Міністерство освіти і науки України
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра ІПІ

Звіт

з лабораторної роботи №3 з дисципліни
«Алгоритми та структури даних 2. Структури даних»

«**Метод швидкого сортування**»

Виконав ІП-23 Зубарев Микола Костянтинович

Перевірив Соколовський В.В.

Київ 2023

ЗМІСТ

1. МЕТА ЛАБОРАТОРНОЇ РОБОТИ

Мета роботи – вивчити основні методи аналізу обчислювальної складності алгоритмів швидкого сортування і оцінити поріг їх ефективності.

2.ЗАВДАННЯ

Виконати аналіз алгоритму швидкого сортування на відповідність наступним властивостям (таблиця 2.1):

- стійкість;
- «природність» поведінки (Adaptability);
- базуються на порівняннях;
- необхідність додаткової пам'яті (об'єму);
- необхідність в знаннях про структуру даних.

Записати алгоритм внутрішнього сортування за допомогою псевдокоду (чи іншого способу по вибору).

Провести аналіз часової складності в гіршому, кращому і середньому випадках та записати часову складність в асимптотичних оцінках.

Виконати програмну реалізацію алгоритму на будь-якій мові програмування з фіксацією часових характеристик оцінювання (кількість порівнянь, кількість перестановок, глибина рекурсивного поглиблення та іншев залежності від алгоритму).

Провести ряд випробувань алгоритму на масивах різної розмірності (10, 100, 1000, 5000, 10000, 20000, 50000 елементів) і різних наборів вхідних даних (впорядкований масив, зворотно упорядкований масив, масив випадкових чисел) і побудувати графіки залежності часових характеристик оцінювання від розмірності масиву, нанести на графік асимптотичну оцінку гіршого і кращого випадків для порівняння.

Зробити порівняльний аналіз двох алгоритмів.

Зробити узагальнений висновок з лабораторної роботи.

Таблиця 2.1 – Варіанти алгоритмів

№	**Алгоритм сортування**
1	Звичайне швидке сортування
2	Швидке сортування з 3-медіаною в якості опорного елемента

3. ВИКОНАННЯ

3.1 Аналіз алгоритму на відповідність властивостям

Аналіз алгоритму звичайного швидкого сортування та швидкого сортування з медіаною в якості опорного елемента на відповідність властивостям наведено в таблиці 3.1.

Таблиця 3.1 – Аналіз алгоритму на відповідність властивостям

Властивість	**Quick Sort**	**Quick Sort (з медіаною)**
Стійкість	Ні	Ні
«Природність» поведінки	Так	Так
Базуються на порівняннях	Так	Так
Необхідність в додатковій пам'яті(об'єм)	Ні	Ні
Необхідність в знаннях про структури даних	Необхідні базові знання про масиви	Необхідні базові знання про масиви

3.2 Псевдокод алгоритму

```
Алгоритм QuickSort(arr, low, high) {
    if (low < high) {
      pi = partition(arr, low, high);
      QuickSort(arr, low, pi – 1); // Before pi
      QuickSort(arr, pi + 1, high); // After pi
    }
}
Функція partition (arr, low, high){
    pivot = arr[high];
    i = (low – 1)
    for (j = low; j <= high- 1; j++){
      if (arr[j] < pivot){
          i++
```

```
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

3.3 Аналіз часової складності

Часова складність алгоритму швидкого сортування залежить від кількості елементів, які потрібно відсортувати, і від того, які елементи включено до сортування.

Сортування	**Найкращий (впорядкований)**	**Випадковий**	**Найгірший(впорядкований у зворотньому напрямку)**
QuickSort	O(n*log(n))	O(n*log(n))	$O(n^2)$
QuickSort з медіаною	O(n*log(n))	O(n*log(n))	$O(n^2)$

3.4 Програмна реалізація алгоритму

3.4.1 Вихідний код

```python
import matplotlib.pyplot as plt
import numpy as np
import random
import math

def InsertionSort(arr, flag):
    count = count2 = 0
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0:
            count += 1
            if key < arr[j]:
                arr[j + 1] = arr[j]
                j -= 1
                count2 += 1
            else:
                break
        arr[j + 1] = key
    if flag:
        return count
    else:
```

```
        return count + count2

def partition(arr, low, high, flag):
    pivot = arr[high]
    i = low - 1
    count = 0
    count2 = 0
    for j in range(low, high):
        count += 1
        if arr[j] <= pivot:
            i += 1
            count2 += 1
            arr[i], arr[j] = arr[j], arr[i]
    count2 += 1
    arr[i+1], arr[high] = arr[high], arr[i+1]
    if flag:
        return i + 1, count
    else:
        return i + 1, count + count2

def median(arr, r, l):
    mid = (r + l) // 2
    if arr[r] < arr[mid] < arr[l] or arr[l] < arr[mid] <
arr[r]:
        return mid
    elif arr[mid] <= arr[r] <= arr[l] or arr[l] <= arr[r] <=
arr[mid]:
        return r
    else:
        return l

def partition_median(array, left, right, flag):
    count = 0
    count2 = 0
    mid = median(array, left, right)
    array[mid], array[right] = array[right], array[mid]
    pivot = array[right]
    i = left - 1
    for j in range(left, right):
        count += 1
        if array[j] <= pivot:
            i += 1
            count2 += 1
            array[i], array[j] = array[j], array[i]
    count2 += 1
    array[i+1], array[right] = array[right], array[i+1]
    if flag:
        return i + 1, count
    else:
        return i + 1, count + count2

def OuickSort(arr, low, high, flag):
    if low < high:
        pivot_index, comparison = partition(arr, low, high,
flag)
        comparison += OuickSort(arr, low, pivot_index-1,
flag)[1]
```

```
        comparison += QuickSort(arr, pivot_index+1, high,
flag)[1]
        return arr, comparison
    else:
        return arr, 0

def QuickSortWithMedian(array, left, right, flag):
    result = 0
    if right - left > 2:
        pivotindex, result1 = partition_median(array, left,
right, flag)
        result += result1
        result += QuickSortWithMedian(array, left, pivotindex-
1, flag)
        result += QuickSortWithMedian(array, pivotindex + 1,
right, flag)
    else:
        result += InsertionSort(array[left:right + 1], flag)
    return result

def apr(sizes, arr):
    fp = np.polyfit(sizes, arr, 2)
    f = np.poly1d(fp)
    return f

def choice():
    print("1. File input")
    print("2. Enter manually")
    ch = int(input("Enter your choice: "))
    if ch == 1:
        file_add()
    elif ch == 2:
        graph()
    return

def graph():
    n = int(input("Enter the size of the array: "))
    a1 = list(range(1, n + 1))
    a2 = list(range(n, 0, -1))
    a3 = random.sample(range(1, n+1), n)
    sizes = range(1, n + 1)
    flag = False
    w = [i ** 2 for i in sizes]
    b = [i * math.log(i) for i in sizes]
    o1 = [QuickSort(a1[:size], 0, size-1, flag) for size in
sizes]
    o2 = [QuickSort(a2[:size], 0, size-1, flag) for size in
sizes]
    o3 = [QuickSort(a3[:size], 0, size-1, flag) for size in
sizes]
    op1 = [tup[1] for tup in o1]
    op2 = [tup[1] for tup in o2]
    op3 = [tup[1] for tup in o3]
    op1_m = [QuickSortWithMedian(a1[:size], 0, size-1, flag)
for size in sizes]
    op2_m = [QuickSortWithMedian(a2[:size], 0, size-1, flag)
for size in sizes]
```

```
    op3_m = [OuickSortWithMedian(a3[:size], 0, size-1, flag)
for size in sizes]
    gr1 = apr(sizes, op1)
    gr2 = apr(sizes, op2)
    gr3 = apr(sizes, op3)
    gr1_m = apr(sizes, op1_m)
    gr2_m = apr(sizes, op2_m)
    gr3_m = apr(sizes, op3_m)
    plt.title('Quick Sort')
    plt.plot(sizes, w, color='purple', label="Worst time
asymptote", linewidth=0.8)
    plt.plot(sizes, b, 'y', label="Best time asymptote",
linewidth=0.8)
    plt.plot(sizes, gr1(sizes), 'g', label="Best",
linewidth=0.8)
    plt.plot(sizes, gr2(sizes), 'r', label="Worst",
linewidth=0.8)
    plt.plot(sizes, gr3(sizes), 'b', label="Random",
linewidth=0.8)
    plt.xlabel('Elements')
    plt.ylabel('Operations')
    plt.legend(loc="upper left")
    plt.show()
    plt.title('Quick Sort with median')
    plt.plot(sizes, w, color='purple', label="Worst time
asymptote",linewidth=0.8)
    plt.plot(sizes, b, 'y', label="Best time asymptote",
linewidth=0.8)
    plt.plot(sizes, gr1_m(sizes), 'g', label="Best",
linewidth=0.8)
    plt.plot(sizes, gr2_m(sizes), 'r', label="Worst",
linewidth=0.8)
    plt.plot(sizes, gr3_m(sizes), 'b', label="Random",
linewidth=0.8)
    plt.xlabel('Elements')
    plt.ylabel('Operations')
    plt.legend(loc="upper left")
    plt.show()

def file_add():
    name = input("Enter the file name: ")
    file_path = name

    with open(file_path, 'r') as file:
        numbers = [int(line.strip()) for line in
file.readlines()]
    del numbers[0]
    size = len(numbers)
    numbers2 = numbers.copy()
    flag = True
    array, comp = OuickSort(numbers, 0, size - 1, flag)
    comp2 = OuickSortWithMedian(numbers2, 0, size - 1, flag)
    newname = name[5:-4]
    newname = "Zubarev_ip23" + newname + "_output.txt"

    with open(newname, 'w') as out:
        out.write(str(comp) + " " + str(comp2))
```

```
            out.write("\n")
        out.close()
        print(str(comp) + " " + str(comp2))

if __name__ == '__main__':
    choice()
```

3.5 Тестування алгоритму

На рисунках показані приклади роботи програми сортування двох алгоритмів для масивів на 10, 100 і 500 елементів відповідно. На рисунках упорядкований масив - червоний графік, зворотно упорядкований масив - чорний графік, випадкова послідовність - синій графік, також показані асимптотичні оцінки гіршого – пурпурний(маджента) графік і кращого - жовтий графік - випадків для порівняння.

3.5.1 Графіки залежності часових характеристик оцінювання від розмірності масиву

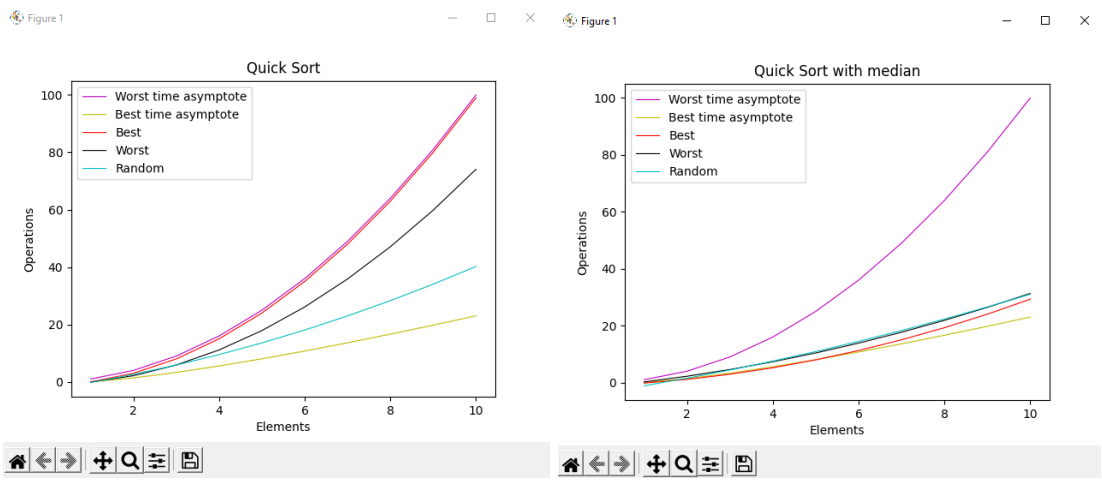

Рисунок 3.5.1(1) – Сортування масиву на 10 елементів

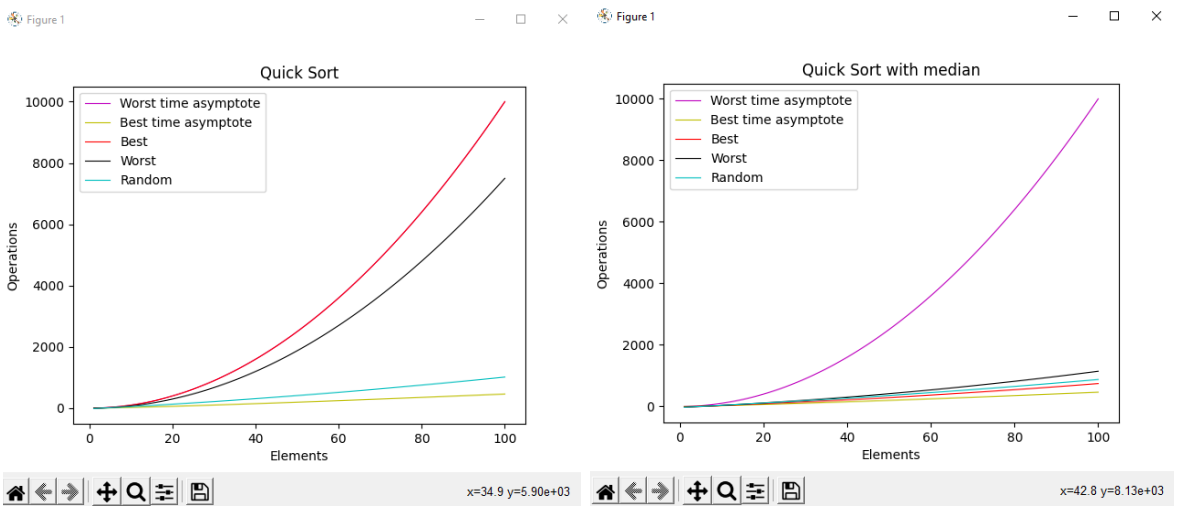

Рисунок 3.5.1(2) – Сортування масиву на 100 елементів

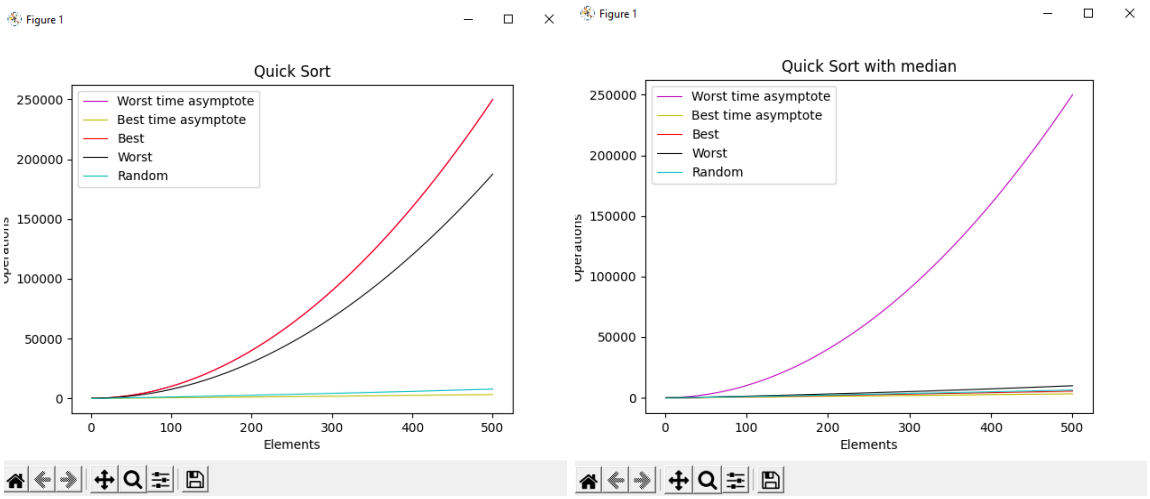

Рисунок 3.5.1(3) – Сортування масиву на 500 елементів

4. ВИСНОВОК

У ході даної роботи було реалізовано дві модифікації алгоритму швидкого сортування (Quick Sort) і порівняно їх швидкодію на основі підрахунку кількості порівнянь елементів масиву під час виконання алгоритмів.

Результати показали, що швидкість алгоритмів швидкого сортування значно залежить від властивостей вхідних даних. У найкращому випадку, коли дані вже відсортовані або розташовані випадково, всі модифікації Quick Sort демонструють подібну швидкість з часовою складністю O(nlogn). Однак, у найгіршому випадку, коли масив відсортований в зворотному порядку, часова складність може досягати O(n^2), що робить алгоритми менш ефективними.

Для порівняння швидкодії модифікацій алгоритму Quick Sort було використано підрахунок кількості порівнянь елементів масиву. Цей критерій дозволяє оцінити ефективність алгоритмів та порівняти їх між собою. Проте варто зауважити, що кількість порівнянь може бути лише однією з багатьох метрик оцінки швидкодії алгоритмів.

Загалом, робота над реалізацією та порівнянням модифікацій алгоритму швидкого сортування дозволила отримати уявлення про їхню швидкодію залежно від властивостей вхідних даних.