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 1

Выполнил:
Бен Шамех Абделазиз
Группа: K3239

Проверила:
Ромакина Оксана Михайловна

Санкт-Петербург
2025 г.

Содержание

1 Задача 1. Улучшение Quick sort

1.1 Условие

Переделать рандомизированный алгоритм быстрой сортировки на трехстороннее разделение (Partition3), чтобы эффективно обрабатывать массивы с большим количеством одинаковых элементов.

1.2 Листинг кода

```python
import random

def quick_sort_p3(arr, l, r):
    if l < r:
        lt, gt = partition3(arr, l, r)
        quick_sort_p3(arr, l, lt - 1)
        quick_sort_p3(arr, gt + 1, r)

def partition3(arr, l, r):
    x_ind = random.randint(l, r)
    arr[l], arr[x_ind] = arr[x_ind], arr[l]
    x = arr[l]
    lt = l
    eq = l
    gt = r
    while eq <= gt:
        if arr[eq] < x:
            arr[lt], arr[eq] = arr[eq], arr[lt]
            lt += 1
            eq += 1
        elif arr[eq] > x:
            arr[gt], arr[eq] = arr[eq], arr[gt]
            gt -= 1
        else:
            eq += 1
    return lt, gt
```

1.3 Текстовое объяснение

Разделяем массив на три части: меньше опорного, равные ему и больше него. Это позволяет алгоритму работать за $O(n)$ в случаях, когда в массиве много дубликатов, предотвращая деградацию до $O(n^2)$.

1.4 Результаты выполнения

- **Input:** `5 \n 2 3 9 2 2`
- **Output:** `2 2 2 3 9`

Метрики эффективности:

Сценарий	**Время (сек)**	**Память (Mb)**
Пример из задачи	0.00894 sec	30.25 Mb
Верхняя граница (n=100000)	0.16941 sec	31.46 Mb

2 Задача 2. Анти-quick sort

2.1 Условие

Написать программу, генерирующую перестановку чисел от 1 до n, на которой быстрая сортировка с выбором среднего элемента выполнит максимальное число сравнений.

2.2 Листинг кода

```
def generate_worst_case(ln):
    arr = []
    for i in range(0, ln):
        arr += [i + 1]
        if i > 1:
            arr[-1], arr[i // 2] = arr[i // 2], arr[-1]
    return arr
```

2.3 Текстовое объяснение

Добавляем числа по порядку и на каждом шаге меняем последний элемент с центральным. Это гарантирует, что при разделении относительно среднего элемента массив всегда будет делиться максимально несбалансированно.

2.4 Результаты выполнения

- **Input:** `3` → **Output:** `1 3 2`
- **Input:** `5` → **Output:** `1 4 5 3 2`

Метрики эффективности:

Сценарий	**Время (сек)**	**Память (Mb)**
Нижняя граница (n=2)	0.00451 sec	30.24 Mb
Верхняя граница (n=100000)	0.08889 sec	31.90 Mb

3 Задача 4. Точки и отрезки

3.1 Условие

Дан набор из s отрезков и p точек. Вычислить для каждой точки, в скольких отрезках она содержится.

3.2 Листинг кода

```python
def points_and_segments(segments, points):
    start_seg_dct = {}
    end_seg_dct = {}
    for a, b in segments:
        start_seg_dct[a] = start_seg_dct.get(a, 0) + 1
        end_seg_dct[b + 1] = end_seg_dct.get(b + 1, 0) + 1

    points_c = points.copy()
    points = sorted(set(points))
    start_seg_arr = sorted(start_seg_dct.keys())
    end_seg_arr = sorted(end_seg_dct.keys())

    i = j = cur_layers = 0
    ans_arr = {}
    for k in range(len(points)):
        while i < len(start_seg_arr) and points[k] >= start_seg_arr[i]:
            cur_layers += start_seg_dct[start_seg_arr[i]]
            i += 1
        while j < len(end_seg_arr) and points[k] >= end_seg_arr[j]:
            cur_layers -= end_seg_dct[end_seg_arr[j]]
            j += 1
        ans_arr[points[k]] = cur_layers
    return [ans_arr[p] for p in points_c]
```

3.3 Текстовое объяснение

Используем метод сканирующей прямой. Сортируем начала и концы отрезков и проходим по ним, поддерживая количество активных отрезков («слоев») для каждой уникальной точки.

3.4 Результаты выполнения

- **Input:** `2 3 \n 0 5 \n 7 10 \n 1 6 11`
- **Output:** `1 0 0`

Метрики эффективности:

Сценарий	Время (сек)	Память (Mb)
Пример из задачи	0.00554 sec	30.25 Mb
Верхняя граница	0.77409 sec	31.79 Mb

4 Задача 6. Сортировка целых чисел

4.1 Условие

Отсортировать массив попарных произведений элементов массивов A и B и найти сумму каждого десятого элемента.

4.2 Листинг кода

```python
def sort_integer_nums(arr_a, arr_b):
    arr = [i * j for i in arr_a for j in arr_b]
    arr.sort() #                          Timsort (O(N log N))
    sm = sum(arr[i] for i in range(0, len(arr), 10))
    return sm
```

4.3 Текстовое объяснение

Генерируем все произведения в один список. Сортируем его и с помощью среза или цикла с шагом 10 суммируем необходимые элементы.

4.4 Результаты выполнения

- **Input:** `4 4 \n 7 1 4 9 \n 2 7 8 11`
- **Output:** `51`

Метрики эффективности:

Сценарий	Время (сек)	Память (Mb)
Нижняя граница	0.0055 sec	30.16 Mb
Пример из задачи	0.00563 sec	30.57 Mb

5 Задача 7. Цифровая сортировка (Radix Sort)

5.1 Условие

Выполнить k фаз поразрядной сортировки (LSD) для n строк длиной m. Формат ввода — вертикальный.

5.2 Листинг кода

```python
import sys

def solve_radix():
    input_data = sys.stdin.read().split()
    if not input_data: return
    n, m, k = int(input_data[0]), int(input_data[1]), int(input_data[2])
    columns = input_data[3:]
    p = list(range(n))
    for step in range(k):
        col_idx = m - 1 - step
        current_col = columns[col_idx]
        buckets = [[] for _ in range(26)]
        for idx in p:
            char_code = ord(current_col[idx]) - 97
            buckets[char_code].append(idx)
        p = [idx for bucket in buckets for idx in bucket]
    print(*(x + 1 for x in p))
```

5.3 Текстовое объяснение

Используем стабильную сортировку подсчетом по каждой позиции (начиная с младшего разряда). Благодаря вертикальному формату ввода, доступ к символу определенного разряда для всех строк происходит максимально быстро.

5.4 Результаты выполнения

- **Input:** `3 3 1 \n bab \n bba \n baa` → **Output:** `2 3 1`

Метрики эффективности:

Сценарий	Время (сек)	Память (Mb)
Пример из задачи	0.003 sec	30.15 Mb
Верхняя граница	2.15 sec	215.40 Mb

6 Задача 8. K ближайших точек к началу координат

6.1 Листинг кода

```python
def find_k_closest_points(points, k):
    #
    points = sorted([[a, b, (a**2 + b**2)**0.5] for a, b in points], key=lambda x: x[2])
    ans = ", ".join([f"[{p[0]},{p[1]}]" for p in points[:k]])
    return ans
```

6.2 Текстовое объяснение

Для каждой точки вычисляем евклидово расстояние до $(0, 0)$. Сортируем список точек по этому значению и выбираем первые K элементов.

6.3 Результаты выполнения

- **Input:** `2 1 \n 1 3 \n -2 2` → **Output:** `[-2,2]`

Метрики эффективности:

Сценарий	Время (сек)	Память (Mb)
Пример из задачи	0.00264 sec	30.13 Mb
Верхняя граница	0.75601 sec	30.20 Mb

7 Задача 9. Ближайшие точки

7.1 Листинг кода

```python
def recursion_pair_closest(arr_s_x, arr_s_y):
    if len(arr_s_x) <= 3:
        min_dist = 10**18
        best_pair = ()
        for i in range(len(arr_s_x)):
            for j in range(i + 1, len(arr_s_x)):
                min_d = find_dist(arr_s_x[i], arr_s_x[j])
                if min_d < min_dist:
                    min_dist, best_pair = min_d, (arr_s_x[i], arr_s_x[j
    ])
        return min_dist, best_pair

    mid = len(arr_s_x) // 2
    mid_sep = arr_s_x[mid][0]
    left_x, right_x = arr_s_x[:mid], arr_s_x[mid:]
    left_y = [d for d in arr_s_y if d[0] <= mid_sep]
    right_y = [d for d in arr_s_y if d[0] > mid_sep]

    d1, p1 = recursion_pair_closest(left_x, left_y)
    d2, p2 = recursion_pair_closest(right_x, right_y)
    min_d, best_pair = (d1, p1) if d1 < d2 else (d2, p2)

    strip = [dot for dot in arr_s_y if abs(dot[0] - mid_sep) < min_d]
    for i in range(len(strip)):
        for j in range(i + 1, min(i + 7, len(strip))):
            ds = find_dist(strip[i], strip[j])
            if ds < min_d:
                min_d, best_pair = ds, (strip[i], strip[j])
    return min_d, best_pair

def find_dist(dot1, dot2):
    return ((dot1[0] - dot2[0])**2 + (dot1[1] - dot2[1])**2)**0.5
```

7.2 Текстовое объяснение

Используется метод «Разделяй и властвуй». Сортируем точки по координате X, делим массив пополам, рекурсивно ищем минимум в каждой части и проверяем «полосу» шириной $2d$ вдоль линии раздела, чтобы учесть пары точек из разных половин.

7.3 Результаты выполнения

- **Input:** 11 точек → **Output:** `1.4142`

Метрики эффективности:

Сценарий	Время (сек)	Память (Mb)
Пример из задачи	0.00253 sec	30.49 Mb
Верхняя граница	0.81082 sec	30.41 Mb

8 Вывод

1. В ходе лабораторной работы были изучены и реализованы алгоритмы быстрой сортировки (*Quick Sort*) и методы её оптимизации (Partition 3).

2. Реализована эффективная цифровая сортировка (*Radix Sort*), работающая за линейное время $O(n \cdot k)$ для строковых данных.

3. Применен метод «Разделяй и властвуй» для решения классической задачи нахождения пары ближайших точек за $O(n \log n)$.

4. Было практически подтверждено, что выбор правильного алгоритма (линейного или логарифмического вместо квадратичного) позволяет обрабатывать огромные наборы данных (до 10^6 элементов) в рамках заданных ограничений по времени.