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 6

Выполнил:
Данилова А.В.
К3141

Проверила:
Артамонова В.Е.

Санкт-Петербург
2024 г.

Содержание отчета

Содержание отчета **2**

Задачи по варианту **3**

Задача №1. Улучшение Quick sort

Задача №5. Индекс Хирша

Задача №8. K ближайших точек к началу координат

Дополнительные задачи

Задача №2. Анти-quick sort

Задача №3. Сортировка пугалом

Задача №7. Цифровая сортировка 12

Вывод **13**

Задачи по варианту

Задача №1. Улучшение Quick sort

1. Используя псевдокод процедуры Randomized - QuickSort, а так же Partition из презентации к Лекции 3 (страницы 8 и 12), напишите программу быстрой сортировки на Python и проверьте ее
2. Основное задание. Цель задачи - переделать данную реализацию рандомизированного алгоритма быстрой сортировки, чтобы она работала быстро даже с последовательностями, содержащими много одинаковых элементов. Чтобы заставить алгоритм быстрой сортировки эффективно обрабатывать последовательности с несколькими уникальными элементами, нужно заменить двухстороннее разделение на трехстороннее (смотри в Лекции 3 слайд 17).

```
import random
from lab_3.utils import *
from lab_3.utils_for_tests import *


@timeit
def partition(a: list, l: int, r: int) -> list:
    x = a[l]
    j = l
    for i in range(l + 1, r + 1):
        if a[i] <= x:
            j += 1
            a[i], a[j] = a[j], a[i]
    a[l], a[j] = a[j], a[l]
    return a


def partition3(a: list, l: int, r: int) -> tuple:
    x = a[l]
    m1 = l
    m2 = r
    i = l

    while i <= m2 and m1 <= m2:
        if a[i] < x:
            a[m1], a[i] = a[i], a[m1]
            m1 += 1
            i += 1
        elif a[i] > x:
            a[i], a[m2] = a[m2], a[i]
            m2 -= 1
        elif a[i] == x:
            i += 1
    return m1, m2
```

```
def randomized_quicksort(a: list, l: int, r: int) -> list:
    if l < r:
        k = random.randint(l, r)
        a[l], a[k] = a[k], a[l]
        m1, m2 = partition3(a, l, r)
        randomized_quicksort(a, l, m1 - 1)
        randomized_quicksort(a, m2 + 1, r)
    return a


if __name__ == "__main__":
    work(randomized_quicksort, 0)
```

Функция partition:

Эта функция выполняет простое разделение массива на основе опорного элемента (первый элемент массива). Она перемещает все элементы, меньшие или равные опорному элементу, влево. В конце функция возвращает измененный массив и выводит его на экран.

Функция partition3:

Эта функция реализует трехпартитное разделение массива. Она делит массив на три части. Это позволяет эффективно обрабатывать массивы с большим количеством одинаковых элементов и улучшает производительность алгоритма в таких случаях.

Функция randomized_quicksort:

Эта функция реализует сам алгоритм быстрой сортировки. Она выбирает случайный опорный элемент из подмассива и использует partition(3) для разделения массива на три части. Затем она рекурсивно сортирует две полученные подчасти.

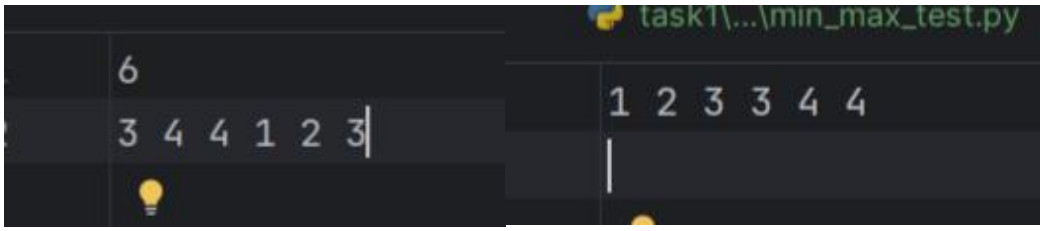
Сравнение разных сортировок:

	Merge Sort		
n	10^3	10^4	10^5
Время	0.0014175 секунд	0.019536999985575676 секунд	0.2018225999781862 секунд
Память	0.00787325 MB	0.0765380859375 MB	0.76318359375 MB

	Partition		
n	10^3	10^4	10^5
Время	0.00083250005263835 19 секунд	0.0109183000167831 78 секунд	0.161976399947889 15 секунд
Память	0.0017242431640625 MB	0.0029754638671875 MB	0.003890991210937 5 MB

	Partition3		
n	10^3	10^4	10^5
Время	0.001217999961227178 6 секунд	0.0168575999559834 6 секунд	0.247054500039666 9 секунд
Память	0.00213623046875 MB	0.0034561157226562 5 MB	0.004486083984375 MB

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	6.800051778554916e-06 секунд	6.866455078125e-05 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.22215389995835721 секунд	0.004241943359375 MB

Выводы по задаче:

Время выполнения: Обычно QuickSort будет быстрее Merge Sort для небольших и средних наборов данных, особенно если данные имеют много одинаковых элементов из-за оптимизации Partition3. Однако для

больших наборов данных Merge Sort может оказаться более стабильным и предсказуемым по времени выполнения.
Память: Merge Sort требует дополнительной памяти для хранения временных массивов во время слияния, тогда как QuickSort работает на месте, но может потребовать дополнительной памяти для стека вызовов в случае глубоких рекурсий.

Задача №5.

Для заданного массива целых чисел citations, где каждое из этих чисел - число цитирований i-ой статьи ученого-исследователя, посчитайте индекс Хирша этого ученого. По определению Индекса Хирша на Википедии: Учёный имеет индекс h, если h из его/её Np статей цитируются как минимум h раз каждая, в то время как оставшиеся (Np − h) статей цитируются не более чем h раз каждая. Иными словами, 5 учёный с индексом h опубликовал как минимум h статей, на каждую из которых сослались как минимум h раз. Если существует несколько возможных значений h, в качестве h-индекса принимается максимальное из них.

```
from lab_3.utils import *
from lab_3.task1.src.task1 import partition3


def hirsh_index(a: list, l: int, r: int) -> int:
    res = []
    for i in range(r):
        k = i
        a[l], a[k] = a[k], a[l]
        m1, m2 = partition3(a, l, r)
        if m1 <= r + 1 - a[m1]:
            res += [a[m1]]
    if len(res) == 0:
        return -1
    else:
        return max(res)


if __name__ == "__main__":
    work(hirsh_index, 0)
```

Функция hirsh_index:

• Создается пустой список res для хранения подходящих значений индекса.

• В цикле перебираются все элементы массива от 0 до r:

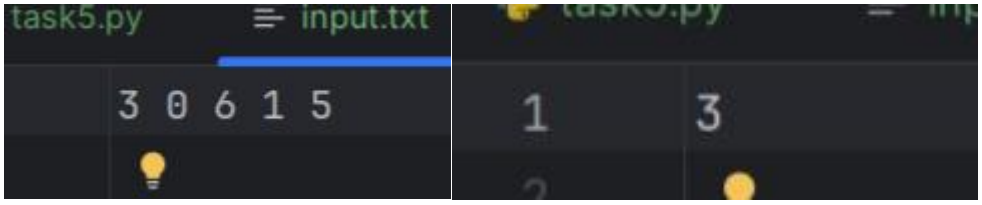
• Каждый элемент на позиции k меняется местами с элементом на позиции l.

• Затем вызывается функция partition3, которая делит массив на три части.

• Проверяется условие: если количество элементов, которые больше или равны индексу m1, больше или равно индексу m1, то значение добавляется в список res.

• Если список res пустой, возвращается -1. В противном случае возвращается максимальное значение из списка.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	5.200039595365524e-06 секунд	7.62939453125e-05 MB
Верхняя граница диапазона значений входных данных из текста задачи	7.777868499979377 секунд	0.000244140625 MB

Если бы массив был отсортирован, то можно было бы просто циклом пройтись с конца до элемента, для которого значение счётчика равняется его значению.

Вывод по задаче: Быстрая сортировка эффективно решает разные задачи.

Задача №8. K ближайших точек к началу координат

В этой задаче, ваша цель - найти K ближайших точек к началу координат среди данных n точек. • Цель. Заданы n точек на поверхности, найти K точек, которые находятся ближе к началу координат (0, 0), т.е. имеют наименьшее расстояние до начала координат. Напомним, что расстояние между двумя точками (x1, y1) и (x2, y2) равно sqrt((x1 − x2) ^2 + (y1 − y2) ^2).

```
def distance(t: list) -> float:
    x = t[0]
    y = t[1]
    return sqrt(x**2 + y**2)


def partition3(a: list, l: int, r: int) -> tuple:
    x = a[l]
    m1 = l
    m2 = r
    i = l

    while i <= m2 and m1 <= m2:
        if distance(a[i]) < distance(x):
            a[m1], a[i] = a[i], a[m1]
            m1 += 1
            i += 1
        elif distance(a[i]) > distance(x):
            a[i], a[m2] = a[m2], a[i]
            m2 -= 1
        elif distance(a[i]) == distance(x):
            i += 1
    return m1, m2


def randomized_quicksort(a: list, l: int, r: int) -> list:
    if l < r:
        k = random.randint(l, r)
        a[l], a[k] = a[k], a[l]
        m1, m2 = partition3(a, l, r)
        randomized_quicksort(a, l, m1 - 1)
        randomized_quicksort(a, m2 + 1, r)
    return a


def nearest_point(n: int, k: int, a: list) -> list:
    a = randomized_quicksort(a, 0, n - 1)
    return a[:k]


if __name__ == "__main__":
    work(nearest_point)
```

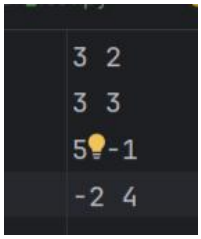
Функция distance:

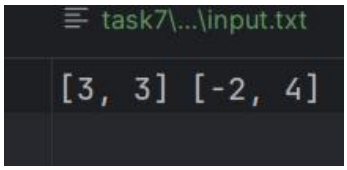
Эта функция принимает координаты точки t (где t — это список или кортеж с двумя элементами: x и y) и возвращает расстояние от этой точки до начала координат (0, 0).

Функция partition3:

Обычный partition3, но только сравнение относительно расстояний от точки до начала координат.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	2.300017513334751e-06 секунд	7.62939453125e-06 MB
Верхняя граница диапазона значений входных данных из текста задачи	1.5365257000084966 секунд	0.763031005859375 MB

Вывод по задаче: Сортировку можно модифицировать под разные ситуации.

Дополнительные задачи

Задача №2. Анти-quick sort.

Хотя QuickSort является очень быстрой сортировкой в среднем, существуют тесты, на которых она работает очень долго. Оценивать время работы алгоритма будем числом сравнений с элементами массива (то есть, суммарным числом сравнений в первом и втором while). Требуется написать программу, генерирующую тест, на котором быстрая сортировка сделает наибольшее число таких сравнений.

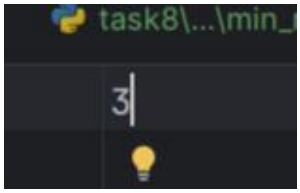
.

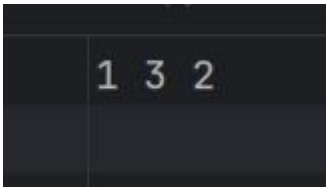
```
from lab_3.utils import *


def qsort(a: list, left: int, right: int) -> list:
    key = a[(left + right) // 2]
    i = left
    j = right
    while i <= j:
        while a[i] < key:
            i += 1
        while a[j] > key:
            j -= 1
        if i <= j:
            if a[i] != a[j]:
```

```
                print(a, 1, end=' ')
            a[i], a[j] = a[j], a[i]
            print(a)
            i += 1
            j -= 1

        if left < j:
            qsort(a, left, j)
        if i < right:
            qsort(a, i, right)
        return a


def anti_qsort(n: int) -> list:
    a = [i + 1 for i in range(n)]
    for i in range(2, len(a)):
        a[i], a[i // 2] = a[i // 2], a[i]
    return a


if __name__ == "__main__":
    work(anti_qsort)
```

Алгоритм быстрой сортировки работает лучше всего на сбалансированных разделениях массива. Если опорный элемент выбирается неудачно (например, всегда самый маленький или самый большой элемент), это может привести к неэффективной работе алгоритма. В случае массива, созданного функцией anti_qsort, опорный элемент будет часто находиться в позиции, которая приводит к очень неравномерному разделению (например, один элемент будет в одной части массива, а остальные — в другой).

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	1.800013706088066e-06 секунд	0.0001068115234375 MB
Верхняя граница диапазона значений входных данных из	0.010055600083433092 секунд	3.807903289794922 MB

текста задачи		

Вывод по задаче: Ничто не идеально, для любой сортировки можно придумать пример с наихудшим случаем времени.

Задача №3 Сортировка пугалом.

«Сортировка пугалом» — это давно забытая народная потешка. Участнику под верхнюю одежду продевают деревянную палку, так что у него оказываются растопырены руки, как у огородного пугала. Перед ним ставятся n матрёшек в ряд. Из-за палки единственное, что он может сделать — это взять в руки две матрешки на расстоянии k друг от друга (то есть i-ую и i + k-ую), развернуться и поставить их обратно в ряд, таким образом поменяв их местами. Задача участника — расположить матрёшки по неубыванию размера. Может ли он это сделать?

```
from lab_3.utils import *


def is_correct(a: list) -> bool:
    for i in range(1, len(a)):
        if a[i] < a[i - 1]:
            return False
    return True


def bugaboo(n: int, k: int, a: list) -> str:
    for i in range(n - k):
        if a[i] > a[i + k]:
            a[i], a[i + k] = a[i + k], a[i]
    if is_correct(a):
        return "ДА"
    else:
        return "НЕТ"


if __name__ == "__main__":
    work(bugaboo)
```

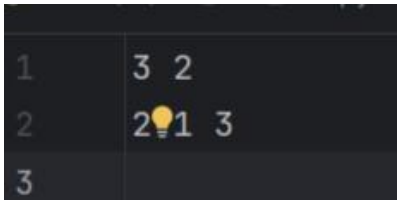
Функция bugaboo:

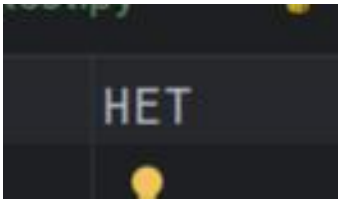
1. Проходит по всем индексам от 0 до n-k-1 (то есть до элемента, у которого есть соответствующий элемент на позиции i + k).

2. Сравнивает элемент a[i] с элементом a[i + k]. Если a[i] больше, они меняются местами.

3. После завершения сортировки (или частичной сортировки) массив проверяется на правильность с помощью функции is_correct.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	3.400025889277458e-06 секунд	7.62939453125e-05 MB
Верхняя граница диапазона значений входных данных из текста задачи	0.008153600036166608 секунд	0.000164031982421875 MB

Задача №7. Цифровая сортировка

Дано n строк, выведите их порядок после k фаз цифровой сортировки.

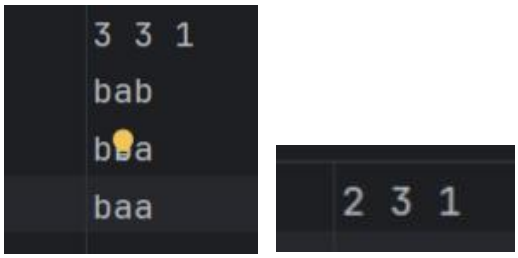
```
from lab_3.utils import *


def radix_sort(n: int, m: int, k: int, a: list) -> list:
    original = a
    alpha = 26
    for i in range(m - 1, m - k - 1, -1):
        res = [''] * n
        c = [0] * alpha

        for j in range(n):
            d = ord(a[j][i]) - ord('a')
            c[d] += 1

        count = 0
        for j in range(alpha):
            tmp = c[j]
            c[j] = count
            count += tmp

        for j in range(n):
            d = ord(a[j][i]) - ord('a')
            res[c[d]] = a[j]
            c[d] += 1
        a = res
    ans = [original.index(a[i]) + 1 for i in range(len(a))]
    return ans


if __name__ == "__main__":
    work(radix_sort)
```

Внешний цикл проходит по символам строк с конца (от последнего символа к первому). Внутренние циклы выполняют следующие действия:

- Подсчет частоты символов
- Преобразование массива частот в массив индексов, чтобы знать, на какие позиции в результирующем массиве будут помещены строки
- Сборка отсортированных строк

В конце производится сборка отсортированных строк.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	5.799927748739719e-06 секунд	0.00069427490234375 MB
Верхняя граница диапазона значений входных данных из текста задачи	2.3678527999436483 секунд	0.00220489501953125 MB

Вывод о задаче: Изучила поразрядную сортировку.

Вывод:

В процессе выполнения Лабораторной работы №3 были реализованы быстрая сортировка сортировки за линейное время, проверены умение работать с файлами, создание тестов минимальных, максимальных значений и оценка времени работы, затрат памяти для случайных больших данных.