



*School of  
Computer  
Science*

# **ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ АЛГОРИТМОВ**

## **ПРОГРАММИРОВАНИЕ НА PYTHON**

**Лекции для IT-школы**



# ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ

- Вычислительная сложность алгоритма это ***зависимость объема работы, выполняемой алгоритмом, от размера входных данных***
- Рассматривается в теории алгоритмов и изучения сложности вычислений, которая:
  - Исследует **скорость работы алгоритма**
  - Анализирует объемы потребляемых им ресурсов (памяти)



# ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ

## ВАРИАНТЫ ОЦЕНКИ СЛОЖНОСТИ

- Сложность алгоритмов в теории обозначается буквой  $O$  – например,  $O(n)$
- Сложность называется **временно́й**, если оценивается время работы алгоритма
- Можно также измерять **пространственную (объемную)** сложность, которая показывает сколько памяти/ресурса CPU занимает алгоритм
- Мы будем проводить эксперименты с временно́й сложностью



# ДЛЯ ЧЕГО ЭТО НУЖНО:

- Выбор лучших алгоритмов при разработке
- Обзоры/инспекции кода (Code Review),  
проверяющие:
  - следование правилам оформления кода
  - соответствие кода решаемой задаче
  - безопасность кода
  - оптимальное использование ресурсов
  - выбор оптимального алгоритма
- Оптимизация/рефакторинг кода
- Тестирование с выявлением «узких» мест:
  - автономное (модульное)
  - функциональное, интеграционное, регресс
  - нагрузочное (стресс) тестирование

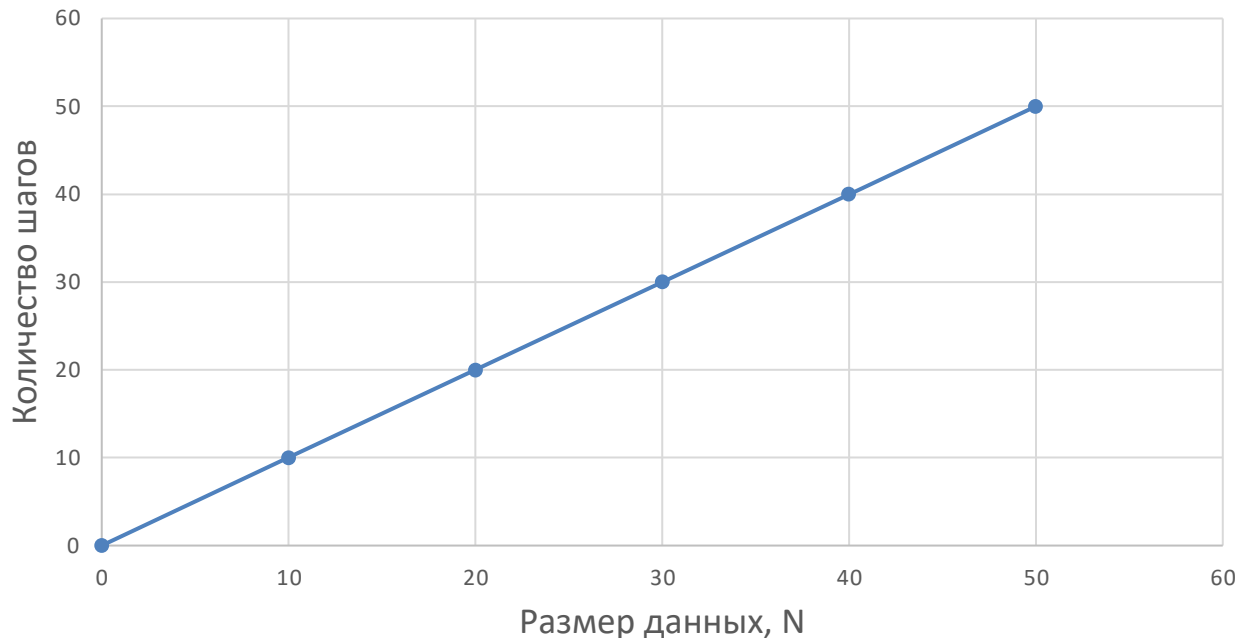


# ПРИМЕРЫ АЛГОРИТМОВ

## ЛИНЕЙНАЯ СЛОЖНОСТЬ

- `complexity_analysis.py / print_all()`
- График показывает количество шагов алгоритма в зависимости от  $N$ :

Количество напечатанных строк  
в зависимости от размера входных данных



Сложность алгоритма пропорциональна  $N$

В этом случае говорят, что алгоритм имеет линейную сложность

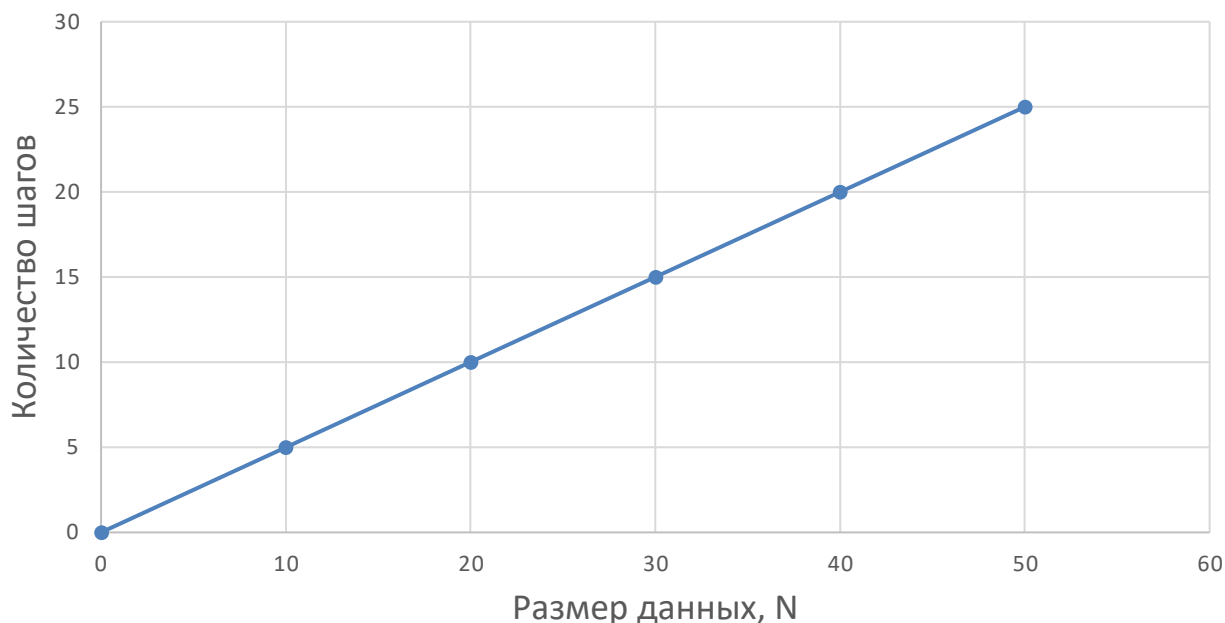


# ПРИМЕРЫ АЛГОРИТМОВ

## ЛИНЕЙНАЯ СЛОЖНОСТЬ

- `complexity_analysis.py / print_odd()`
- График показывает количество шагов алгоритма в зависимости от N:

Количество напечатанных строк  
в зависимости от размера входных данных



Количество шагов  
алгоритма равно  
 $N/2$

Сложность  
алгоритма также  
пропорциональна  
 $N$

Этот алгоритм  
также имеет  
линейную  
сложность

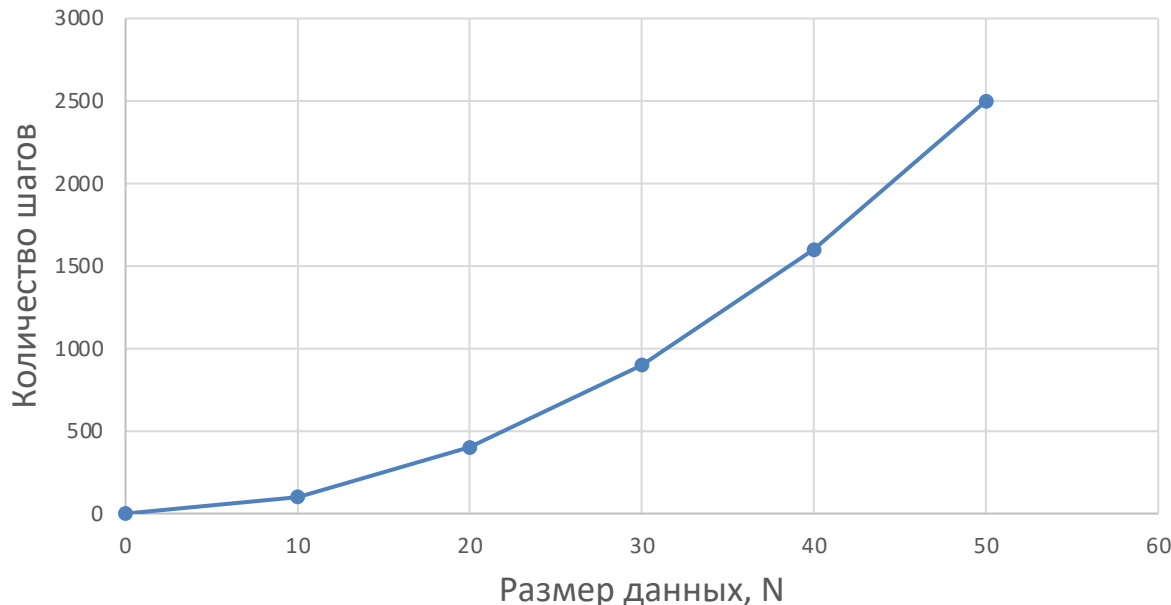


# ПРИМЕРЫ АЛГОРИТМОВ

## КВАДРАТИЧНАЯ СЛОЖНОСТЬ

- `complexity_analysis.py / print_pairs()`
- График показывает количество шагов алгоритма в зависимости от N:

Количество напечатанных строк  
в зависимости от размера входных данных



Количество шагов  
алгоритма равно  
 $N * N = N^2$

Сложность  
алгоритма также  
пропорциональна  
 $N^2$

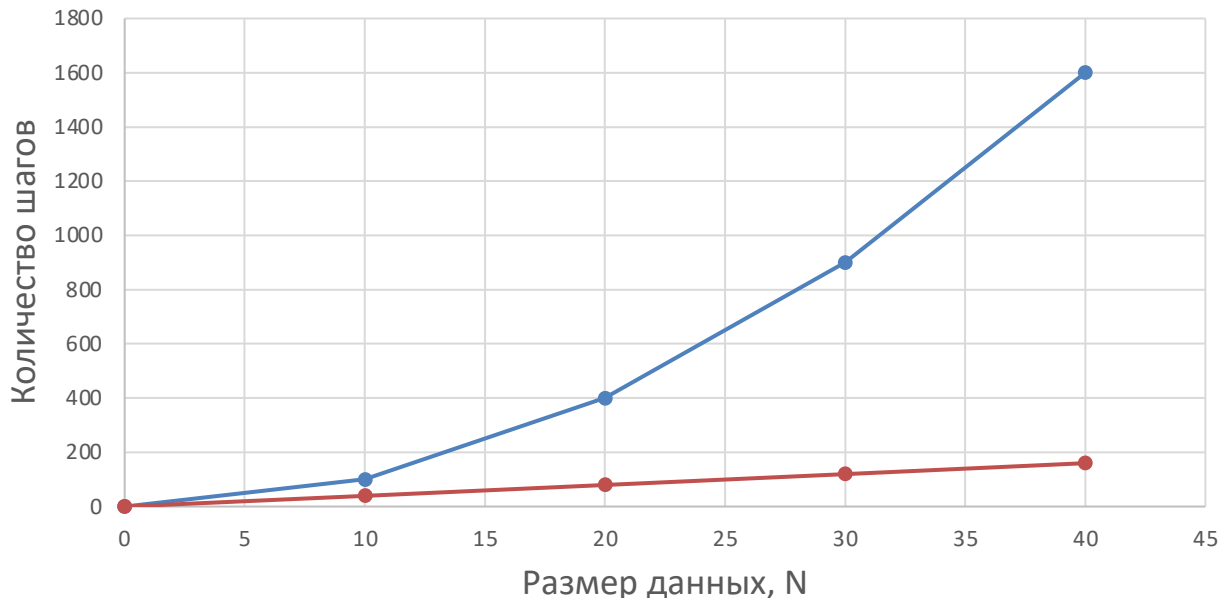
Этот алгоритм  
имеет  
квадратичную  
сложность



# СРАВНЕНИЕ АЛГОРИТМОВ ЛИНЕЙНОЙ И КВАДРАТИЧНОЙ СЛОЖНОСТИ

- Графики алгоритмов линейной и квадратичной сложности в зависимости от объемов данных (N):

Количество напечатанных строк  
в зависимости от размера входных данных



Если одну и ту же задачу можно решить линейным и квадратичным алгоритмом, то какой из них следует выбрать?





# ПРИМЕРЫ АЛГОРИТМОВ

## ЛОГАРИФМИЧЕСКАЯ СЛОЖНОСТЬ

- `complexity_analysis.py / print_dbl_steps()`
- График показывает количество шагов алгоритма в зависимости от N:



Количество шагов  
пропорционально  
 $\log_2(N)$

Сложность  
алгоритма также  
пропорциональна  
 $\log_2(N)$

Этот алгоритм  
имеет  
логарифмическую  
сложность



# ОПРЕДЕЛЕНИЕ ВРЕМЕНИ РАБОТЫ ПРОГРАММЫ

- Функция `time()` в модуле `time`
- Определите время начала работы алгоритма
- Дайте отработать алгоритму
- Определите разницу между текущим временем и временем начала работы
- С помощью скрипта `ops.py` найдите сколько операций в секунду выполняет ваш Python
- Какова вычислительная сложность алгоритма, приведенного в этом скрипте?



# ПРАВИЛА ОЦЕНКИ СЛОЖНОСТИ

## КАК ОПРЕДЕЛИТЬ ОБЩУЮ СЛОЖНОСТЬ?

- Оценивается сложность при стремлении размера входных данных к бесконечности
- Оценивается худший вариант работы алгоритма
- Используется только та часть алгоритма, которая дает наибольший прирост времени исполнения от объема данных:

Допустим, некоторому алгоритму нужно выполнить  $4n^3 + 7n$  условных операций, чтобы обработать  $n$  элементов входных данных. При увеличении  $n$  на итоговое время работы будет значительно больше влиять возведение  $n$  в куб, чем умножение его на  $4$  или же прибавление  $7n$ . Тогда говорят, что временная сложность этого алгоритма равна  $O(n^3)$ , т. е. зависит от размера входных данных кубически.

<https://tproger.ru/articles/computational-complexity-explained>



# ОПРЕДЕЛЕНИЕ СЛОЖНОСТИ №1

Рассмотрите следующий код:

```
def repeat_chars(st):  
    """ (str) -> str """  
    double = ''  
    for ch in st:  
        double = double + ch * 2  
    return double
```

Как растёт число шагов алгоритма при росте  $\text{len}(st)$ :

1. Число шагов постоянно и не зависит от длины строки
2. Растёт логарифмически в зависимости от  $\text{len}(st)$
3. Растёт линейно в зависимости от длины строки  $st$
4. Растёт квадратично в зависимости от  $\text{len}(st)$



# ОПРЕДЕЛЕНИЕ СЛОЖНОСТИ №2

Рассмотрите следующий код:

```
def first_thirty(s):  
    """ (str) -> str """  
    thirty = ''  
    for i in range(30):  
        thirty = thirty + s[i]  
    return thirty
```

Как растёт число шагов алгоритма при росте  $\text{len}(s)$ :

1. Число шагов постоянно и не зависит от  $\text{len}(s)$
2. Растёт логарифмически в зависимости от  $\text{len}(s)$
3. Растёт линейно в зависимости от длины строки  $s$
4. Растёт квадратично в зависимости от  $\text{len}(s)$





# ОПРЕДЕЛЕНИЕ СЛОЖНОСТИ №3

Рассмотрите следующий код:

```
def some_total(a_list):  
    """ (list of numbers) -> number """  
    total = 0  
    for item in a_list:  
        for i in range(10):  
            total = total + item * i  
    return total
```

Как растёт число шагов алгоритма в зависимости от длины списка:

1. Число шагов постоянно
2. Растёт логарифмически в зависимости от `len(a_list)`
3. Растёт линейно в зависимости от длины списка
4. Растёт квадратично в зависимости от `len(a_list)`



# ОПРЕДЕЛЕНИЕ СЛОЖНОСТИ №4

Рассмотрите следующий код:

```
>>> x = 5
>>>
>>> st = "some string"
>>>
>>> first_char = st[0]
>>>
>>>
if x > 10:
    print("A lot of!")
else:
    print("So few...")
```

So few...

Определите сложность для каждого оператора в этом коде.

Какова эта сложность?



# СРАВНЕНИЕ СЛОЖНОСТЕЙ

- Имеются такие сложности алгоритмов:
  - $O(N)$ ,  $O(N^2)$ ,  $O(N^3)$ ,  $O(\log N)$ ,  $O(1)$
- Алгоритм с какой сложностью будет:
  - Самый быстрый?
  - Самый медленный?
- Перечислите все эти сложности по возрастанию времени работы алгоритма, т.е. от самого быстрого алгоритма к самому медленному





# ВЫЗОВЫ ФУНКЦИЙ

## КАК ОПРЕДЕЛИТЬ ОБЩУЮ СЛОЖНОСТЬ?

- Если одна функция вызывает другую, то их сложности перемножаются
- Если функции вызываются по очереди, то их сложности складываются
- См. `func_complexity.py`
- Определите сложности алгоритмов для `multiply_complexity()` и `add_complexity()`



# ЛИНЕЙНЫЙ ПОИСК И ВРЕМЯ РАБОТЫ

## ДОКАЖИТЕ СЛОЖНОСТЬ

- См. [search\\_linear.py](#) – реализация линейного поиска
- Измерьте время работы с помощью `time`
- Как меняется время работы алгоритма в зависимости от объемов данных?
- Докажите линейную сложность алгоритма как теоретически, так и практически



# БИНАРНЫЙ ПОИСК

## ДОКАЖИТЕ СЛОЖНОСТЬ

- Для использования бинарного поиска массив данных должен быть отсортирован
- См. [search\\_binary.py](#) – реализация бинарного поиска
- С каждой итерацией половина элементов массива отбрасывается
- Докажите теоретически, что сложность этого алгоритма логарифмическая



# СРАВНЕНИЕ АЛГОРИТМОВ ПОИСКА

## ЛИНЕЙНЫЙ ПОИСК

Размер данных	Количество шагов
1	1
2	2
3	3
4	4
...	...
N	N



# СРАВНЕНИЕ АЛГОРИТМОВ ПОИСКА

## БИНАРНЫЙ ПОИСК

Размер данных	Количество шагов
2	1
4	2
8	3
16	4
32	5
...	...
N	$\text{Log}_2(N)$



# СРАВНЕНИЕ АЛГОРИТМОВ ПОИСКА

## ПРОФИЛИРОВАНИЕ

- Профилирование – тестовый прогон программы со сбором статистики
- Профилировать – исполнять код для измерения потребления памяти и скорости
- В Python имеется встроенный модуль `cProfile`:
  - `import cProfile`
  - `help(cProfile.run)`
- Профилируем алгоритмы линейного и бинарного поиска
- Находим вариант оптимизации для алгоритма линейного поиска



# СЛОЖНОСТЬ ОПЕРАЦИЙ СО СПИСКОМ

- Список `list` реализован в Python как массив переменной длины
- Алгоритмические сложности основных операций со списком:
  - Вставка в произвольную позицию –  $O(N)$
  - **Добавление в конец списка –  $O(1)$**
  - **Получение элемента –  $O(1)$**
  - **Удаление с хвоста, `pop()` –  $O(1)$**
  - Удаление любого элемента –  $O(N)$
  - Проход по списку –  $O(N)$
  - Получение длины –  $O(1)$
- Результаты измерений см.  
<http://aliev.me/runestone/AlgorithmAnalysis/Lists.html>



# СЛОЖНОСТЬ ОПЕРАЦИЙ СО МНОЖЕСТВОМ

- Алгоритмические сложности основных операций со множеством:
  - Проверить наличие элемента/добавить элемент в множество –  $O(1)$
  - Отличие множества A от B –  $O(\text{длина A})$
  - Пересечение множеств A и B –  $O(\text{минимальная длина A или B})$
  - Объединение множеств A и B –  $O(N)$ , где N – это длина A + длина B
  - Получение длины –  $O(1)$
- См. <https://tproger.ru/translations/data-structure-time-complexity-in-python>





# СЛОЖНОСТЬ ОПЕРАЦИЙ СО СЛОВАРЕМ

- В словаре ключ используется для получения, установки и удаления элемента
- Алгоритмические сложности основных операций со словарем:
  - Получение элемента –  $O(1)$
  - Установка элемента –  $O(1)$
  - Удаление элемента –  $O(1)$
  - Проход по словарю –  $O(N)$
  - Получение длины –  $O(1)$
- См. <https://tproger.ru/translations/data-structure-time-complexity-in-python>

**СПАСИБО ЗА ВНИМАНИЕ !**  
**ВОПРОСЫ ?**



*School of  
Computer  
Science*