Алгоритмы и структуры данных

Анализ сложности алгоритмов

Вопросы лекции

- Анализ сложности алгоритмов
- Временная сложность
- Асимптотичиская сложность

" Поразительно, скольким программистам приходится слишком дорогим способом выяснять, что их программа не может обработать входные данные раньше, чем через несколько дней машинного времени. Лучше было бы предугадать такие случаи с помощью карандаша и бумаги. "

С. Гудман (S. Goodman), С. Хидетниеми (S. Hedetniemi)

Для решения большинства проблем существует много различных алгоритмов.

Какой из них выбрать для решения конкретной задачи?

Анализ сложности необходим для получения оценок или границ для объема и времени работы, необходимых алгоритму для успешной обработки входных данных.

Т.е.

1. Сколько времени нужно алгоритму для выполнения.
2. Сколько ресурсов (памяти) требуется для выполнения.

В чем можно измерять сложность алгоритмов?

Сложность алгоритмов можно измерять в минутах, секундах. Но…

Время выполнения алгоритма зависит от частоты процессора, архитектуры процессора.

Один и тот же алгоритм на разных компьютерах будет выполняться за разное время.

Можно измерять количеством выполненных операций.

Что считать операцией?

Например.

Сложение двух чисел одна операция?

Если числа 64 бита, а процессор 32 бита, то будет 3 операции.

1. *Сложение младшей части чисел.*
2. *Сложение старшей части чисел.*
3. *Операция переноса.*

Количество операций понятие тоже не точное .

Сколько операций выполнит программа логично считать от размера входных данных n.

Оценим время выполнения программы через количество операций следующим образом:

$$\left.\begin{array}{l} 34n^3 + 10n^2 + 234 \\ 2n^4 + 25 \end{array}\right\} операций$$

Какая из этих программ «лучше» ???

- Зависти от значения n.
- При малых значения n все программы работают одинаково. Эта проблема возникает при больших n.
- Степень у n (n^2, n^3 ...) играет большую роль, чем коэффициенты.
- Программа с n^4 работает «хуже» чем n^3 .

- Предположим, что есть N программ, которые работают за n, n^2 , n^3 , и 2^n операций.

- Увеличим размер данных n, с которыми работает программа в 10 раз.

Во сколько раз будет медленнее работать программы?

n	**Увеличится в 10 раз**	**1-я программа**
n^2	Увеличится в 100 раз	2-я программа
n^3	Увеличится в 1000 раз	3-я программа
2^n	*Если на 10 то в 1024* *Если в 10 то сложно оценить*	4-я программа

- Если n^k , где к некоторая константа, то говорят, что программа работает за полином. Или ***полиномиальное время работы.***

- Если 2^n (a^n) , где к некоторая константа, то говорят, что программа работает за ***экспоненциальное время.***

Еще пример

Что быстрее: $50N^2 + 31N^3 + 24N + 15$ или

$3N^2 + N + 21 + 4*3^N$?

Ответ зависит от значения N:

N	$50N^2 + 31N^3 + 24N + 15$	$3N^2 + N + 21 + 4*3^N$
1	120	37
2	511	71
3	1374	159
4	2895	397
5	5260	1073
6	8655	3051
7	13266	8923
8	19279	26465
9	26880	79005
10	36255	236527

Чем объяснить???

N	$3N^2 + N + 21 + 4*3^N$	$4*3^N$	% от суммы
1	37	12	32,4
2	71	36	50,7
3	159	108	67,9
4	397	324	81,6
5	1073	972	90,6
6	3051	2916	95,6
7	8923	8748	98,0
8	26465	26244	99,2
9	79005	78732	99,7
10	236527	236196	99,9

Вывод.

Количество элементарных операций, затраченных алгоритмом для решения конкретной задачи, зависти не только от размера входных данных, но и от самих данных.

Время работы алгоритма зависит :

1. От размера входных данных.
2. От самих данных.

Теория сложности вычислений возникла из потребности:

- сравнивать быстродействие алгоритмов;
- четко описывать их поведение (время исполнения и объем необходимой памяти) в зависимости от размера кода.

Вычислительная сложность - мера использования алгоритмом ресурсов времени или пространства.

Время выполнения алгоритма определяется количеством элементарных шагов, необходимых для решения задачи и зависит от размера входных данных.

Пространство – объем памяти или места на носителе данных, используемых алгоритмом (пространственная сложность).

Анализ алгоритмов

Анализ алгоритма предполагает получение представление о том, сколько ***времени*** будет затрачено на решение задачи с использованием этого алгоритма.

При оценке алгоритма исходят из *количества* наиболее значимых для данного алгоритма *операций*, выполняемых в ходе его работы.

Экспериментальные исследования имеют три **основных** ограничения:

1. Эксперименты могут проводиться лишь с использованием ограниченного набора исходных данных; результаты, полученные с использованием другого набора, не учитываются;

Экспериментальные исследования имеют три **основных** ограничения:

2. Для сравнения эффективности двух алгоритмов необходимо, чтобы эксперименты по определению времени их выполнения проводились на одинаковом аппаратном и программном обеспечении;

3. Для экспериментального изучения времени выполнения алгоритма необходимо провести его реализацию и выполнение.

Общая методология анализа времени выполнения алгоритмов:

1. Описание алгоритма на псевдокоде.
2. Определение числа операций на основе анализа структурных конструкций псевдокода.
3. Определение общего числа операций алгоритма.

Анализ алгоритмов

Общая методология анализа времени выполнения алгоритмов:

- учитывает различные типы входных данных;
- позволяет производить оценку относительной эффективности любых двух алгоритмов независимо от аппаратного и программного обеспечения;
- может проводиться по описанию алгоритма без его непосредственной реализации или экспериментов.

Анализ алгоритмов. Аналитический подход

1. *Записать* алгоритм в виде кода одного из языков программирования высокого уровня.

2. *Перевести* программу в последовательность машинных команд (например, байт-коды, используемые в виртуальной машине Java).

3. *Определить* для каждой машинной команды i время t_i, необходимое для ее выполнения.

Анализ алгоритмов. Аналитический подход

4. *Определить* для каждой машинной команды i количество повторений n_i команды i за время выполнения алгоритма.

5. *Определить* произведение $t_i * n_i$ всех машинных команд, что и будет составлять время выполнения алгоритма.

Простейшие(элементарные) операции высокого уровня, не зависящие от используемого языка программирования и использующиеся в псевдокоде:

- присваивание переменной значения
- вызов метода
- выполнение арифметической операции
- сравнение двух чисел
- индексация массива
- переход по ссылке на объект
- возвращение из метода

Пример.

число простейших операций

$T(n)$,

выполняемых алгоритмом

arrayMax,

минимально равно

$2 + 1 + n + 4(n - 1) + 1 = 5n$

а максимально

$2 + 1 + n + 6(n - 1) + 1 = 7n - 2$

```
Алгоритм arrayMax(A, n):
    currentMax ← A [0]
    for i ← 1 to n - 1 do
        if currentMax <A[i] then
            currentMax ← A [i]
    return currentMax
```

Уточнения

- Время выполнения ***операторов присвоения***, чтения и записи обычно имеют порядок O(1).

- Время выполнения ***последовательности операторов*** определяется с помощью правила сумм.

  Степень роста времени выполнения последовательности операторов без определения констант пропорциональности совпадает с ***наибольшим временем выполнения оператора из данной последовательности.***

Уточнения

- Время выполнения ***условного оператора*** состоит из времени вычисления самого логического условия (обычно O(1)) и времени выполнения операторов тела конструкции.

- Время выполнения ***цикла*** является суммой времени всех исполняемых итераций цикла, в свою очередь состоящих из времени выполнения операторов тела цикла и времени вычисления условия прекращения цикла (обычно последнее имеет порядок O (1)).

Структуры и алгоритмы обработки данных

Временная сложность алгоритмов

Временная сложность алгоритмов

Число T(n) простейших операций, выполняемых внутри алгоритма, пропорционально действительному времени выполнения данного алгоритма .

Для многих программ время выполнения является *функцией входных данных* .

Временная сложность алгоритма – функция времени $T(n)$, от размера входных данных n, определяет ***максимальное количество элементарных операций***, которые были проделаны алгоритмом для решения поставленной задачи заданного размера.

Например, некая программа имеет время выполнения

$T(n) = cn^2$, где c — константа.

Единица измерения $T(n)$ точно не определена, обычно понимается под $T(n)$ ***количество инструкций***, выполняемых на идеализированном компьютере.

Временная сложность алгоритмов.

Точное значение временной сложности зависит от определения элементарных операций.

Операции могут быть:

- Арифметические;
- Побитовые;
- Операции на машине Тьюринга.

Эффективность алгоритмов

При оценке эффективности алгоритма обычно стараются оценить три варианта:

- *наилучший случай* (когда упорядочение достигается за наименьшее время);
- *наихудший случай* (когда упорядочение достигается за максимальное время);
- *средний случай*, анализ которого и является обычно самым сложным.

Эффективность алгоритмов

Временная сложность алгоритма

в худшем случае

– функция размера входных данных, которая показывает максимальное количество элементарных операций, которые могут быть затрачены алгоритмом для решения экземпляра задачи указанного размера - O(f(n)).

Временная сложность алгоритма в

наилучшем случае

- функция размера входных данных, которая показывает минимальное количество элементарных операций, которые могут быть затрачены алгоритмом для решения экземпляра задачи указанного размера $\Omega(g(n))$.

число простейших операций T(n), выполняемых алгоритмом arrayMax, минимально равно $2 + 1 + n + 4(n-1) + 1 = 5n$ а максимально $2 + 1 + n + 6(n-1) + 1 = 7n - 2$

Алгоритм arrayMax(A, n):

```
currentMax ← A [0]
for i ← 1 to n - 1 do
    if currentMax <A[i] then
        currentMax ← A [i]
return currentMax
```

Например

```
min = A[0];
for (i = 1; i < n; i ++ )
{
    if (A[i] < min )
     min = A [i]
}
```

Временная сложность алгоритма в ***худшем случае*** – $f(n) = (n-1)*2 +1 = 2n - 1$

Пример входных данных (9,6,4,3,2,1) (8,6,4,5,2,1)

Временная сложность алгоритма в ***наилучшем случае*** – $g(n) = n$

Пример входных данных (1,3,5,6,7) (1,4,6, 8,2,5)

Пример

№	Управляющая структура	Запись на языке Pascal	Вычислительная сложность структуры

1	Составной оператор	`begin`
		S_1;
		S_2;
		...
		`end;`
			$T = \sum_i T(S_i)$

2	Цикл	`for` i := 1 `to` N `do` S; `While` U `do` S; `Repeat` S; `Until` U;	$T = N * T(S)$, где N – число итераций цикла
3	Условие	`if` U `then` S_1 `else` S_2;	$T = T(U) + \begin{cases} T(S_1), если\ U \\ T(S_2), иначе \end{cases}$

Пример анализа вычислительной сложности алгоритма

```
function Search(A: array [1..n] of integer; x:
    integer): integer;
var i: integer;
begin
  //В цикле обход элементов массива
  for i := 1 to n do
    //Если текущий элемент равен искомому
    if A[i] = x then
    begin
      //то вернуть индекс элемента
      result := i;
      //и выйти из процедуры
      exit;
    end;
  //вернуть признак того, что элемент не найден
  result := -1;
end;
```

```
//Алгоритм простого последовательного поиска
    целого числа в массиве
function Search(A: array [1..n] of integer; x:
    integer): integer;
var i: integer;
begin //УС1
  //В цикле обход элементов массива
  for i := 1 to n do //УС2
    //Если текущий элемент равен искомому
    if A[i] = x then //УС3
    begin //УС4
      //то вернуть индекс элемента
      result := i;
      //и выйти из процедуры
      exit;
    end;
  //вернуть признак того, что элемент не найден
  result := -1;
end;
```

Вычислительная сложность управляющих структур алгоритма:

$$T_4 = 1 + 1 = 2$$

$$T_3 = 1 + \begin{cases} T_4, \text{если } A[k] = x \\ 0, \text{иначе} \end{cases} = \begin{cases} 3, \text{если } A[k] = x \\ 1, \text{иначе} \end{cases}$$

$$T_2 = n * T_3 = \begin{cases} (k-1)*1+3, \text{если } A[k] = x \\ n*1, \text{если } x \text{ нет в } A \end{cases} =$$

$$= \begin{cases} k+2, \text{если } A[k] = x \\ n, \text{если } x \text{ нет в } A \end{cases}$$

$$T_{Search} = T_1 = T_2 + 1 = \begin{cases} k+2, \text{если } A[k] = x \\ n+1, \text{если } x \text{ нет в } A \end{cases}$$

Пример

Временная сложность поиска числа в массиве зависит от содержимого массива A, от искомого числа x и количества элементов в массиве n.

$$T_{Search}(\underline{A, x, n}) = \begin{cases} k+2, \text{если } A[k] = x \\ n+1, \text{если } x \text{ нет в } A \end{cases}$$

Пример

1. **Наилучший случай** – искомое число в первой ячейке $T_{Search}(n)=3$
2. **Наихудший случай** – искомое число в последней ячейке $T_{Search}(n)=n+2$
3. **Средний случай** – при равномерной распределении $T_{Search}(n)=(3+(n+2))/2$

Точное знание количества операций, выполняемых алгоритмом, не очень существенно с точки зрения анализа эффективности.

Более важным является ***скорость роста числа операции*** при возрастании ***n – числа элементов массива.***

Быстрорастущие функции ***доминируют в оценке суммарной эффективности алгоритма.***

Если выясняется, что сложность алгоритма представляет собой сумму линейной и квадратичной функций, то скорость роста будет оцениваться как функция, сопоставимая с n^2 .

Структуры и алгоритмы обработки данных

Асимптотический анализ сложности алгоритмов

Асимптотическая сложность алгоритмов

В большинстве случаев ***временная сложность*** алгоритма не может быть определена точно.

Поэтому чаще используется понятие ***асимптотической сложности алгоритма,*** когда подразумевается анализ времени, которое потребуется для обработки очень большого набора данных.

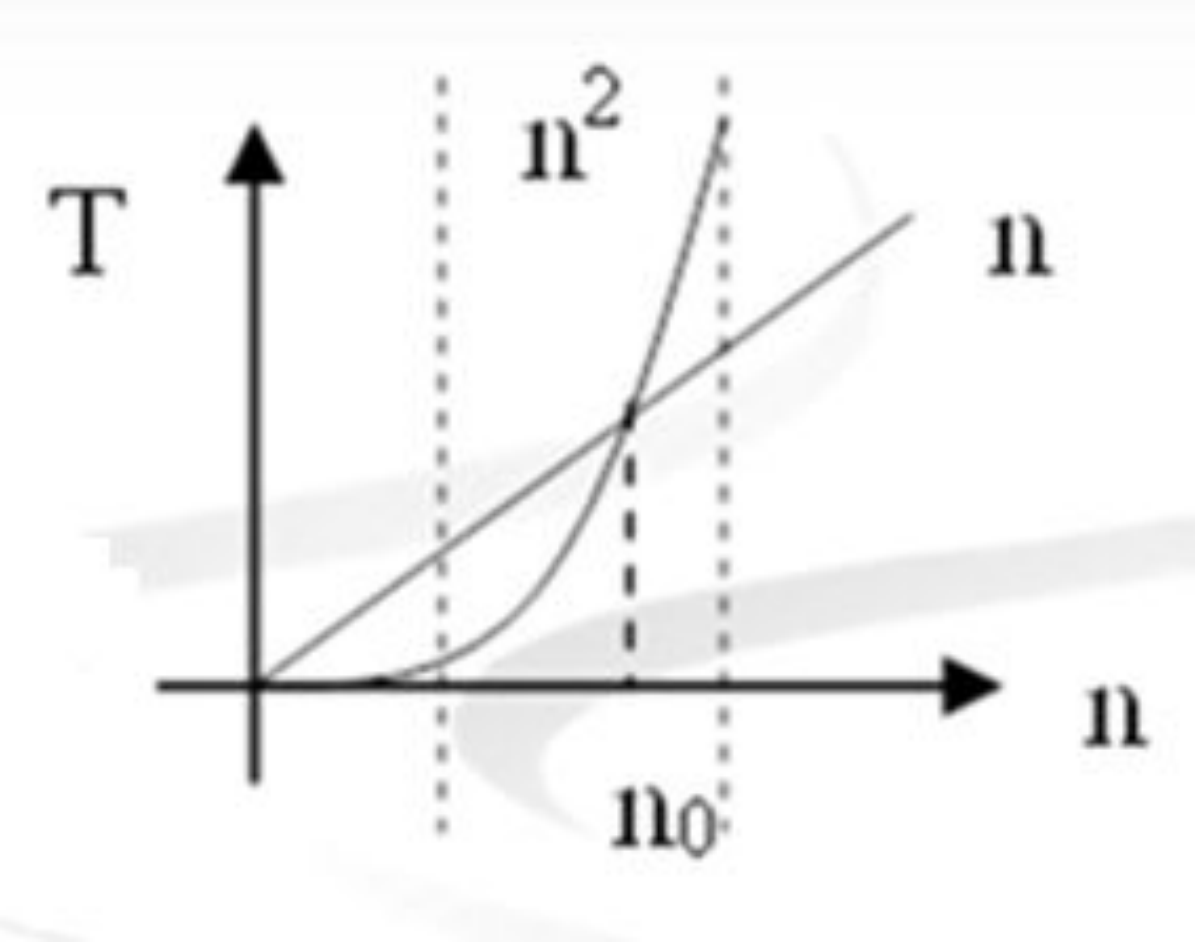
Асимптотическая сложность алгоритмов

Асимптотический анализ справедлив только для больших n.

Для малых n бывают случаи, когда алгоритмы, относящиеся к более эффективному классу, работают медленнее, чем алгоритмы менее эффективного класса.

Например, алгоритм сортировки «пузырьком» при малых n работает быстрее, чем «быстрая» сортировка.

Асимптотическая сложность алгоритмов



При данном анализе возникают вопросы:

1. Сколько времени потребуется на обработку массива из десяти элементов? Тысячи? Десяти миллионов?

2. Если алгоритм обрабатывает тысячу элементов за пять миллисекунд, что случится, если мы передадим в него миллион?

3. Будет ли он выполняться пять минут или пять лет?

При данном анализе следует учитывать:

Порядок роста.

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных.

Чаще всего он представлен в виде О-нотации *(от нем. «Ordnung» - порядок)*:

O(f(x)), где *f(x)* – формула, выражающая сложность алгоритма.

Наиболее часто встречающиеся порядки роста:

Константный – O(1)

Порядок роста *O(1)* означает, что вычислительная сложность алгоритма не зависит от размера входных данных.

Линейный –O(n)

Порядок роста *O(n)* означает, что сложность алгоритма линейно растет с увеличением входного массива.

Если линейный алгоритм обрабатывает один элемент пять миллисекунд, то вероятно ожидать, что тысячу элементов он обработает за пять секунд.

Такие алгоритмы содержат циклы по каждому элементу входного массива.

Логарифмический – O(log n)

Порядок роста *O(*log *n)* означает, что время выполнения алгоритма растет логарифмически с увеличением размера входного массива.

В анализе алгоритмов по умолчанию используется логарифм по основанию 2.

Большинство алгоритмов, работающих по принципу «деления пополам», имеют логарифмическую сложность.

Линеарифметический — O(n·log n)

Линеарифметический (или линейно-логарифмический) алгоритм имеет порядок роста $O(n \cdot \log n)$.

Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию.

Например, сортировка слиянием и быстрая сортировка.

Квадратичный — O(n ^{2})

Время работы алгоритма с порядком роста $O(n^2)$ зависит от квадрата размера входного массива.

Квадратичная сложность – повод пересмотреть используемые алгоритмы или структуры данных.

Проблема в том, что они плохо масштабируются.

Асимптотическая сложность алгоритмов. Квадратичный порядок роста. Пример.

Массив из тысячи элементов потребует 1 000 000 операций.

Массив из миллиона элементов потребует 1 000 000 000 000 операций.

Если одна операция требует миллисекунду для выполнения, квадратичный алгоритм будет обрабатывать миллион элементов 32 года.

Даже если он будет в сто раз быстрее, работа займет 84 дня.

Пример алгоритма с квадратичной сложностью – пузырьковая сортировка.

Асимптотическая сложность алгоритмов. Выводы.

Асимптотическая сложность определяет порядок времени роста отдельно взятого алгоритма.

Для описания времени порядка роста используется О-нотация – математическая запись, которая позволяет учитывать только наиболее весомые элементы функции времени.

Асимптотическая сложность алгоритмов. Определения.

Функция времени T(n) имеет порядок роста O(f(n)), если существует константы c и n_0, такие что для всех выполняется неравенство

$$T(n) \leq cf(n)$$

Асимптотическая сложность алгоритмов

Оценка асимптотической сложности решает задачу масштабируемости данных, т.е. как влияет увеличение объема данных на увеличение времени работы алгоритма.

Асимптотический анализ позволяет оценивать и сравнивать скорость роста функции временной сложности, а так же классифицировать алгоритмы.

№ п.п.(От сложного к простому)	Название сложности	Мат. формула	Примеры алгоритмов
1	Факториальная	N!	Алгоритмы комбинаторики (сочетания, перестановки и т.д.)
2	Экспоненциальная	K^N	Алгоритмы перебора (*brute force*)
3	Полиномиальная	N^K	Алгоритмы простой сортировки (*buble sort*)
4	Линейный логарифм	N * log(N)	Алгоритмы быстрой сортировки (*heap sort*)
5	Линейная	K * N	Перебор элементов массива
6	Логарифмическая	K * log(N)	Бинарный поиск
7	Константная	K	Обращение к элементу массива по индексу

Данные роста функций

Функция	10	100	1000	10000	100000
$\log_2 N$	3	6	9	13	16
N	**10**	**100**	**1000**	**10000**	**100000**
$N \log_2 N$	30	664	9965	10^5	10^6
N^2	10^2	10^4	10^6	10^8	10^{10}
N^3	10^3	10^6	10^9	10^{12}	10^{15}
2^N	10^3	10^{30}	10^{301}	10^{3010}	10^{30103}

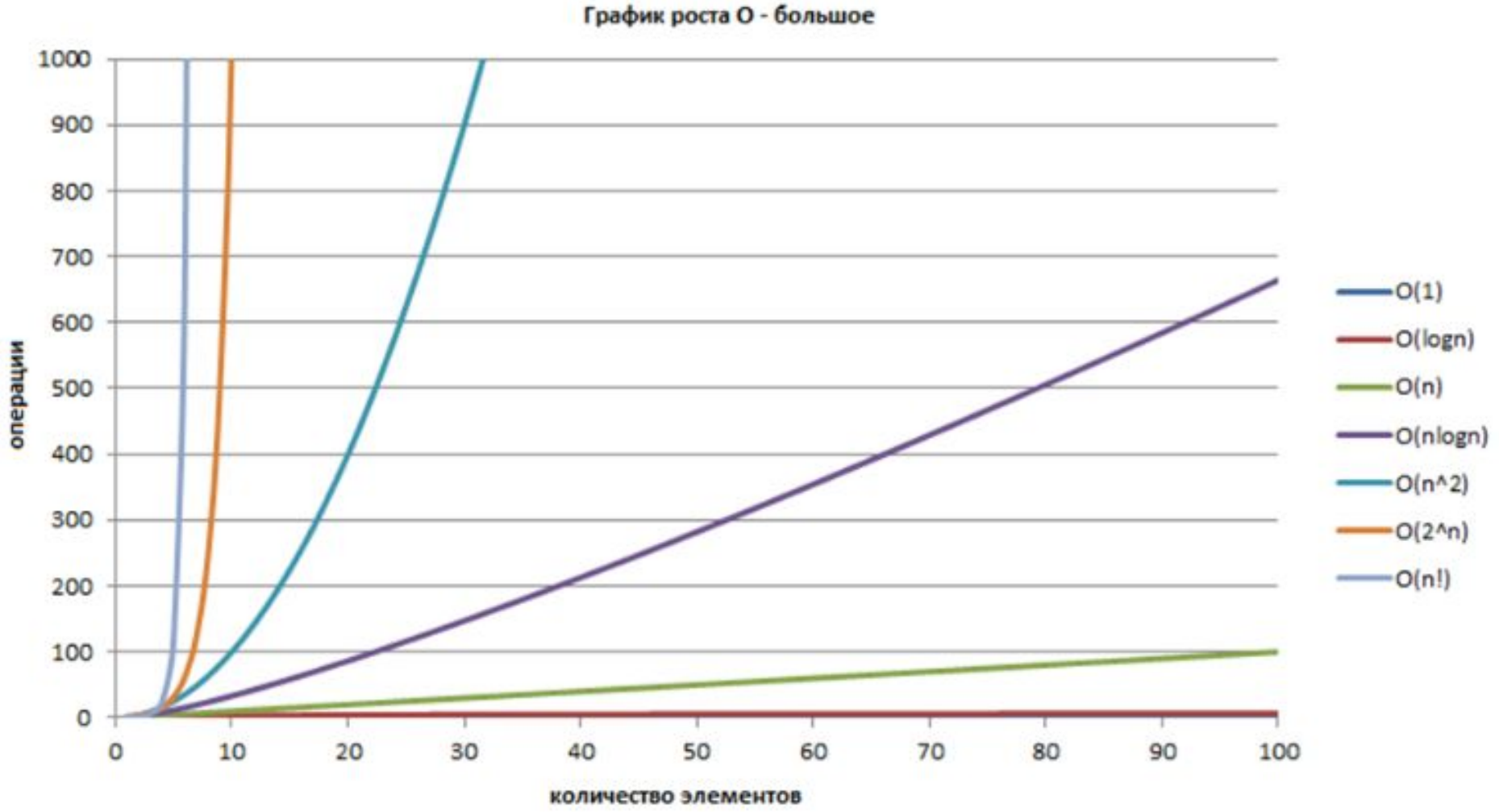


График роста O - большое
1000
900
800
700
600
500
400
300
200
100
0
операции
0
10
20
30
40
50
60
70
80
90
100
количество элементов
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

Временная сложность алгоритмов сортировки

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

Хорошо	Приемлемо	Плохо