Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

(Университет ИТМО)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №3

по курсу «Алгоритмы и структуры данных»

Тема: БЫСТРАЯ СОРТИРОВКА, СОРТИРОВКИ ЗА ЛИНЕЙНОЕ ВРЕМЯ

Вариант 21

СОДЕРЖАНИЕ

[ЗАДАЧИ ПО ВАРИАНТУ 4](#_Toc181018147)

[1 Сортировка слиянием 4](#_Toc181018148)

[Задание 4](#_Toc181018149)

[Код 4](#_Toc181018150)

[Анализ кода 5](#_Toc181018151)

[Вывод 5](#_Toc181018152)

[2 Нахождение количества инверсий 6](#_Toc181018153)

[Задание 6](#_Toc181018154)

[Код 6](#_Toc181018155)

[Анализ кода 7](#_Toc181018156)

[Вывод 7](#_Toc181018157)

[3 Бинарный поиск 8](#_Toc181018158)

[Задание 8](#_Toc181018159)

[Код 8](#_Toc181018160)

[Анализ кода 8](#_Toc181018161)

[Вывод 9](#_Toc181018162)

[4 Поиск мажорирующего элемента 10](#_Toc181018163)

[Задание 10](#_Toc181018164)

[Код 1 10](#_Toc181018165)

[Анализ кода 1 10](#_Toc181018166)

[Код 2 11](#_Toc181018167)

[Анализ кода 2 11](#_Toc181018168)

[Вывод 11](#_Toc181018169)

[5 Поиск подмассива с максимальной суммой за линию 12](#_Toc181018170)

[Задание 12](#_Toc181018171)

[Код 12](#_Toc181018172)

[Анализ кода 12](#_Toc181018173)

[Вывод 13](#_Toc181018174)

[6 Умножение полиномов 14](#_Toc181018175)

[Задание 14](#_Toc181018176)

[Код 14](#_Toc181018177)

[Анализ кода 15](#_Toc181018178)

[Вывод 16](#_Toc181018179)

[ВЫВОД ПО ЛАБОРАТОРНОЙ РАБОТЕ 17](#_Toc181018180)

# ЗАДАЧИ ПО ВАРИАНТУ

## Быстрая сортировка

### Задание

Реализовать алгоритм быстрой сортировки

### Код

from typing import List, Tuple, TypeVar  
from random import randint  
  
  
T = TypeVar("T")  
  
  
def partition(lst: List[T], start: int, end: int, pivot: T) -> Tuple[int, int]:  
 end\_left = end\_mid = start  
 for i in range(start, end):  
 if lst[i] < pivot:  
 lst[i], lst[end\_mid], lst[end\_left] = lst[end\_mid], lst[end\_left], lst[i]  
 end\_left += 1  
 end\_mid += 1  
 elif lst[i] == pivot:  
 lst[end\_mid], lst[i] = lst[i], lst[end\_mid]  
 end\_mid += 1  
  
 return end\_left, end\_mid  
  
  
def quick\_sort(lst: List[T], start: int = 0, end: int = -1) -> None:  
 if end == -1:  
 end = len(lst)  
  
 if end - start < 2:  
 return  
  
 end\_left, end\_mid = partition(lst, start, end, lst[randint(start, end - 1)])  
 quick\_sort(lst, start, end\_left)  
 quick\_sort(lst, end\_mid, end)

### Анализ кода

1. **Функция** partition

Принимает подмассив от start до end, переменную pivot и разделяет массив на три части:

* Элементы, меньшие pivot (слева).
* Элементы, равные pivot (посередине).
* Элементы, большие pivot (в оставшейся части).

Проходит по массиву один раз, используя указатели end\_left и end\_mid для отслеживания текущих позиций, чтобы обменивать значения и сохранять относительный порядок элементов.

1. **Функция** quick\_sort**:**

Рекурсивно сортирует подмассивы. Вызов с диапазоном [start, end\_left) сортирует элементы, меньшие pivot. Вызов с [end\_mid, end) сортирует элементы, большие pivot.

Завершает работу, если в подмассиве менее двух элементов.

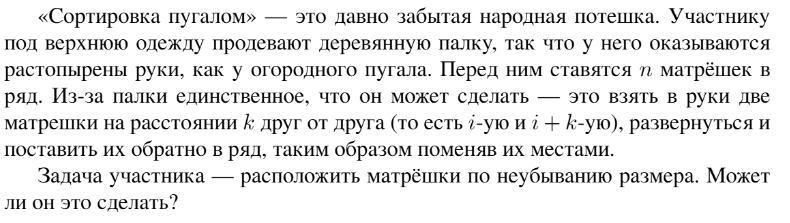
Время работы в среднем случае: O(n\*log ₂n), где n - длина списка. Время работы в худшем случае: O(n^2)

### Вывод

Хотя время работы алгоритма для массива из nn элементов в худшем случае может составить Θ(n2), на практике этот алгоритм является одним из самых быстрых.

## Сортировка пугалом

### Задание



### Код

from typing import List  
from Lab3.Task1.src.QuickSort import quick\_sort  
  
  
def scarecrow\_sort\_checking\_indexes(lst: List[int], delta: int) -> bool:  
 sorted\_lst = lst[:]  
 quick\_sort(sorted\_lst)  
  
 value\_indexes = dict()  
 for index, value in enumerate(lst):  
 if value\_indexes.get(value) is None:  
 value\_indexes[value] = []  
  
 value\_indexes[value].append(dict(index=index, is\_taken=False))  
  
  
 for index, value in enumerate(sorted\_lst):  
 for element in value\_indexes[value]:  
 if not element["is\_taken"] and abs(element["index"] - index) % delta == 0:  
 element["is\_taken"] = True  
 break  
 else: return False  
  
 return True  
  
  
def scarecrow\_sort\_real\_sorting(lst: List[int], delta: int) -> bool:  
 sorted\_lst = lst[:]  
 quick\_sort(sorted\_lst)  
  
 for i in range(delta, len(lst)):  
 cur = i  
 while cur > 0 and lst[cur - delta] > lst[cur]:  
 lst[cur - delta], lst[cur] = lst[cur], lst[cur - delta]  
 cur -= 1  
  
 return sorted\_lst == lst

### Анализ кода

Задача была решена двумя способами

1. scarecrow\_sort\_checking\_indexes

Алгоритм основан на том факте, что исходный список можно отсортировать пугалом тогда и только тогда, когда каждый его элемент может быть поставлен на свое место в отсортированном массиве, т.е. индекс элемента в исходном массиве и индекс этого элемента в отсортированном массиве должны отличаться на значение, кратное delta. Это и проверяет алгоритм.

Для каждого элемента в отсортированном списке алгоритм ищет среди всех индексов (для эффективности используется словарь, который заранее заполняется) элементов с таким же значением из исходного списка (тот, что еще не взят в пару к предыдущим элементам) такой, что его значение отличается от значения текущего индекса на число, кратное delta. Если хотя бы один такой индекс найден (т.е. найден элемент с таким же значением и подходящим индексом), он помечается, как взятый в пару с текущим элементом отсортированного списка, если нет – сортировка пугалом невозможна. Если для каждого элемента отсортированного списка найден подходящий элемент из исходного, функция возвращает истинный результат.

1. scarecrow\_sort\_real\_sorting

Этот алгоритм просто сортирует исходный список пугалом: сортировка основана на сортировке вставкой. В конце функция проверяет, отсортировался ли массив.

**Оценка времени работы**:

Время работы scarecrow\_sort\_checking\_indexes зависит от количества одинаковых элементов в списке. Если все элементы разные, то алгоритм работает за линию. Если все элементы одинаковые – за O(n^2 / delta)

Время работы scarecrow\_sort\_real\_sorting складывается из O(n\*log₂n) на сортировку списка для последующей проверки, и O(n^2 / delta) на сортировку вставкой. Т.е. выходит O(n^2).

Затраты памяти: O(n) в обоих случаях.

## Точки в отрезках

### Задание

Реализовать алгоритм, которых находит для заданных точек количество отрезков (заданных), которым они пренадлежат.

### Код

from collections.abc import Iterator  
from typing import List, Tuple  
from Lab3.Task1.src.QuickSort import quick\_sort  
  
  
# флаги  
START = 0  
POINT = 1  
END = 2  
  
  
# Алгоритм выводит ответы соответственно для точек, отсортированных в возрастающем порядке  
def sections\_n\_points(sections: List[Tuple[int, int]], points: List[int]) -> Iterator[int]:  
 all\_points = []  
  
 for section in sections:  
 all\_points.append((section[0], START))  
 all\_points.append((section[1], END))  
  
 for point in points:  
 all\_points.append((point, POINT))  
  
 quick\_sort(all\_points)  
 active = 0  
 for point in all\_points:  
 if point[1] == START:  
 active += 1  
 elif point[1] == POINT:  
 yield active  
 else: active -= 1

### Анализ кода

sections – список отрезков, каждый представлен как кортеж (начало, конец). points – список точек, для которых нужно посчитать количество пересечений.

Каждое событие (начало или конец отрезка, либо точка) добавляется в all\_points как кортеж (координата, флаг), где флаг принимает значения START, END, POINT для обозначения начала и конца отрезка или точки соответственно.

Весь список all\_points сортируется по координатам, используя быструю сортировку. Сортировка гарантирует, что события будут обработаны слева направо по координатной оси. Важен приоритет флагов: если начало какого-то отрезка совпадает со значением какой-то точки, то точка начала этого отрезка должна обработаться до обработки этой точки (чтобы этот отрезок был активен до обработки точки), аналогично концы отрезков надо обрабатывать после обработки точки, если их значения совпадают. Поэтому были выбраны именно такие значения START, END и POINT. При сортировке он встанут в нужном порядке.

Используется счетчик active для поддержания количества активных отрезков на каждом этапе. При встрече события START счетчик увеличивается, а при END – уменьшается. При встрече события POINT в yield выводится текущее значение active, которое является ответом на количество отрезков, охватывающих точку в этой позиции.

**Оценка времени работы**:

Время работы: O((n + k)\*log ₂(n + k))на сортировку и O(n + k) на основную часть алгоритма, где n – количество отрезков, k – количество точек.

Затраты памяти: O(n + k) на переменные.

### Вывод

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

## Индекс Хирша

### Задание

Реализовать алгоритм, находящий индекс Хирша для списка.

### Код

from typing import List  
from Lab3.Task1.src.QuickSort import quick\_sort  
  
  
def hirsch(lst: List[int]):  
 quick\_sort(lst)  
  
 for i in range(len(lst)):  
 if len(lst) - i <= lst[i] != lst[i - 1]:  
 return len(lst) - i  
  
 return 0

### Анализ кода

Индекс Хирша (h-индекс) – это показатель продуктивности и значимости исследователя, основанный на количестве его публикаций и их цитировании. Список имеет индекс h, если h элементов имеют значение больше или равное h, а остальные имеют значение меньше h.

В отсортированном списке значение n – i для i-ого элемента показывает количество элементов, значение которых больше или равно i-ого элемента. Найдем минимальный элемент (пусть его индекс равен k), значение которого больше n – k. Для этого элемента будет выполняться: n – k элементов (те, что стоят после) больше этого элемента, а значит и больше n – k; (k-1)-ый элемент < n – k + 1, т.е. (k-1)-ый элемент <= n – k, тогда любой элемент с индексом меньше k <= n – k. Соответственно, это и есть индекс Хирша (с учетом, что (k-1)-ый строго меньше k-ого).

**Оценка времени работы**:

Время работы: O(n), где n - это количество элементов в массиве.

Затраты памяти: O(1).

## Сортировка подсчетом

### Задание

Реализовать поразрядную сортировку при помощи сортировки подсчетом, вывести состояния данных при выполнении сортировки.

### Код

from typing import List  
from Utils.Read\_n\_Write import \*  
  
  
def counting\_sort(lst: List[str], indexes\_lst: List[int], index: int):  
 lst\_copy = lst[:]  
 indexes\_lst\_copy = indexes\_lst[:]  
 count = [0] \* (ord('z') - ord('a') + 1)  
  
 for word in lst:  
 count[ord(word[index]) - ord('a')] += 1  
  
 count[0] -= 1  
 for i in range(1, len(count)):  
 count[i] += count[i - 1]  
  
 for i in range(len(lst\_copy) - 1, -1, -1):  
 sym\_index = ord(lst\_copy[i][index]) - ord('a')  
 lst[count[sym\_index]] = lst\_copy[i]  
 indexes\_lst[count[sym\_index]] = indexes\_lst\_copy[i]  
 count[sym\_index] -= 1  
  
  
def radix\_sort(lst: List[str], lens: int, number: int):  
 indexes\_lst = list(range(1, len(lst) + 1))  
  
  
 for ind in range(lens - 1, lens - number - 1, -1):  
 counting\_sort(lst, indexes\_lst, ind)  
 return indexes\_lst

### Анализ кода

Сортировка подсчетом работает следующим образом:

Исходная последовательность чисел длины n, а в конце отсортированная, хранится в массиве lst. Также используется вспомогательный массив count с индексами от 0 до k−1, изначально заполняемый нулями. Последовательно пройдём по массиву lst и запишем в count[i] количество чисел, равных i. Теперь достаточно пройти по массиву count и для каждого элемента el в список lst последовательно записать el count[el] раз.

Функция counting\_sort, основываясь на сортировке подсчетом, создает список count (подсчет идет по индексу index в каждой строке) и абсолютно так же заполняет его. После список count переделывается в список, где каждый элемент обозначает индекс, куда надо поставить соответствующий элемент при встрече. Далее алгоритм проходится по исходному списку и для каждого элемента смотрит, в какой индекс в отсортированном списке его нужно поставить (в соответствии со значением в count) и изменяет соответствующий индекс в count. При этом параллельно заполняется информация о индексах в новом списке.

**Оценка времени работы**:

Время работы: O(n \* k), где n – это количество элементов в массиве, k – количество итераций сортировки подсчетом. Т.к. все происходит за один проход по списку.

Затраты памяти: O(n) на переменные.

### Вывод

Сортировка подсчётом – алгоритм сортировки, в котором используется диапазон чисел сортируемого массива (списка) для подсчёта совпадающих элементов. Применение сортировки подсчётом целесообразно лишь тогда, когда сортируемые числа имеют (или их можно отобразить в) диапазон возможных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел меньших 1000.

## K ближайших точек

### Задание

Найти k ближайших точек к началу координат.

### Код

from Utils.Read\_n\_Write import \*  
from typing import List, Tuple  
from random import randint  
  
  
def get\_norma(point: Tuple[int, int]):  
 return point[0] \*\* 2 + point[1] \*\* 2

def partition\_by\_norma(lst: List[Tuple[int, int]], start: int, end: int, pivot: Tuple[int, int]) -> Tuple[int, int]:  
 end\_left = end\_mid = start  
 pivot\_norma = get\_norma(pivot)  
 for i in range(start, end):  
 cur\_norma = get\_norma(lst[i])  
 if cur\_norma < pivot\_norma:  
 lst[i], lst[end\_mid], lst[end\_left] = lst[end\_mid], lst[end\_left], lst[i]  
 end\_left += 1  
 end\_mid += 1  
 elif cur\_norma == pivot\_norma:  
 lst[end\_mid], lst[i] = lst[i], lst[end\_mid]  
 end\_mid += 1  
  
 return end\_left, end\_mid

def nearest\_points\_as\_quick\_sort(lst: List[Tuple[int, int]], number: int, output: List[Tuple[int, int]],  
 start: int = 0, end: int = -1):  
 if end == -1:  
 end = len(lst)  
  
 if end - start < 1:  
 return  
  
 end\_left, end\_mid = partition\_by\_norma(lst, start, end, lst[randint(start, end - 1)])  
  
 if end\_left <= number:  
 for i in range(end\_left):  
 output.append(lst[i])  
  
 if end\_mid <= number:  
 for i in range(end\_left, end\_mid):  
 output.append(lst[i])  
 nearest\_points\_as\_quick\_sort(lst, number - end\_mid, output, end\_mid, end)  
 else:  
 for i in range(end\_left, number):  
 output.append(lst[i])  
 return  
 else:  
 nearest\_points\_as\_quick\_sort(lst, number, output, start, end\_left)

for i in range(len(sum\_f\_sum\_g)):  
 result[mid + i] += sum\_f\_sum\_g[i]  
  
 return result  
  
 return mult\_polynomials\_recursion(f, g, 0, len(f), 0, len(g))

### Анализ кода

Алгоритм проверяет, сколько точек находится в подмассивах, разделенных partition\_by\_norma. Если количество точек в левом подмассиве (end\_left) меньше number, все они добавляются в output. Далее проверяется количество точек в среднем подмассиве (end\_mid). Если количество точек из end\_left до end\_mid также не превосходит number, все они добавляются в output, а функция рекурсивно вызывается для правого подмассива (чтобы найти оставшиеся ближайшие точки). Если ни один из случаев не выполняется, ближайшие точки находятся исключительно в левом подмассиве, и рекурсия продолжается только для этого подмассива.

**Оценка времени работы**:

Время работы: O(n) в среднем, где n - длина списка.

Память: O(n)

### Вывод

Этот код реализует алгоритм поиска ближайших к началу координат точек, используя принцип быстрой сортировки (Quick Sort) с кастомной функцией разбиения по расстоянию до начала координат (partition\_by\_norma).