САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3

по курсу «Алгоритмы и структуры данных»

Тема: БЫСТРАЯ СОРТИРОВКА, СОРТИРОВКИ ЗА ЛИНЕЙНОЕ ВРЕМЯ.

Вариант 21

Выполнил:

Ступичев М. Н.

К3139

Проверил:

…

Санкт-Петербург

2024 г.

# Содержание отчета

[Содержание отчета 2](#_Toc184922714)

[Задачи по варианту 3](#_Toc184922715)

[Задача №1. Быстрая сортировка. 3](#_Toc184922716)

[Задача №3. Сортировка пугалом. 5](#_Toc184922717)

[Задача №4. Умножение многочленов. 6](#_Toc184922718)

[Вывод 8](#_Toc184922719)

# Задачи по варианту

## Задача №1. Быстрая сортировка.

Реализовать алгоритм быстрой сортировки.

Решение:

def partition(array: List, start: int, end: int) -> Tuple[int, int]:

   pivot = array[randint(start, end - 1)]

   end\_less\_part = end\_equal\_part = start

   for i in range(start, end):

       if array[i] < pivot:

           array[i], array[end\_equal\_part], array[end\_less\_part] = array[end\_equal\_part], array[end\_less\_part], array[i]

           end\_less\_part += 1

           end\_equal\_part += 1

       elif array[i] == pivot:

           array[end\_equal\_part], array[i] = array[i], array[end\_equal\_part]

           end\_equal\_part += 1

​

   return end\_less\_part, end\_equal\_part

​

​

def quick\_sort(array: List, start: int = 0, end: int = -1) -> None:

   if end == -1:

       end = len(array)

​

   if end - start <= 1:

       return

​

   end\_less\_part, end\_equal\_part = partition(array, start, end)

   quick\_sort(array, start, end\_less\_part)

   quick\_sort(array, end\_equal\_part, end)

Текстовое объяснение решения.

Этот код реализует быструю сортировку (QuickSort) с учетом повторяющихся элементов. Разберем функции по отдельности:

**1. partition(array, start, end):**

Эта функция разделяет подмассив array[start:end] на три части:

* Элементы меньше, чем выбранный pivot. Они располагаются в array[start:end\_less\_part].
* Элементы равные pivot. Они располагаются в array[end\_less\_part:end\_equal\_part].
* Элементы больше, чем pivot. Они располагаются в array[end\_equal\_part:end].

Выбор pivot происходит случайным образом (randint(start, end - 1)), что помогает избежать худшего случая O(n²) при уже отсортированном или почти отсортированном массиве.

Алгоритм перестановки элементов нетривиален и эффективен. Он использует три указателя:

* end\_less\_part: указывает на конец части элементов меньших pivot.
* end\_equal\_part: указывает на конец части элементов равных pivot.
* Цикл итеративно проходит по массиву. Если элемент меньше pivot, он переставляется в начало end\_less\_part части, end\_less\_part и end\_equal\_part инкрементируются. Если элемент равен pivot, он переставляется в end\_equal\_part часть, и end\_equal\_part инкрементируется.

**Асимптотика partition:** Функция итерируется по подмассиву длиной end - start. Поэтому её асимптотическая сложность - **O(n)**, где n - размер подмассива.

**2. quick\_sort(array, start, end):**

Это рекурсивная функция, реализующая быструю сортировку.

* Базовый случай: если длина подмассива end - start <= 1, сортировка не нужна, функция возвращается.
* Рекурсивный шаг:
  + Вызывается partition для разделения подмассива.
  + Рекурсивно вызывается quick\_sort для левой части (меньше pivot) и правой части (больше pivot).

**Асимптотика quick\_sort:**

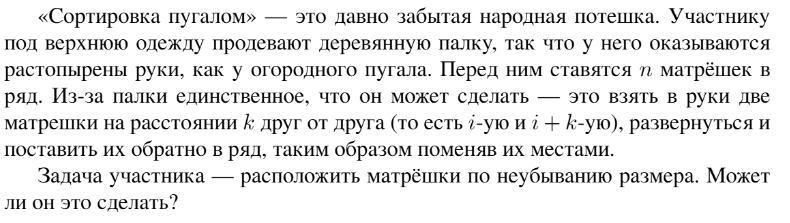
* **Лучший и средний случай:** O(n log n). Каждый вызов partition делит массив примерно пополам. Количество уровней рекурсии - log₂n. На каждом уровне выполняется O(n) операций.
* **Худший случай:** O(n²). Это происходит, если pivot всегда выбирается крайним элементом (минимумом или максимумом), и массив делится на подмассивы размера 1 и n-1 на каждом шаге. Это маловероятно при случайном выборе pivot, как в данном коде.

**В целом:** Представленный код является эффективной реализацией быстрой сортировки с обработкой дубликатов. Средняя и лучшая временная сложность - **O(n log n)**, худшая - **O(n²)**, но вероятность худшего случая низка благодаря случайному выбору pivot. Пространственная сложность - **O(log n)** в среднем и лучшем случае (из-за рекурсии), и **O(n)** в худшем случае (также из-за рекурсии, в случае сильно несбалансированного разбиения).

Вывод по задаче:

Реализован стандартный алгоритм быстрой сортировки.

## Задача №3. Сортировка пугалом.



Решение:

def fast\_scarecrow\_sort\_check(array: List[int], k: int) -> bool:

   sorted\_array = array[:]

   quick\_sort(sorted\_array)

​

   number\_indexes = dict()

   for index, value in enumerate(array):

       if number\_indexes.get(value) is None:

           number\_indexes[value] = []

​

       number\_indexes[value].append([index, False])

​

​

   for index, value in enumerate(sorted\_array):

       for cur\_index in number\_indexes[value]:

           if not cur\_index[1] and (cur\_index[0] - index) % k == 0:

               cur\_index[1] = True

               break

       else:

           return False

​

   return True​​

​

Текстовое объяснение решения.

Функция fast\_scarecrow\_sort\_check проверяет, можно ли расположить элементы массива array в отсортированном порядке, перемещая каждый элемент на кратное k расстояние.

Разберем пошагово:

1. **sorted\_array = array[:]**: Создается копия входного массива array. Это важно, чтобы не модифицировать исходный массив.
2. **quick\_sort(sorted\_array)**: Копия массива сортируется с помощью функции quick\_sort (описанной в предыдущем примере). Это дает отсортированный вариант исходного массива.
3. **number\_indexes = dict()**: Создается словарь, где ключи - значения из исходного массива, а значения - списки. Каждый список содержит пары [index, False], где index - индекс элемента в исходном массиве, а False - флаг, указывающий, был ли элемент уже “размещен” в отсортированном порядке.
4. **Цикл for index, value in enumerate(array)**: Этот цикл заполняет словарь number\_indexes. Для каждого элемента исходного массива сохраняется его индекс и флаг False.
5. **Цикл for index, value in enumerate(sorted\_array)**: Этот цикл итерируется по отсортированному массиву. Для каждого элемента он ищет все вхождения этого элемента в исходном массиве (через словарь number\_indexes).
6. **Внутренний цикл for cur\_index in number\_indexes[value]**: Проверяется каждый индекс cur\_index[0] элемента value в исходном массиве. Условие (cur\_index[0] - index) % k == 0 проверяет, можно ли переместить элемент из cur\_index[0] в позицию index (в отсортированном массиве), при этом расстояние должно быть кратно k.
7. **cur\_index[1] = True**: Если условие выполняется, флаг устанавливается в True, и внутренний цикл прерывается (break). Это означает, что элемент успешно размещен.
8. **else:**: Если внутренний цикл завершился без break, значит, для текущего элемента из отсортированного массива не найдено подходящего места в исходном массиве на расстоянии, кратном k. Функция возвращает False.
9. **return True**: Если все элементы из отсортированного массива успешно размещены, функция возвращает True.

**Асимптотическая сложность:**

* **quick\_sort:** O(n log n) в среднем, O(n²) в худшем случае.
* **Создание словаря number\_indexes:** O(n)
* **Два вложенных цикла:** В худшем случае O(n²), если все элементы в массиве одинаковые. В среднем случае – ближе к O(n), так как в каждом внутреннем цикле в среднем обрабатывается константное количество элементов.

Таким образом, **общая асимптотическая сложность – O(n log n)** в среднем случае, **O(n²)** в худшем случае (если все элементы одинаковые и k=1). Пространственная сложность - O(n) из-за словаря и копии массива.

Вывод по задаче:

Реализован алгоритм, проверяющий можно ли отсортировать массив сортировкой пугалом.

## Задача №4. Умножение многочленов.

Реализовать алгоритм, которых находит для заданных точек количество отрезков, которым они принадлежат.

Решение:

def points\_in\_section(sections: List[Tuple[int, int]], points: List[int]):

   main\_array = []

​

   for section in sections:

       main\_array.append((section[0], 0))

       main\_array.append((section[1], 2))

​

   for point in points:

       main\_array.append((point, 1))

​

   quick\_sort(main\_array)

​

   cur = 0

   ans = []

   for point in main\_array:

       if point[1] == 0:

           cur += 1

       elif point[1] == 1:

           ans.append(cur)

       elif point[1] == 2:

           cur -= 1

   return ans

​​​

Текстовое объяснение решения.

Функция points\_in\_section подсчитывает количество активных секций для каждой точки. Она использует сортировку и подход, похожий на подсчет событий.

Разберем пошагово:

1. **main\_array = []**: Создается пустой массив, который будет содержать тройки (значение, тип события).
2. **Цикл for section in sections**: Для каждой секции (start, end) добавляются две записи в main\_array:
   * (start, 0): начало секции (событие типа 0).
   * (end, 2): конец секции (событие типа 2).
3. **Цикл for point in points**: Для каждой точки добавляется запись (point, 1) в main\_array (событие типа 1).
4. **quick\_sort(main\_array)**: Массив main\_array сортируется по первому элементу (значению) с помощью быстрой сортировки. Это гарантирует, что события обрабатываются в порядке возрастания значений.
5. **cur = 0**: Переменная cur отслеживает количество активных секций.
6. **Цикл for point in main\_array**: Цикл итерируется по отсортированному массиву.
   * Если point[1] == 0: Началась секция, cur увеличивается.
   * Если point[1] == 1: Встречена точка. Текущее значение cur (количество активных секций) добавляется в массив ans.
   * Если point[1] == 2: Закончилась секция, cur уменьшается.
7. **return ans**: Возвращается массив ans, содержащий количество активных секций для каждой точки.

**Пример:**

sections = [(1, 5), (3, 7)] points = [2, 4, 6]

main\_array после первого цикла: [(1, 0), (5, 2), (3, 0), (7, 2)] main\_array после второго цикла: [(1, 0), (5, 2), (3, 0), (7, 2), (2, 1), (4, 1), (6, 1)] main\_array после сортировки: [(1, 0), (2, 1), (3, 0), (4, 1), (5, 2), (6, 1), (7, 2)]

Итерация по отсортированному main\_array:

* (1, 0): cur = 1
* (2, 1): ans.append(1)
* (3, 0): cur = 2
* (4, 1): ans.append(2)
* (5, 2): cur = 1
* (6, 1): ans.append(1)
* (7, 2): cur = 0

Результат: ans = [1, 2, 1]

**Асимптотическая сложность:**

* Создание main\_array: O(n + m), где n - количество секций, m - количество точек.
* Сортировка quick\_sort: O((n + m) log(n + m)) в среднем, O((n + m)²) в худшем случае.
* Итерация по main\_array: O(n + m)

Таким образом, общая асимптотическая сложность - **O((n + m) log(n + m))** в среднем случае и **O((n + m)²)** в худшем случае. Пространственная сложность - O(n + m).

Вывод по задаче:

Алгоритм работает эффективно даже при больших значениях n и **m**, делая его подходящим для задачи нахождения количества отрезков, включающих каждую из точек.

# Вывод

Решены 3 задачи из 21 варианта.