

# *Algorithms and data structures*

lecture #1. Big-O notation, Simple Problems, Naive Approach

Mentor: Rustam Khakov

# *lecture #1. Big-O notation, Simple Problems, Naive Approach*

- Big-O notation
  - Asymptotic analysis
  - Growth Order
  - Constant -  $O(1)$  (константная)
  - Logarithmic -  $O(\log n)$  (логарифмическая)
  - Linear -  $O(n)$  (линейная)
  - Linearithmic -  $O(n \log n)$  (линейно-логарифмическая)
  - Quadratic -  $O(n^2)$  (квадратичная)
  - $O(n!)$  — (факториальная)
  - Best, average and worst case
  - What are we measuring

# *Асимптотический анализ*

Оцениваем производительность алгоритма с точки зрения размера входных данных по времени выполнения и используемой памяти.

- как мы узнаем, какой алгоритм лучше?
- время (time complexity)
- пространство (space complexity).

# *Асимптотический анализ в действии*

Пример:

задача поиска (поиск заданного элемента) в отсортированном массиве.

линейный поиск (порядок роста — линейный  $O(n)$ )

бинарный поиск (порядок роста — логарифмический  $O(\log n)$ )

Компьютер А — константное время 0.2 сек

Компьютер В — константное время 1000 сек

# Асимптотический анализ в действии

Время выполнения линейного поиска в секундах для A :  $0,2 * n$

Время выполнения двоичного поиска в секундах для B :  $1000 * \log(n)$

-----		
n	time on A	time on B
-----		
10	2 sec	~ 1 h
-----		
100	20 sec	~ 1.8 h
-----		
$10^6$	~ 55.5 h	~ 5.5 h
-----		
$10^9$	~ 6.3 years	~ 8.3 h
-----		

порядок роста бинарного поиска по размеру входных данных является логарифмическим, а порядок роста линейного поиска — линейным.

# *Асимптотический анализ*

- $O$  (О-большое) – верхняя асимптотическая оценка роста временной функции;
- $\Omega$  (Омега) – нижняя асимптотическая оценка роста временной функции;
- $\Theta$  (Тета) – нижняя и верхняя асимптотические оценки роста временной функции.

# Асимптотический анализ - правила

1.  $O(k*f) = O(f)$  –  $k$  (константа) отбрасывается, поскольку с ростом объема данных, его смысл теряется, например:

$$O(9,1n) = O(n)$$

2.  $O(f*g) = O(f)*O(g)$  – оценка сложности произведения двух функций равна произведению их сложностей, например:

$$O(5n*n) = O(5n)*O(n) = O(n)*O(n) = O(n*n) = O(n^2)$$

3.  $O(f/g) = O(f)/O(g)$  – оценка сложности частного двух функций равна частному их сложностей, например:

$$O(5n/n) = O(5n)/O(n) = O(n)/O(n) = O(n/n) = O(1)$$

4.  $O(f+g)$  равна доминанте  $O(f)$  и  $O(g)$  – оценка сложности суммы функций определяется как оценка сложности доминанты первого и второго слагаемых, например:

$$O(n^5+n^{10}) = O(n^{10})$$

# Асимптотический анализ – порядок роста

Порядок роста описывает то, как сложность алгоритма растет с увеличением размера входных данных. Порядок роста представляется в виде O-нотации:  $O(f(x))$ , где  $f(x)$  — формула, выражающая сложность алгоритма.

$O(1)$  – Константный

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

```
public int getSize(int[] arr) {  
    return arr.length;  
}
```



# Асимптотический анализ – порядок роста

$O(n)$  – линейный

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

Если линейный алгоритм обрабатывает один элемент 1 секунду, то сто элементов обработается за сто секунд.

```
public long getSum(int[] arr) {  
    long sum = 0;  
    for (int i = 0; i < arr.length; i++) {  
        sum += i; }  
    return sum;  
}
```

# *Асимптотический анализ – порядок роста*

$O(\log n)$  – логарифмический

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

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

Алгоритм двоичного поиска

# *Асимптотический анализ – порядок роста*

$O(n \log n)$  – линейно-логарифмический

Некоторые алгоритмы типа «разделяй и властвуй» попадают в эту категорию.  
Алгоритмы: Сортировка слиянием и быстрая сортировка

# Асимптотический анализ – порядок роста

## $O(n^2)$ – квадратичный

Время работы алгоритма  $O(n^2)$  зависит от квадрата размера входного массива. Квадратичная сложность — повод задуматься и переписать алгоритм.

Массив из 100 элементов потребует 1 0000 операций,

Массив из миллиона элементов потребует 1 000 000 000 000 (триллион) операций.

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

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

Алгоритм пузырьковая сортировка

# Асимптотический анализ – порядок роста

$O(n!)$  – факториальный

Очень медленный алгоритм.

<https://ru.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0%D0%BA%D0%BE%D0%BC%D0%BC%D0%B8%D0%B2%D0%BE%D1%8F%D0%B6%D1%91%D1%80%D0%B0>

# *Асимптотический анализ – Наилучший, средний и наихудший случаи*

Обычно имеется в виду наихудший случай, за исключением тех случаев, когда наихудший и средний сильно отличаются. (оценка сверху)

Например: `ArrayList.add()`

В среднем имеет порядок роста  $O(1)$ , но иногда может иметь  $O(n)$ .

В этом случае мы будем указывать, что алгоритм работает в среднем за константное время, и объяснять случаи, когда сложность возрастает.

Самое важное здесь то, что  $O(n)$  означает, что алгоритм потребует не более  $n$  шагов

# *Асимптотический анализ – Что мы в итоге измеряем и всегда ли это работает?*

- вычислительная сложность
- пространственная сложность

Алгоритм, который выполняется в десять раз быстрее, но использует в десять раз больше места, может вполне подходить для серверной машины с большим объемом памяти. Но на встроенных системах, где количество памяти ограничено, такой алгоритм использовать нельзя.

Асимптотический анализ не идеален, но это один из лучших доступных способов анализа алгоритмов.

- два алгоритма сортировки, которые занимают на машине  $1000 n \cdot \log(n)$  и  $2 n \cdot \log(n)$
- не можем судить, какой из них лучше, поскольку мы игнорируем константы
- можно в конечном итоге выбрать алгоритм, который асимптотически медленнее, но быстрее для вашего программного обеспечения.

# *Асимптотический анализ – основное*

1. Скорость алгоритма измеряется не в секундах, а в приросте количества операций.
2. Насколько быстро возрастает время работы алгоритма в зависимости от увеличения объема входящих данных.
3. Время работы алгоритма выражается при помощи нотации большого «O».
4. Алгоритм со скоростью  $O(\log n)$  быстрее, чем со скоростью  $O(n)$ , становится намного быстрее по мере увеличения списка элементов.



# *Понимание $time\ complexity$ на простых примерах*

Я кому то из вас дал ручку

У меня есть несколько способов найти мою ручку.

# *Понимание $time\ complexity$ на простых примерах*

Я кому то из вас дал ручку

У меня есть несколько способов найти мою ручку.

1. Я иду и спрашиваю у первого человека в классе, есть ли у него ручка. А дальше мне самому лениво идти к следующему и я просто начинаю спрашивать этого человека о других людях в классе, есть ли у них эта ручка? и так далее.

# *Понимание $time\ complexity$ на простых примерах*

Я кому то из вас дал ручку

У меня есть несколько способов найти мою ручку.

- ~~1. Я иду и спрашиваю у первого человека в классе, есть ли у него ручка. А дальше мне самому лениво идти к следующему и я просто начинаю спрашивать этого человека о других людях в классе, есть ли у них эта ручка? и так далее.~~
2. Я иду и спрашиваю каждого ученика по отдельности.

# *Понимание $time\ complexity$ на простых примерах*

Я кому то из вас дал ручку

У меня есть несколько способов найти мою ручку.

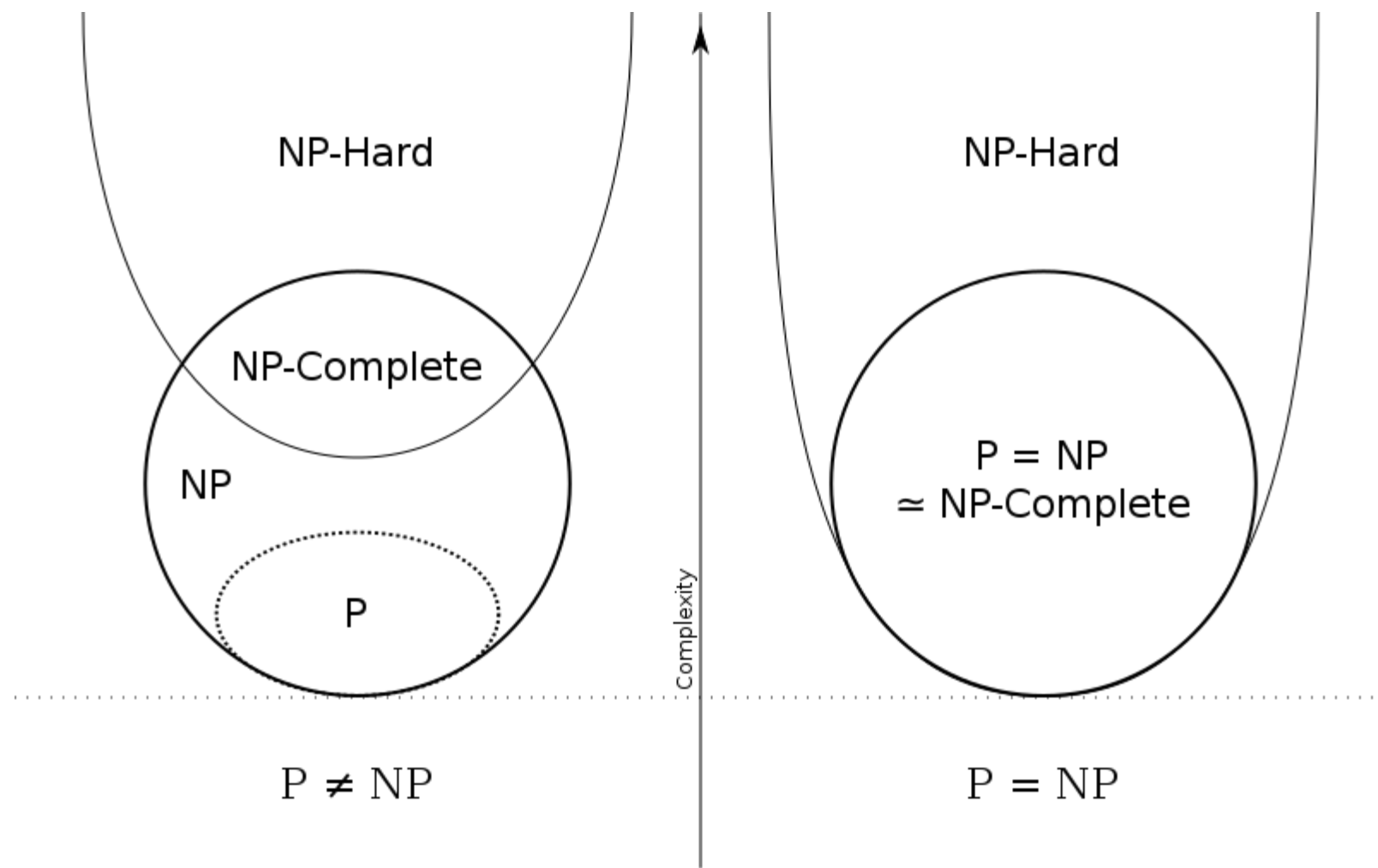
- ~~1. Я иду и спрашиваю у первого человека в классе, есть ли у него ручка. А дальше мне самому лениво идти к следующему и я просто начинаю спрашивать этого человека о других людях в классе, есть ли у них эта ручка? и так далее.~~
- ~~2. Я иду и спрашиваю каждого ученика по отдельности.~~
3. Я делю класс на две группы и спрашиваю: «Где моя ручка?» Затем я беру эту половину, делю ее на две части и спрашиваю снова, и так далее. Пока у меня не останется один ученик, у которого есть моя ручка.

# *Понимание $time\ complexity$ на простых примерах*

Я кому то из вас дал ручку

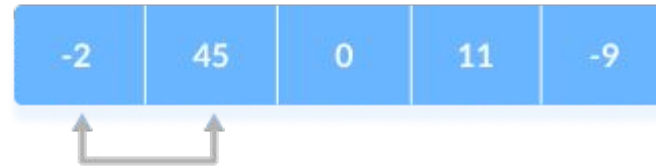
У меня есть несколько способов найти мою ручку.

- ~~1. Я иду и спрашиваю у первого человека в классе, есть ли у него ручка. А дальше мне самому лениво идти к следующему и я просто начинаю спрашивать этого человека о других людях в классе, есть ли у них эта ручка? и так далее.~~
- ~~2. Я иду и спрашиваю каждого ученика по отдельности.~~
- ~~3. Я делю класс на две группы и спрашиваю: «Где моя ручка?» Затем я беру эту половину, делю ее на две части и спрашиваю снова, и так далее. Пока у меня не останется один ученик, у которого есть моя ручка.~~
4. Точно, я вспомнил кому я ее дал.

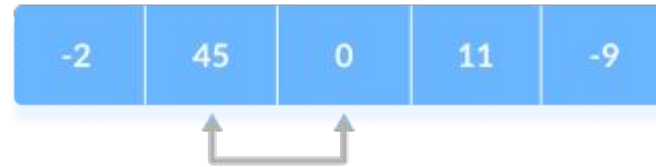


step = 0

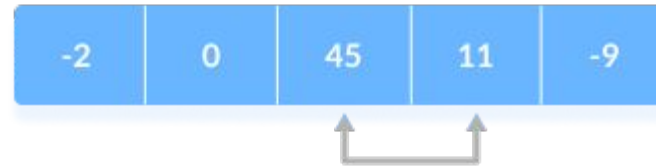
i = 0



i = 1



i = 2



i = 3

