

# Оценка сложности алгоритмов

## О-нотация

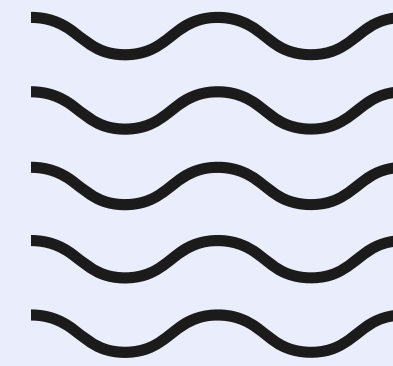
---

ПОДГОТОВИЛ: АЛИШЕР ХАМИДОВ



# План лекции

## Основные моменты:



1. Вычислительная временная сложность
2. Вычислительная ёмкостная сложность
3. Асимптотической оценка функции сложности
4. О-нотация
5. Линейная сложность
6. Логарифмическая сложность
7. Квадратичная сложность
8. Константная сложность



# Вычислительная временная сложность (time complexity)

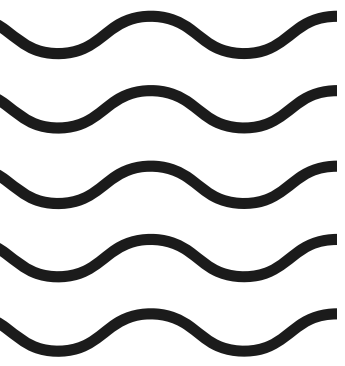
— это максимальное возможное количество  
выполненных алгоритмом элементарных операций,  
как **функция от размера входных данных**.



# Вычислительная ёмкостная сложность (space complexity)



- — это максимальный возможный размер занятой алгоритмом дополнительной памяти, как функция от размера входных данных.






1. Это функция.
2. Зависит от размера входных данных.
3. Возвращает максимум операций/памяти для худшего, самого затратного случая.



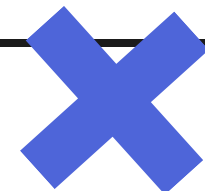


```
for (int i=0; i < n; i++){  
    count++;  
}
```

Посчитаем количество элементарных операций:

- 
- 1 для `int i = 0`
  - $n+1$  для `i < n`
  - $2n$  для `i++` (что эквивалентно `i = i + 1`, а это две операции: присваивание и сложение)
  - $2n$  для `count++`

Получаем, что временную сложность алгоритма  
 $C(n) = 2 + 5n$





# Асимптотической оценка функции сложности




Получить асимптотическую оценку можно так: отбросьте в функции сложности все слагаемые, кроме одного с самой быстрой скоростью роста. А потом отбросьте все константы. То что получится и будет асимптотической оценкой сложности.

- Обычно время работы алгоритма нас интересует на больших данных, когда алгоритм может существенно замедлиться. Асимптотическая оценка как раз отвечает на вопрос, как сильно деградирует производительность с ростом размера входа.



# O-нотация

Использование заглавной буквы O (или так называемая O-нотация) пришло из математики, где её применяют для сравнения асимптотического поведения функций. Формально **O(f(n))** означает, что время работы алгоритма (или объём занимаемой памяти) растёт в зависимости от объёма входных данных не быстрее, чем некоторая константа, умноженная на **f(n)**.





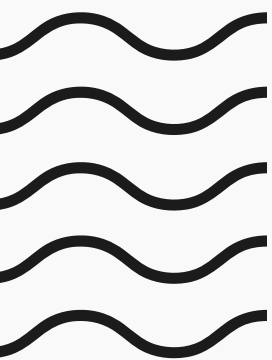
## Линейная сложность — $O(n)$



Такой сложностью обладает, например, алгоритм поиска наибольшего элемента в не отсортированном массиве. Нам придётся пройти по всем  $n$  элементам массива, чтобы понять, какой из них максимальный.

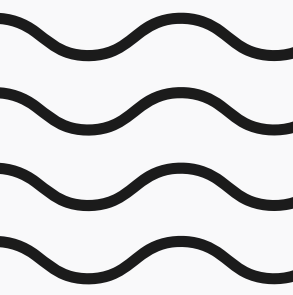
## $O(\log n)$ — логарифмическая сложность

Простейший пример — бинарный поиск. Если массив отсортирован, мы можем проверить, есть ли в нём какое-то конкретное значение, методом деления пополам. Проверим средний элемент, если он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. И так будем продолжать делить пополам, в итоге проверим  $\log n$  элементов.



## Квадратичная сложность — $O(n^2)$

Вспоминаем алгоритм сортировки пузырьком. В классической реализации речь идёт о двух вложенных циклах. В результате, число операций зависит от размера массива как  $n * n$ , то есть  $n^2$ .

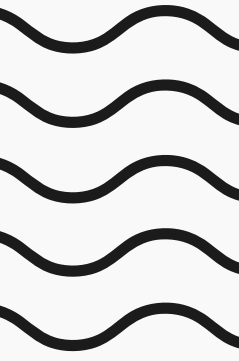
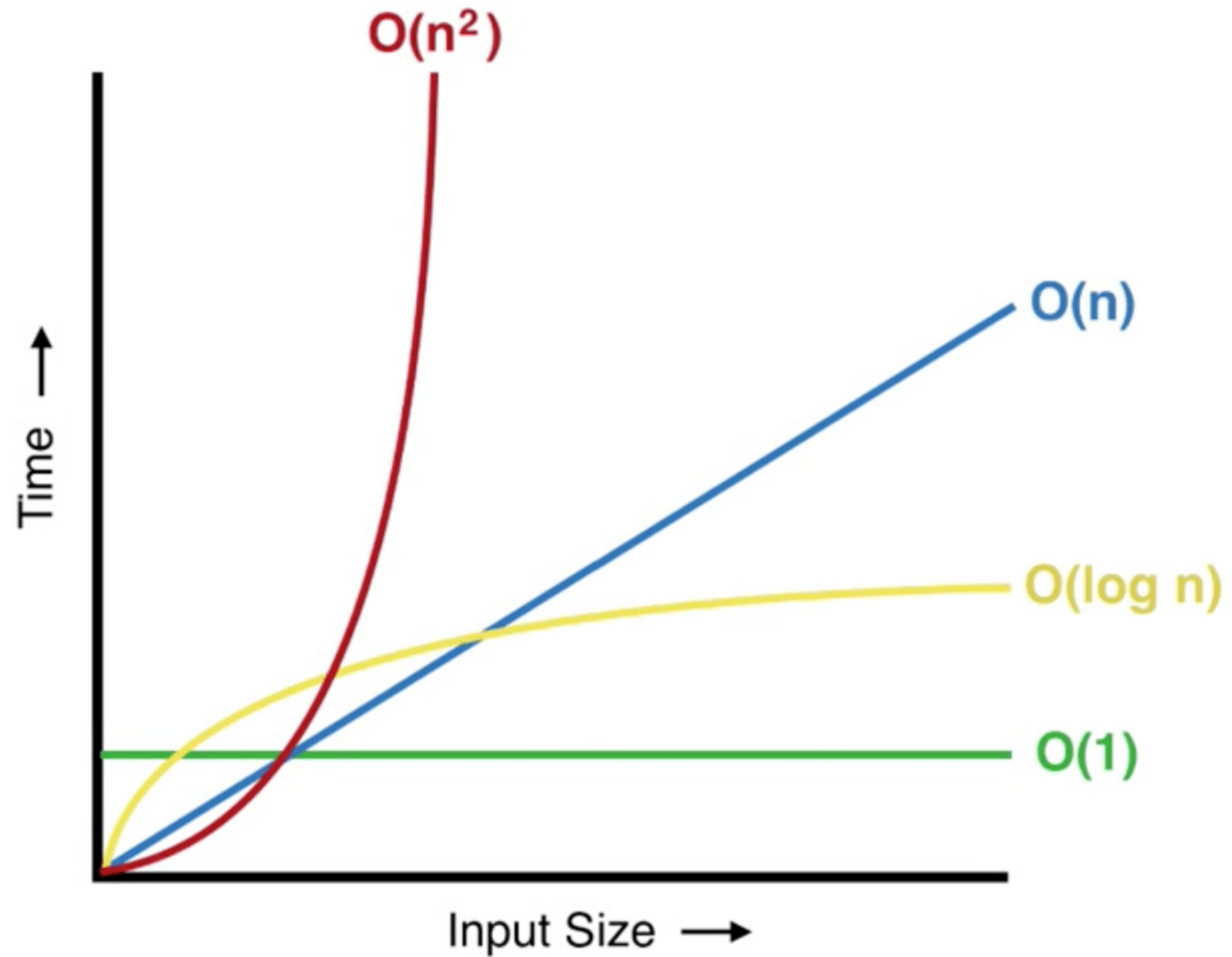


## Константная сложность- $O(1)$



Также бывает, что время работы алгоритма не зависит от размера входных данных. В этом случае сложность обозначается как  $O(1)$ . К примеру, чтобы определить значение третьего элемента массива не надо ни запоминать элементы массива, ни проходить по ним некоторое число раз. Здесь надо дождаться в потоке входных данных 3-й элемент, что и будет итогом, на вычисление которого для любого объёма данных потребуется одно и то же время. Схожим образом проводят оценку по памяти.

## Big O Notation



# Дополнительная литература



- <https://webdevblog.ru/bolshoe-o-cto-eto-takoe-pochemu-eto-vazhno-i-pochemu-eto-ne-vazhno/>
- <https://techrocks.ru/2020/07/15/big-o-explanation-for-newbies/>
-