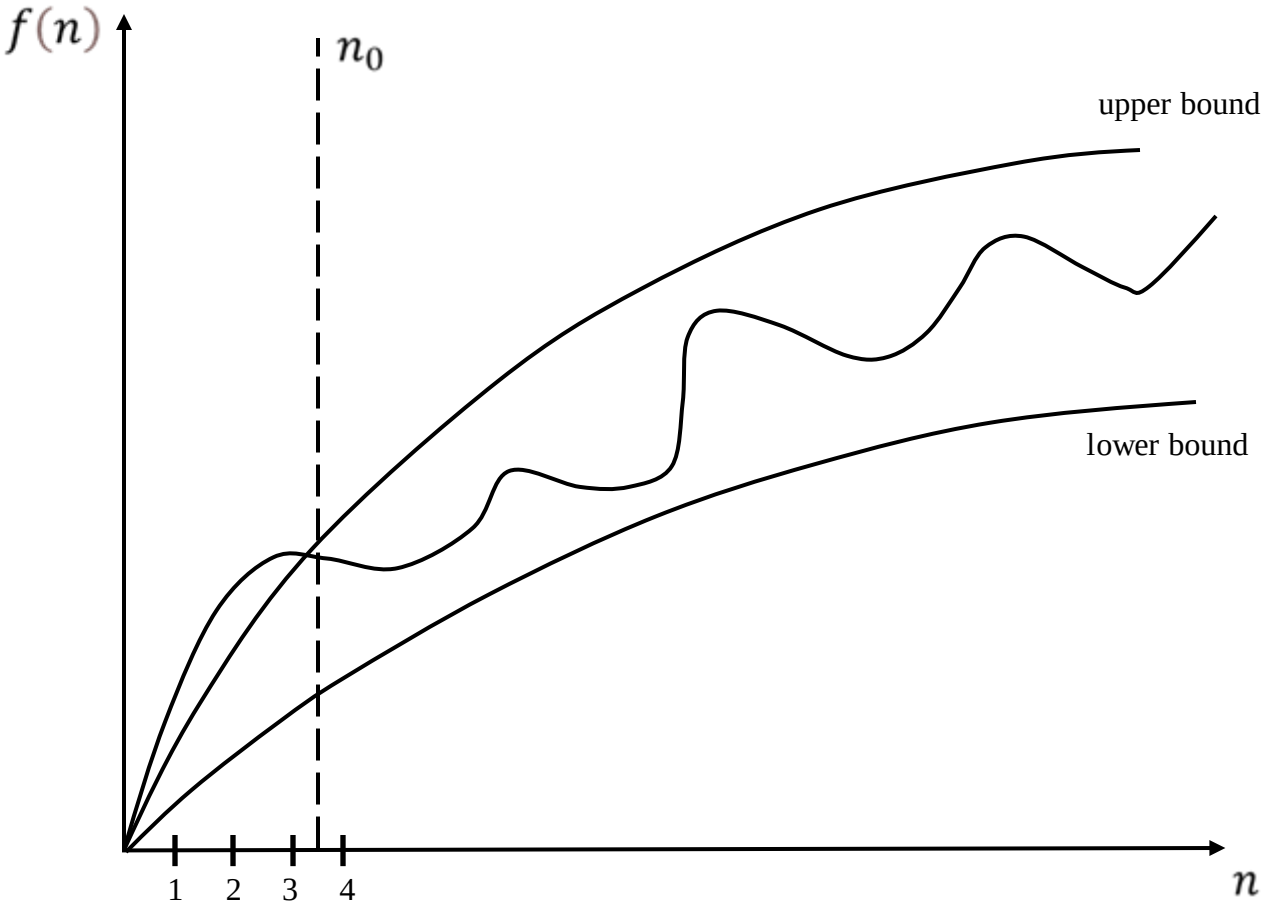
$$\exists c_1 > 0, c_2 > 0, n_0 > 0 : \forall n > n_0 \quad c_1 g(n) \le f(n) \le c_2 g(n)$$

Функция $g(n)$ является асимптотически точной оценкой функции $f(n)$, т.к. функция $f(n)$ отличается от функции $g(n)$ на положительный ограниченный множитель при всех значениях аргумента $n > n_0$.

Запись $f(n) = \Theta(1)$ означает, что функция $f(n)$ или равна константе, не равной нулю, или ограничена двумя положительными константами при любых значениях аргумента $n > n_0$.

Обозначение $\Theta(g(n))$ есть обозначение класса функций, каждая из которых удовлетворяет условию.

Оценка Θ (Тета)



Существуют положительные константы c_1, c_2 и n_o такие что

$o \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

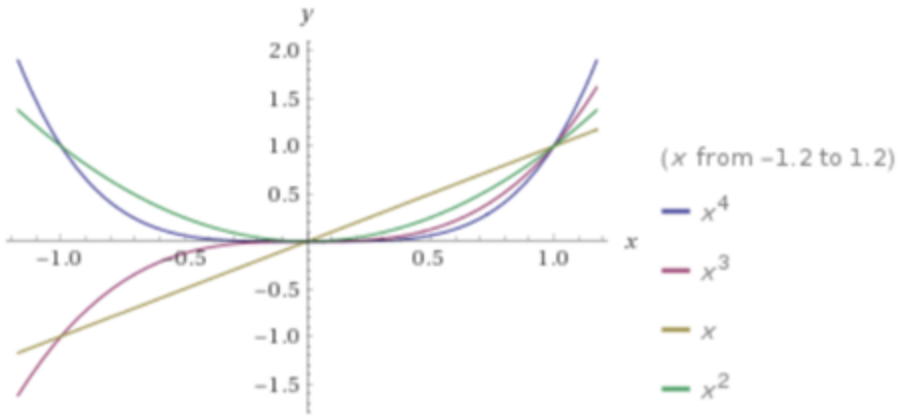
для всех $n \geq n_o$

Решим пример

Доказать, что $n^2 + 2n + 1 = O(n^2)$. Найти c, n_0.

Доказать, что $n^2 + 2n + 1 = O(n^2)$. Найти c, n_0.

$n^2 + 2n + 1 \leq C \cdot n^2$



при $n \geq n_0$, $n_0 = 1$ $\quad n^2 \geq n$

$n^2 \geq 1$

$n^2 + 2n^2 + n^2 \leq C \cdot n^2$

$4n^2 \leq Cn^2$

$n_0 = 1$, $C = 4 \Rightarrow \exists\, c > 0, n_0 > 0 : \forall n > n_0 \quad |f(n)| \leq c \cdot |g(n)|$

ч.т.д.

```
i = 0
while i < n and A[i] != x:
    i += 1

if i < n:
    print('Yes')
else:
    print('No')
```

Логарифм

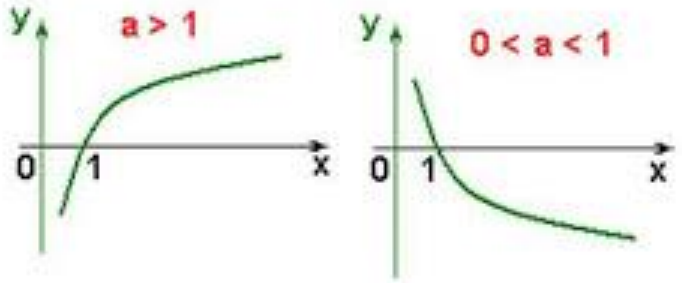
Логарифм

$\log_a x = b$, если $a^b = x$

$\log_a a^b = b$	$a^{\log_a x} = x$

Логарифмическая функция

$y = \log_a x \qquad a > 0,\ a \neq 1$



$\lg x = \log_{10} x$	$e^{\ln x} = x$
$\ln x = \log_e x$	$e^{x \ln a} = a^x$
$e = 2{,}71828\ldots$	$\ln e^x = x$

Свойства

1) $\log_a x\,y = \log_a x + \log_a y$

2) $\log_a \frac{x}{y} = \log_a x - \log_a y$

3) $\log_a x^p = p \cdot \log_a x$

4) $\log_a 1 = 0$ 5) $\log_a a = 1$

6) $\log_{a^n} x = \frac{1}{n} \log_a x$

7) $\log_{a^n} a^m = \frac{m}{n}$

8) $\log_{(\frac{1}{a})^n} a^m = -\frac{m}{n}$

9) $\log_{a^n} (\frac{1}{a})^m = -\frac{m}{n}$

10) $\log_b x = \frac{\log_a x}{\log_a b}$

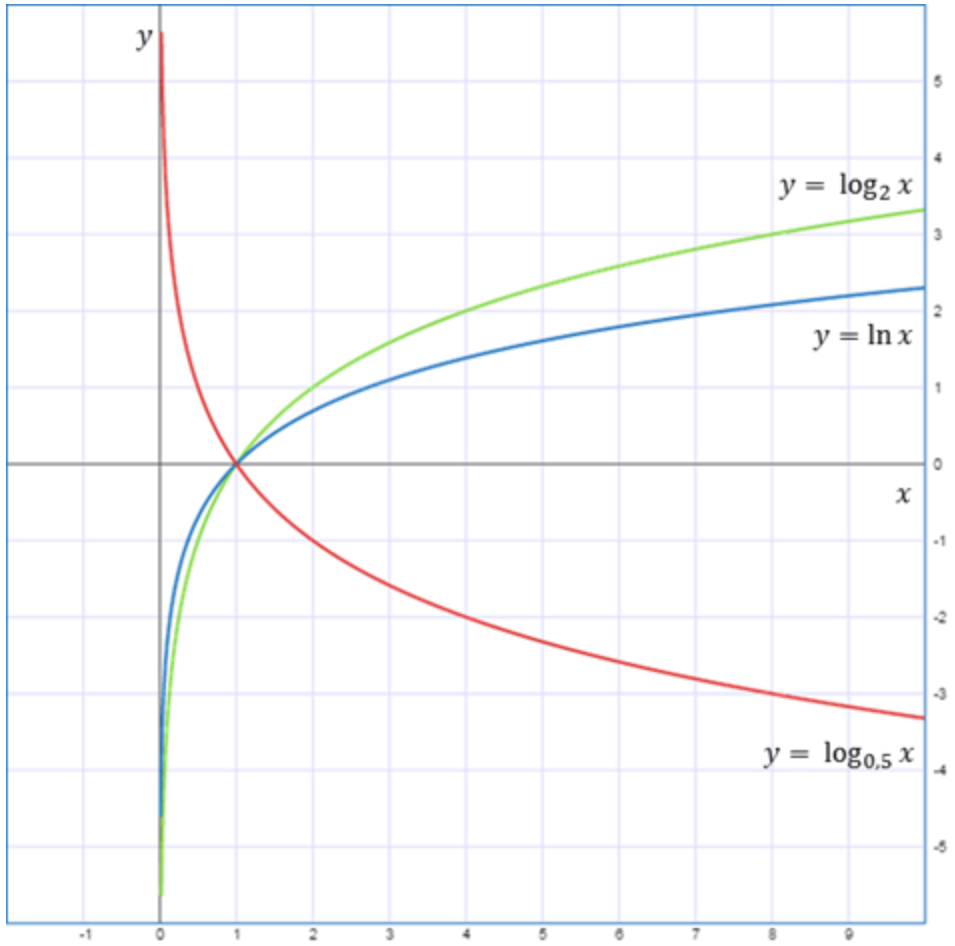
$\log_2 8 = 3,$ так как $2^3 = 8;$

$\log_7 49 = 2,$ так как $7^2 = 49;$

$\log_5 \frac{1}{5} = -1,$ так как $5^{-1} = \frac{1}{5};$

$\log_3 \sqrt{3} = \frac{1}{2},$ так как $3^{\frac{1}{2}} = \sqrt{3}.$

Логарифм



Решим пример

Метод сортировки со сложностью «Big-O» O (n log n) тратит ровно 1 миллисекунду на сортировку 1000 элементов данных. Предполагая, что время T(n) сортировки n элементов прямо пропорционально nlog n, то есть T (n) = c n log n, выведите формулу для T (n), учитывая время T (N) для сортировки N и оцените, как долго этот метод будет сортировать 1 000 000 элементов.

Решим пример

Метод сортировки со сложностью «Big-O» O (n log n) тратит ровно 1 миллисекунду на сортировку 1000 элементов данных. Предполагая, что время T(n) сортировки n элементов прямо пропорционально nlog n, то есть T (n) = c n log n, выведите формулу для T (n), учитывая время T (N) для сортировки N и оцените, как долго этот метод будет сортировать 1 000 000 элементов.

$$T(N) = c \cdot N \cdot \log N$$

$$T(n) = c \cdot n \cdot \log n \Rightarrow c = \frac{T(n)}{n \cdot \log(n)}$$

$$T(N) = \frac{T(n) \cdot N \cdot \log N}{n \cdot \log n}; \quad T(10^6) = \frac{10^3 \cdot 10^6 \cdot 6}{10^6 \cdot 3} = 2000 \text{ ms}$$

Решим пример

Предположим, что каждое из приведенных ниже выражений дает время обработки T (n), затраченное алгоритмом на решение задачи размера n. Выберите доминирующие члены, имеющие наибольший рост n, и укажите наименьшую сложность Big-Oh для каждого алгоритма.

Выражение	Доминирующий член	$O(\dots)$
$5 + 0.001n^3 + 0.025n$		
$500n + 100n^{1.5} + 50n \log_{10} n$		
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$		
$n^2 \log_2 n + n (\log_2 n)^2$		
$n \log_3 n + n \log_2 n$		
$3 \log_3 n + \log_2 \log_2 \log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$2n + n^{0.5} + 0.5n^{1.25}$		
$0.01n \log_2 n + n (\log_2 n)^2$		
$100n \log_3 n + n^3 + 200n$		
$0.003 \log_4 n + \log_2 \log_2 n$		

Решим пример

Предположим, что каждое из приведенных ниже выражений дает время обработки T (n), затраченное алгоритмом на решение задачи размера n. Выберите доминирующие члены, имеющие наибольший рост n, и укажите наименьшую сложность Big-Oh для каждого алгоритма.

Выражение	Доминирующий член	$O(\dots)$
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10} n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2 \log_2 n + n\,(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log n)$
$n\log_3 n + n\log_2 n$	$n\log_3 n\,, n\log_2 n$	$O(n\log n)$
$3\log_3 n + \log_2\log_2\log_2 n$	$3\log_3 n$	$O(\log n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01n\log_2 n + n\,(\log_2 n)^2$	$n\,(\log_2 n)^2$	$O(n(\log n)^2)$
$100n\log_3 n + n^3 + 200n$	n^3	$O(n^3)$
$0.003\log_4 n + \log_2\log_2 n$	$0.003\log_4 n$	$O(\log n)$

Решим пример

Алгоритмы A и B тратят ровно $T_A(n) = c_A n \log_2 n$ и $T_B(n) = c_B n^2$ микросекунды соответственно для задачи размера n. Найдите лучший алгоритм для обработки n = 220 элементов данных, если алгоритм A тратит 10 микросекунд на обработку 1024 элементов, а алгоритм B тратит всего 1 микросекунду на обработку 1024 элементов.

Решим пример

Алгоритмы A и B тратят ровно $T_A(n) = c_A n \log_2 n$ и $T_B(n) = c_B n^2$ микросекунды соответственно для задачи размера n. Найдите лучший алгоритм для обработки n = 220 элементов данных, если алгоритм A тратит 10 микросекунд на обработку 1024 элементов, а алгоритм B тратит всего 1 микросекунду на обработку 1024 элементов.

The constant factors for **A** and **B** are:

$$c_{\mathrm{A}} = \frac{10}{1024 \log_2 1024} = \frac{1}{1024}; \quad c_{\mathrm{B}} = \frac{1}{1024^2}$$

Thus, to process $2^{20} = 1024^2$ items the algorithms **A** and **B** will spend

$$T_{\mathrm{A}}(2^{20}) = \frac{1}{1024} 2^{20} \log_2(2^{20}) = 20280\mu\mathrm{s} \text{ and } T_{\mathrm{B}}(2^{20}) = \frac{1}{1024^2} 2^{40} = 2^{20}\mu\mathrm{s},$$

respectively. Because $T_{\mathrm{B}}(2^{20}) \gg T_{\mathrm{A}}(2^{20})$, the method of choice is **A**.

Решим пример

Один из двух пакетов программного обеспечения, A или B, должен быть выбран для обработки очень больших баз данных, каждая из которых содержит до 1012 записей. Среднее время обработки пакетом A составляет $T_A(n) = 0{,}1n \log_2 n$ микросекунд, а среднее время обработки пакетом B составляет $T_B(n) = 5n$ микросекунд. Какой алгоритм имеет лучшую производительность в смысле «Big-O»? Определите точные условия, когда эти пакеты превосходят друг друга.

Решим пример

Один из двух пакетов программного обеспечения, A или B, должен быть выбран для обработки очень больших баз данных, каждая из которых содержит до 1012 записей. Среднее время обработки пакетом A составляет $T_A(n) = 0{,}1n \log_2 n$ микросекунд, а среднее время обработки пакетом B составляет $T_B(n) = 5n$ микросекунд. Какой алгоритм имеет лучшую производительность в смысле «Big-O»? Определите точные условия, когда эти пакеты превосходят друг друга.

In the "Big-Oh" sense, the algorithm **B** of complexity $O(n)$ is better than **A** of complexity $O(n \log n)$. he package **B** has better performance in a The package **B** begins to outperform **A** when $(T_{\mathrm{A}}(n) \geq T_{\mathrm{B}}(n)$, that is, when $0.1n \log_2 n \geq 5 \cdot n$. This inequality reduces to $0.1 \log_2 n \geq 5$, or $n \geq 2^{50} \approx 10^{15}$. Thus for processing up to 10^{12} data items, the package of choice is **A**.

Решим пример

Пусть время обработки алгоритма сложности Big-O $(f(n))$ прямо пропорционально $f(n)$. Пусть три таких алгоритма A, B и C имеют временную сложность $O(n^2)$, $O(n^{1.5})$ и $O(n\ log\ n)$ соответственно. Во время теста каждый алгоритм тратит 10 секунд на обработку 100 элементов данных. Найдите время, которое каждый алгоритм потратит на обработку 10000 элементов.

Решим пример

Пусть время обработки алгоритма сложности Big-O ($f(n)$) прямо пропорционально $f(n)$. Пусть три таких алгоритма A, B и C имеют временную сложность $O\,(n^2)$, $O\,(n^{1.5})$ и $O\,(n\,log\,n)$ соответственно. Во время теста каждый алгоритм тратит 10 секунд на обработку 100 элементов данных. Найдите время, которое каждый алгоритм потратит на обработку 10000 элементов.

	Complexity	Time to process 10,000 items
A1	$O(n^2)$	$T(10,000) = T(100) \cdot \frac{10000^2}{100^2} = 10 \cdot 10000 = 100,000$ sec.
A2	$O(n^{1.5})$	$T(10,000) = T(100) \cdot \frac{10000^{1.5}}{100^{1.5}} = 10 \cdot 1000 = 10,000$ sec.
A3	$O(n \log n)$	$T(10,000) = T(100) \cdot \frac{10000 \log 10000}{100 \log 100} = 10 \cdot 200 = 2,000$ sec.

Пример

```
def sum(n):
    if n == 1:
        return 1
    return n + sum(n - 1)
```

Пример

```
def sum(n):
    if n == 1:
        return 1
    return n + sum(n - 1)
```

```
def pairSumSequence(n):
    sum = 0
    for i in range(n):
        sum += pairSum(i, i + 1)
    return sum

def pairSum(a, b):
    return a + b
```

Пример

```
for a in arrA:
    print(a)

for b in arrB:
    print(b)
```

```
for a in arrA:
    for b in arrB:
        print(str(a) + "," + str(b))
```

Пример

```
def foo(array):
    sum = 0
    product = 1
    for i in range(len(array)):
        sum += array[i]
    for i in range(len(array)):
        product *= array[i]
    print(str(sum) + "," + str(product))
```

Пример

```
def printPairs(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            print(str(arr[i]) + "," + str(arr[j]))
```

Пример

```
for i in range(N):
    for j in range(N):
        foo()
```

```
for i in range(N):
    for j in range(i, N):
        foo()
```

Пример

```
for i in range(N):
    for j in range(N):
        foo()
```

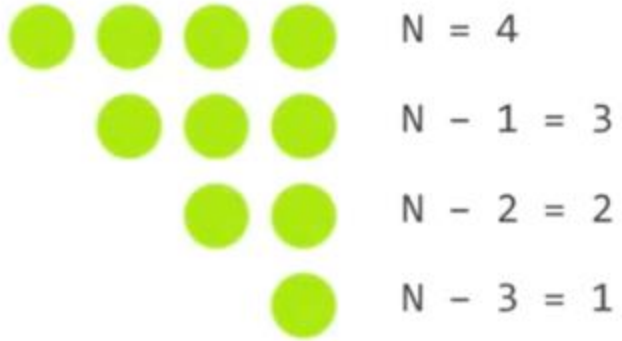
Пусть **N = 4**, тогда:

Сложность выполнения кода

N^2

```
for i in range(N):
    for j in range(i, N):
        foo()
```

Пусть **N = 4**, тогда:



Сложность выполнения кода

$$\frac{N^2}{2} \approx N^2$$

Пример

```
def printPairs(arrA, arrB):
    for i in range(len(arrA)):
        for j in range(len(arrB)):
            if arrA[i] < arrB[j]:
                print(str(arrA[i]) + "," + str(arrB[j]))
```

Пример

```
def printPairs(arrA, arrB):
    for i in range(len(arrA)):
        for j in range(len(arrB)):
            for k in range(100000):
                print(str(arrA[i]) + "," + str(arrB[j]))
```

Пример

```
def reverse(arr):
    for i in range(len(arr) // 2):
        other = len(arr) - i - 1
        temp = arr[i]
        arr[i] = arr[other]
        arr[other] = temp
```

Поиск в массиве

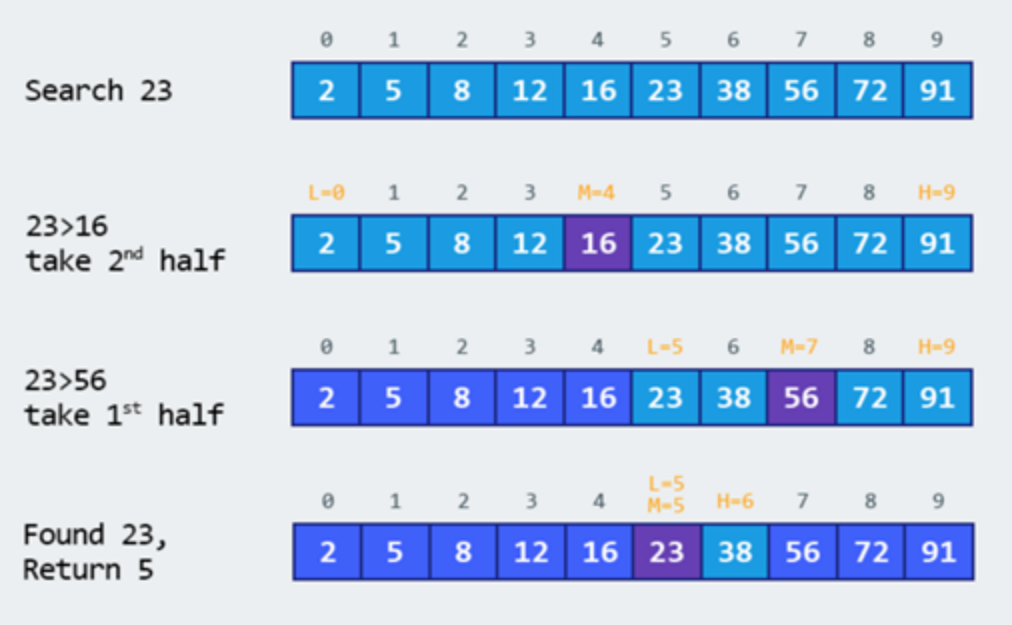
А теперь мы добавим в наш пример одно уточнение: массив, в котором тебе нужно найти число, отсортирован по возрастанию.

В верхнем ряду на изображении мы видим отсортированный массив. В нем нам необходимо найти число 23. Вместо того, чтобы перебирать числа, мы просто делим массив на 2 части и проверяем среднее число в массиве. Находим число, которое располагается в ячейке 4 и проверяем его (второй ряд на картинке). Это число равно 16, а мы ищем 23. Текущее число меньше. Что это означает? Что все предыдущие числа (которые расположены до числа 16) можно не проверять: они точно будут меньше того, которое мы ищем, ведь наш массив отсортирован! Продолжим поиск среди оставшихся 5 элементов.

Поиск в массиве

Обрати внимание: мы сделали всего одну проверку, но уже отмели половину возможных вариантов. У нас осталось всего 5 элементов. Мы повторим наш шаг — снова разделим оставшийся массив на 2 и снова возьмем средний элемент (строка 3 на рисунке). Это число 56, и оно больше того, которое мы ищем. Что это означает? Что мы отметаем еще 3 варианта — само число 56, и два числа после него (они точно больше 23, ведь массив отсортирован). У нас осталось всего 2 числа для проверки (последний ряд на рисунке) — числа с индексами массива 5 и 6. Проверяем первое из них, и это то что мы искали — число 23! Его индекс = 5!

Поиск в массиве



Давай рассмотрим результаты работы нашего алгоритма, а потом разберемся с его сложностью. (Кстати, теперь ты понимаешь, почему его называют двоичным: его суть заключается в постоянном делении данных на 2). Результат впечатляет! Если бы мы искали нужное число линейным поиском, нам понадобилось бы 10 проверок, а с двоичным поиском мы уложились в 3! В худшем случае их было бы 4, если бы на последнем шаге нужным нам числом оказалось второе, а не первое. А что с его сложностью? Это очень интересный момент :)

https://javarush.com/groups/posts/2325-slozhnostjh-algoritmov

Поиск в массиве

Алгоритм двоичного поиска гораздо меньше зависит от числа элементов в массиве, чем алгоритм линейного поиска (то есть, простого перебора). При **10** элементах в массиве линейному поиску понадобится максимум 10 проверок, а двоичному — максимум 4 проверки. Разница в 2,5 раза. Но для массива в **1000 элементов** линейному поиску понадобится 1000 проверок, а двоичному — **всего 10**! Разница уже в 100 раз! Обрати внимание: число элементов в массиве увеличилось в 100 раз (с 10 до 1000), а количество необходимых проверок для двоичного поиска увеличилось всего в 2,5 раза — с 4 до 10. Если мы дойдем до **10000 элементов**, разница будет еще более впечатляющей: 10000 проверок для линейного поиска, и **всего 14 проверок** для двоичного. И снова: число элементов увеличилось в 1000 раз (с 10 до 10000), а число проверок увеличилось всего в 3,5 раза (с 4 до 14). **Сложность алгоритма двоичного поиска логарифмическая**, или,если использовать обозначения Big-O, — **O(log n)**.

Array Size	Linear — N	Binary — Log_2N
10	10	4
50	50	6
100	100	7
500	500	9
1000	1000	10
2000	2000	11
3000	3000	12
4000	4000	12
5000	5000	13
6000	6000	13
7000	7000	13
8000	8000	13
9000	9000	14
10000	10000	14

Оценки сложности

1. *O(1)* – постоянное время
2. *O(log(n))* – логарифмическое время
3. *O(n)* – линейное время
4. *O(n log(n))* – «n-log-n» время
5. $O(n^2)$ – квадратичное время
6. $O(n^3)$ – кубическое время
7. $O(c^n)$ – экспоненциальное время
8. *O(n!)* – факториальное время

A. *Сложение двух чисел*
B. *Двоичный поиск*
C. *Просмотр элементов массива*
D. *Некоторые сортировки*
E. Просмотр пар элементов (сортировка вставками)
F. Динамическое программирование
G. Перечисление всех подмножеств множества
H. Все перестановки n элементов

Что лучше $\boldsymbol{n^2}$ или $\boldsymbol{n\ log(n)}$?

Что лучше $\boldsymbol{n^{100}}$ или $\boldsymbol{2^n}$?

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	2×10^8 веков	$1{,}3 \times 10^{13}$ веков

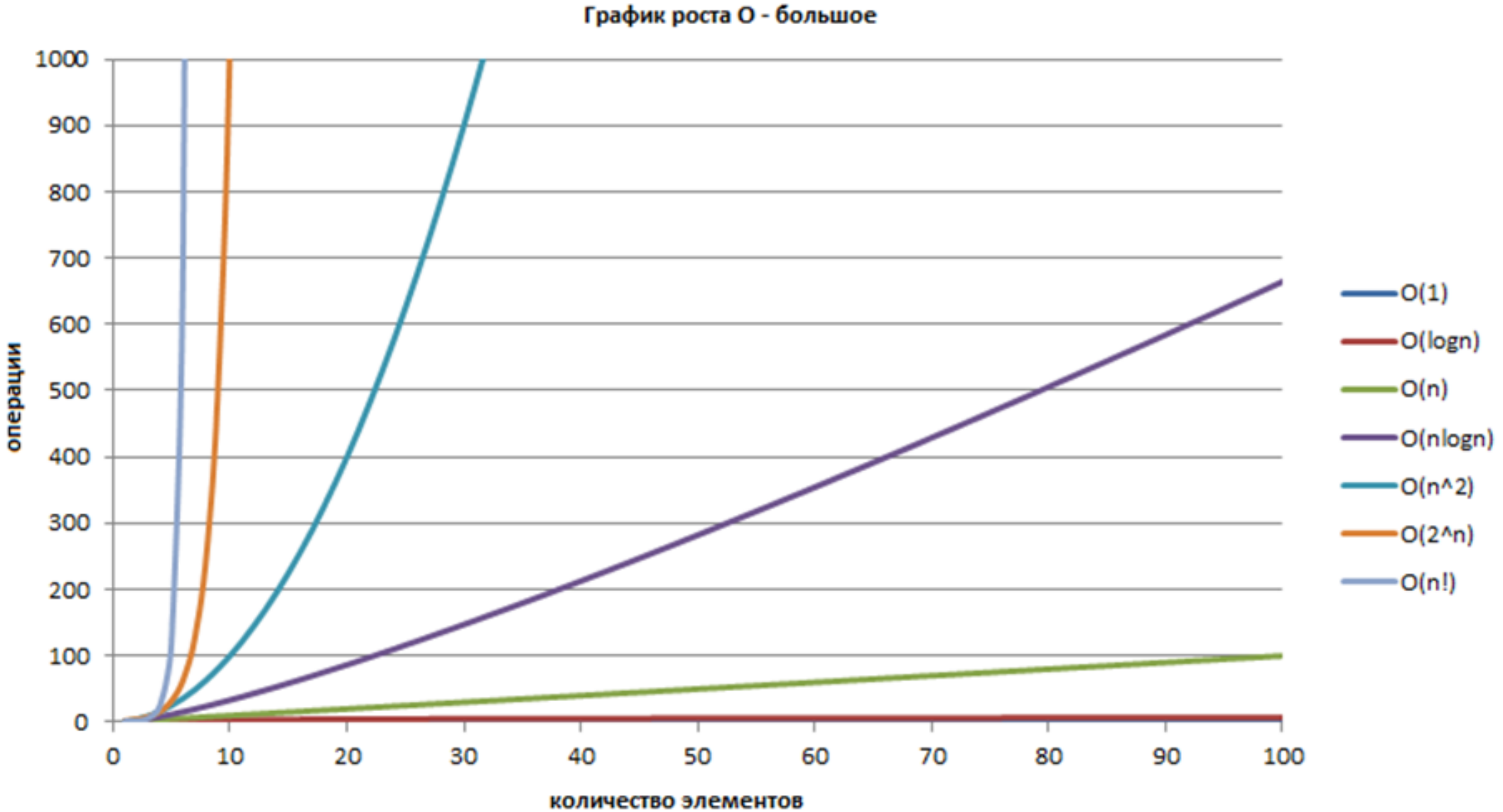
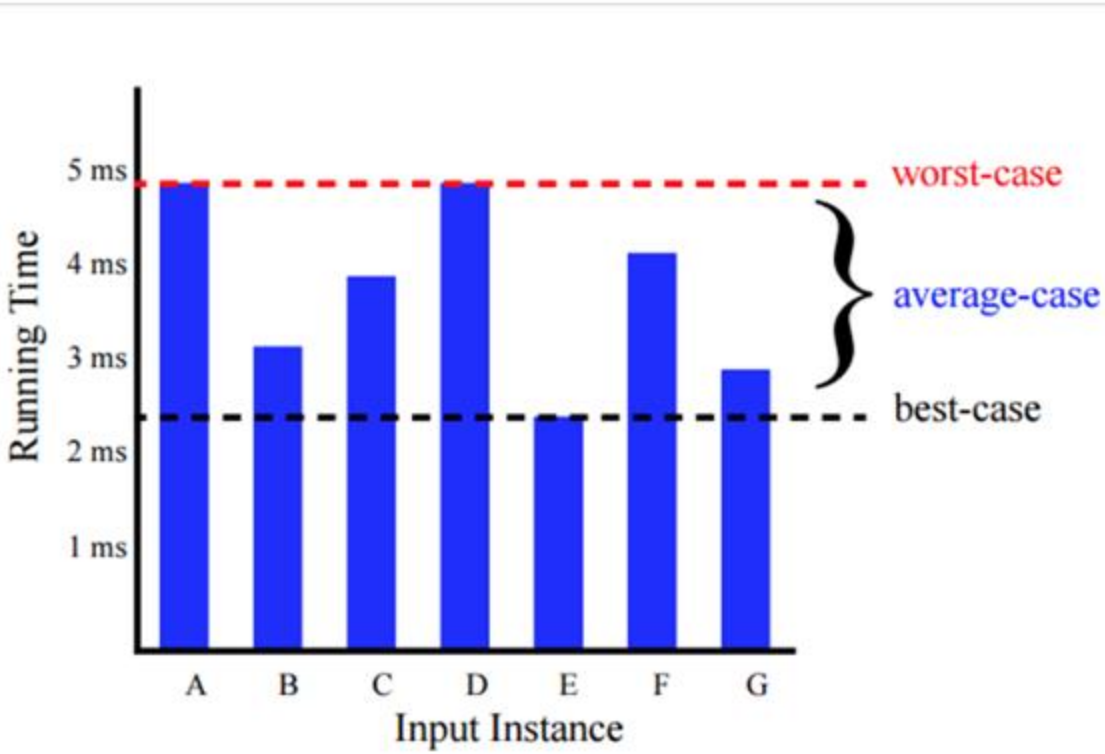
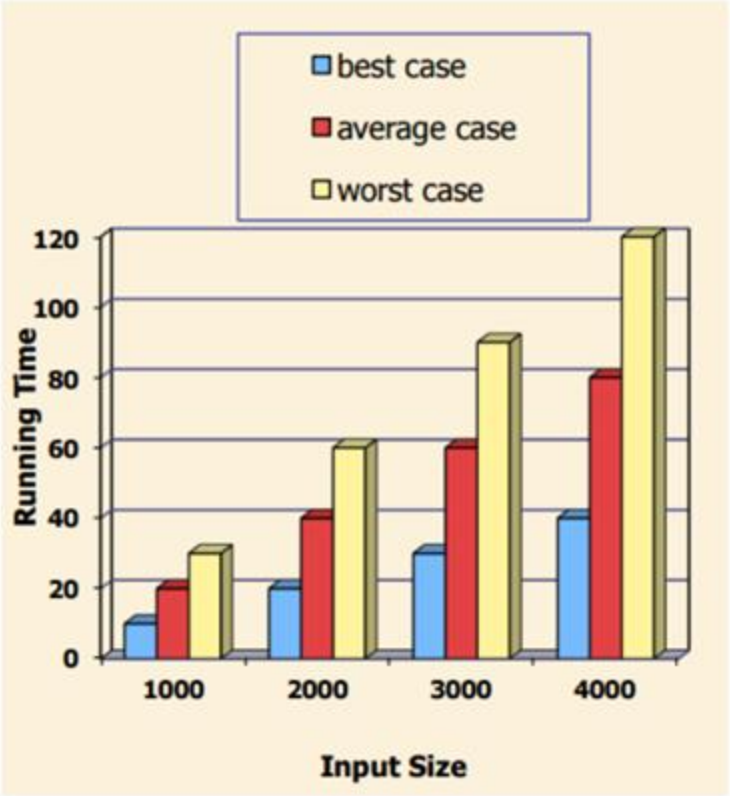


График роста O - большое
операции
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
количество элементов
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Худший, средний и лучший случаи





Пример – последовательный поиск

```
def linear_search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

Алгоритм	Структура данных	Временная сложность			Вспомогательные данные
		Лучшее	В среднем	В худшем	В худшем
Быстрая сортировка	Массив	O(n log(n))	O(n log(n))	O(n^2)	O(n)
Сортировка слиянием	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Пирамидальная сортировка	Массив	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Пузырьковая сортировка	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка вставками	Массив	O(n)	O(n^2)	O(n^2)	O(1)
Сортировка выбором	Массив	O(n^2)	O(n^2)	O(n^2)	O(1)
Блочная сортировка	Массив	O(n+k)	O(n+k)	O(n^2)	O(nk)
Поразрядная сортировка	Массив	O(nk)	O(nk)	O(nk)	O(n+k)

Полиномиально эквивалентные функции

Функции f(x) и g(x) называются *полиномиально эквивалентными*, если существуют такие полиномы p(x) и p′(x), что (начиная с некоторого числа)

$$f(x) \leq p(g(x))$$
$$g(x) \leq p'(f(x)).$$

Если функции f(x) и g(x) принадлежат разным классам, то функция с более быстрым темпом роста доминирует над менее быстро растущей функцией

Классы сложности алгоритмов

Понятие в теории алгоритмов, обозначающее функцию зависимости объёма работы, которая выполняется некоторым алгоритмом, от размера входных данных.

https://ru.wikipedia.org/wiki/Вычислительная_сложность

Классы P и NP

Класс P

Проблемы, решение которых считается «быстрым», то есть полиноминально зависящим от размера входных данных (например, $O(n)$, $O(n^2)$)

Примеры:

- Сортировка
- Поиск во множестве
- Выяснение связности графов

Класс NP

Проблемы, для решения которых используются лишь алгоритмы, экспоненциально зависящие от размера входных данных (например, $O(2^n)$)

Примеры:

- Задача коммивояжера
- Задача выполнимости булевых формул

Классы ?

Проблемы, для которых сводимость к классу P-проблем
и NP-проблем не доказана

Примеры:

- Факторизация длинных чисел
- Дискретное логарифмирование