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(ла) *ІП-22, Андреєва Уляна Андріївна*

(шифр, прізвище, ім'я, по батькові)

Перевірила *Халус Олена Андріївна*

(прізвище, ім'я, по батькові)

Київ 2023

Практичне завдання №3

Метод швидкого сортування

Завдання

Реалізувати наступні три модифікації алгоритму швидкого сортування (Quick Sort) та порівняти їх швидкодію. Швидкість алгоритмів порівнюється на основі підрахунку кількості порівнянь елементів масиву під час роботи алгоритмів.

Код програми

Task.java

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;

public class Task {
    public static void main(String[] args) throws IOException {
        String fileName = args[0];
       File myFile = new File ( fileName );
        StringBuilder sb = new StringBuilder();
        try(FileReader reader = new FileReader(myFile)) {
            int c;
            while ((c = reader.read()) != -1) {
                sb.append((char)c);
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        String fileContent = sb.toString();
        System.out.println(fileContent);


        ArrayList <String> splittedArray2 = new ArrayList<>(
Arrays.asList(fileContent.split("\n")));
        splittedArray2.remove(0);
        ArrayList<Integer> integerArrayList = new ArrayList<> ();
        for (String element: splittedArray2) {
            integerArrayList.add ( Integer.parseInt ( element ) );
        }
        //for quicksort
        ArrayList<Integer> array1 = new ArrayList<Integer>
(integerArrayList);
        QuickSort sorter = new QuickSort();
        sorter.quickSort (array1, 0, array1.size () - 1);
        int comparison1 = sorter.comparisons;
        //for medianQuickSort

        ArrayList<Integer> array2 = new ArrayList<Integer>
(integerArrayList);
        int comparisons2 = QuickSortMedian.quickSort ( array2,0,array2.size
() - 1 );
        //for QuickSortThree
```

```
        int [] array3 = integerArrayList.stream ().mapToInt ( i -> i
).toArray ();

        QuickSortThree sorter2 = new QuickSortThree();
        sorter2.sort ( array3 );
        int comparisons3 = sorter2.comparisons;

        FileWriter myFile1 = new FileWriter ("ip22_andreieva_03_output.txt"
);
        myFile1.write ( comparison1 + " " + comparisons2 + " " +
comparisons3);
        myFile1.close ();
    }
}
```

QuickSort.java

```
import java.util.ArrayList;

public class QuickSort {
    public int comparisons = 0;

    static void swap(ArrayList<Integer> arr, int a, int b) {
        int temp = arr.get ( a );
        arr.set ( a, arr.get ( b ) );
        arr.set ( b, temp );
    }

    int partition(ArrayList<Integer> arr, int low, int high) {
        int pivot = arr.get ( high );
        int i = low - 1;

        for (int j = low; j <= high - 1; j++) {
            this.comparisons++;
            if (arr.get ( j ) <= pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return i + 1;
    }

    public void quickSort(ArrayList<Integer> arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
}
```

QuickSortMedian.java

```
import java.util.ArrayList;
import java.io.*;
```

```java
public class QuickSortMedian {
    static int comparisonCount = 0;


    static void swap(ArrayList<Integer> arr, int i, int j)
    {
        int temp = arr.get ( i );
        arr.set ( i, arr.get ( j ) );
        arr.set ( j, temp );
    }

    static int median(ArrayList<Integer> arr, int a, int b, int c){
        if(arr.get ( a ) > arr.get ( b )){
            if(arr.get ( b ) > arr.get ( c ))
                return b;
            else if(arr.get ( a ) > arr.get ( c ))
                return c;
            else
                return a;
        }
        else{
            if(arr.get ( b ) < arr.get ( c ))
                return b;
            else if(arr.get ( a ) < arr.get ( c ))
                return c;
            else
                return a;
        }
    }

    static int partition(ArrayList<Integer> arr, int low, int high)
    {

        int medianIndex = median(arr, low, (low + high) / 2, high);
        int pivot = arr.get ( medianIndex );
        swap(arr, medianIndex, high);

        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {
            comparisonCount++;
            if (arr.get ( j ) < pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }
    public static int quickSort(ArrayList<Integer> arr, int low, int high)
    {
        if (low < high) {
            if (high - low > 2) { // compare only if size > 3
                int pi = partition(arr, low, high);
                quickSort(arr, low, pi - 1);
                quickSort(arr, pi + 1, high);
            } else {
// sorting without partitioning
                for (int i = low; i < high; i++) {
                    for (int j = i + 1; j <= high; j++) {
                        comparisonCount++;
```

```
                        if (arr.get ( i ) > arr.get ( j )) {
                            swap(arr, i, j);
                        }
                    }
                }
            }
        }
        return comparisonCount;
    }
}
```

QuickSortThree.java

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

public class QuickSortThree extends Sort {
    public void sort(int[] arrayToSort) {
        quickSort(arrayToSort, 0, arrayToSort.length - 1);
    }

    private void quickSort(int[] array, int lowIndex, int highIndex) {
        if (lowIndex >= highIndex) {
            return;
        }

        int[] pivots = partition(array, lowIndex, highIndex);
        comparisons+=3; // increment counter by 3 for the comparisons in
above line
        if (pivots != null) {
            quickSort(array, lowIndex, pivots[0] - 1);
            quickSort(array, pivots[0] + 1, pivots[1] - 1);
            quickSort(array, pivots[1] + 1, pivots[2] - 1);
            quickSort(array, pivots[2] + 1, highIndex);
        }
    }

    private int[] partition (int[] inArray, int lowIndex, int highIndex) {
        int aPointer = lowIndex + 2, bPointer = lowIndex + 2;
        int cPointer = highIndex - 1, dPointer = highIndex - 1;

        int lowPivot, middlePivot, highPivot;
        if (divisible(inArray, lowIndex, highIndex)) {

            // знаходимо pivot-и
            int[] array = new int[inArray.length];
            List<Integer> differentNumbers = new LinkedList<>();
            List<Integer> differentNumbersIndexes = new LinkedList<>();

            for (int i = lowIndex; i < highIndex; i++) {
               /// comparisons++;


                if (!differentNumbers.contains(inArray[i])) {
                    differentNumbers.add(inArray[i]);
                    differentNumbersIndexes.add(i);
                }
```

```
                if (differentNumbers.size() == 3) {
                    int arrayCounter = lowIndex + 2;
                    for (int j = lowIndex + 2; j <= highIndex; j++) {
                        comparisons++;
                        if (!differentNumbersIndexes.contains(j)) {
                            array[arrayCounter] = inArray[j];
                            arrayCounter++;
                        }
                    }
                    break;
                }

            }
            pivotSort(differentNumbers, differentNumbersIndexes, 0,
differentNumbers.size() - 1);
            array[lowIndex] = differentNumbers.get(0);
            array[lowIndex + 1] = differentNumbers.get(1);
            array[highIndex] = differentNumbers.get(2);

            for (int i = lowIndex; i < highIndex + 1; i++) {
                //comparisons++;

                inArray[i] = array[i];
            }

            lowPivot = inArray[lowIndex];
            middlePivot = inArray[lowIndex + 1];
            highPivot = inArray[highIndex];

            // partition algorithm
            while (bPointer <= cPointer) {
                //comparisons++;

                while (inArray[bPointer] < middlePivot && bPointer <=
cPointer) {
                    //comparisons++;

                    if (inArray[bPointer] < lowPivot) {
                        //comparisons++;
                        swap(inArray, aPointer, bPointer);
                        aPointer++;
                    }
                    bPointer++;
                    // comparisons++;

                }
                while (inArray[cPointer] > middlePivot && bPointer <=
cPointer) {


                    if (inArray[cPointer] > highPivot) {
                        swap(inArray, cPointer, dPointer);
                        dPointer--;
                    }
                    cPointer--;

                }

                if (bPointer <= cPointer) {
                    if (inArray[bPointer] > highPivot) {
```

```
                    if (inArray[cPointer] < lowPivot) {
                        swap(inArray, bPointer, aPointer);
                        swap(inArray, aPointer, cPointer);
                        aPointer++;

                    } else {

                        swap(inArray, bPointer, cPointer);
                    }
                    swap(inArray, cPointer, dPointer);
                    bPointer++;
                    cPointer--;
                    dPointer--;
                    //comparisons++;
                } else {
                    if (inArray[cPointer] < lowPivot) {
                        swap(inArray, bPointer, aPointer);
                        swap(inArray, aPointer, cPointer);
                        aPointer++;

                    } else {
                        swap(inArray, bPointer, cPointer);
                    }
                    bPointer++;
                    cPointer--;
                }
            }
        }
        aPointer--;
        bPointer--;
        dPointer++;

        swap(inArray, lowIndex + 1, aPointer);
        swap(inArray, aPointer, bPointer);
        int newMiddlePivotPosition = bPointer;
        aPointer--;
        swap(inArray, lowIndex, aPointer);
        int newLowPivotPosition = aPointer;
        swap(inArray, highIndex, dPointer);
        int newHighPivotPosition = dPointer;

        return new int[] {newLowPivotPosition, newMiddlePivotPosition,
newHighPivotPosition};

    } else {
        insertionSort(inArray, lowIndex, highIndex);
    }
    return null;
}

static private void insertionSort(int[] array, int lowIndex, int
highIndex) {
    for (int i = lowIndex + 1; i <= highIndex; i++) {
        int key = array[i];
        int j = i - 1;

        while (j >= lowIndex && array[j] > key) {
            array[j + 1] = array[j];
            j--;
        }
```

```
            array[j + 1] = key;
        }
    }


    static private void pivotSort(List<Integer>pivots, List<Integer>
pivotsIndexes, int lowIndex, int highIndex) {
        for (int i = lowIndex; i < highIndex; i++) {
            for (int j = lowIndex; j < highIndex; j++) {
                if (pivots.get(j) > pivots.get(j + 1)) {
                    Collections.swap(pivots, j, j + 1);
                    Collections.swap(pivotsIndexes, j, j + 1);
                }
            }
        }
    }

    // перевірка на наявність в масиві 3 різних чисел
    static boolean divisible(int[] array, int lowIndex, int highIndex) {
        if (highIndex - lowIndex < 3) {
            return false;
        } else {
            List<Integer> differentNumbers= new LinkedList<>();
            for (int i = lowIndex; i < highIndex + 1; i++) {
                if (!differentNumbers.contains(array[i])) {
                    differentNumbers.add(array[i]);
                }
                if (differentNumbers.size() >= 3) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Sort.java

```
public abstract class Sort {
        public int comparisons = 0;
        public abstract void sort(int[] arrayToSort);
        static void swap(int[] array, int a, int b) {
            int temp = array[a];
            array[a] = array[b];
            array[b] = temp;
        }
        public int partition(int[] array, int lowIndex, int highIndex, int
pivot) {
            int leftPointer = lowIndex;
            int rightPointer = highIndex - 1;
            while (leftPointer < rightPointer) {
                while (array[leftPointer] <= pivot && leftPointer <
rightPointer) {
                    //comparisons++;
                    leftPointer++;
                }
                while (array[rightPointer] >= pivot && leftPointer <
rightPointer) {
                    //comparisons++;
```

```
                    rightPointer--;
                }
                swap(array, leftPointer, rightPointer);
            }
            if (array[leftPointer] > array[highIndex]) {
                swap(array, leftPointer, highIndex);
            } else {
                leftPointer = highIndex;
            }
            //comparisons++;
            return leftPointer;
        }
        public int getComparisons() {
            return comparisons;
        }

}
```

Difficulties.java

```
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.concurrent.ThreadLocalRandom;

public class Difficulties {
    public static void GenerateDescendingList(ArrayList < Integer > list) {
        for (int i = list.size (); i > 0; i--) {
            list.set ( list.size () - i, i );
        }
    }

    private static void GenerateAscendingList(ArrayList < Integer > list) {
        for (int i = 0; i < list.size (); i++) {
            list.set ( i, i );
        }
    }

    private static void GenerateRandomList(ArrayList < Integer > list) {
//        for (int i = 0; i < list.size (); i++) {
//            list.set ( i, (int) (Math.random () * 1000) + 1 );
        for (int i=1; i<list.size (); i++){
            list.set ( i, i );
        };
        Collections.shuffle(list);
    }


    public static void main(String[] args) {
        int listSize = 100;
        ArrayList < Integer > list = new ArrayList <Integer> (
Collections.nCopies ( listSize,0 ) );
        System.out.println ("-----RANDOM ARRAY-----");
        GenerateRandomList(list);
        ArrayList<Integer> list2 = new ArrayList<> (list);
        int [] listInt = list.stream ().mapToInt ( i -> i ).toArray ();
        QuickSort sorter = new QuickSort();

        sorter.quickSort (list, 0, list.size () - 1);
```

```
        int comparison1 = sorter.comparisons;

        int comparisons2 = QuickSortMedian.quickSort ( list2,0,list.size ()
- 1 );
        QuickSortThree sorter2 = new QuickSortThree();
        sorter2.sort ( listInt );
        int comparisons3 = sorter2.comparisons;



        System.out.print ("Comparisons counter of simple Quick sort: " +
comparison1 +"\n" );
        System.out.print ("Comparisons counter of median Quick Sort: " +
comparisons2 +"\n" );
        System.out.print ("Comparisons counter of 3Pivot Quick Sort: " +
comparisons3 +"\n" );

        ArrayList < Integer > listAsc = new ArrayList <Integer> (
Collections.nCopies ( listSize,0 ) );
        System.out.println ("-----ASCENDING ARRAY-----");
        GenerateAscendingList ( listAsc );
        ArrayList<Integer> list2Asc = new ArrayList<> (listAsc);
        int [] listIntAsc = listAsc.stream ().mapToInt ( i -> i ).toArray
();
        QuickSort sorterAsc = new QuickSort();

        sorterAsc.quickSort (listAsc, 0, listAsc.size () - 1);
        int comparison1Asc = sorterAsc.comparisons;

        int comparisons2Asc = QuickSortMedian.quickSort (
list2Asc,0,listAsc.size () - 1 );
        QuickSortThree sorter2Asc = new QuickSortThree();
        sorter2Asc.sort ( listIntAsc );
        int comparisons3Asc = sorter2Asc.comparisons;

        System.out.print ("Comparisons counter of simple Quick sort: " +
comparison1Asc +"\n" );
        System.out.print ("Comparisons counter of median Quick Sort: " +
comparisons2Asc +"\n" );
        System.out.print ("Comparisons counter of 3Pivot Quick Sort: " +
comparisons3Asc +"\n" );

        ArrayList < Integer > listDsc = new ArrayList <Integer> (
Collections.nCopies ( listSize,0 ) );
        System.out.println ("-----DESCENDING ARRAY-----");
        GenerateDescendingList ( listDsc );
        ArrayList<Integer> list2Dsc = new ArrayList<> (listDsc);
        int [] listIntDsc = listDsc.stream ().mapToInt ( i -> i ).toArray
();
        QuickSort sorterDsc = new QuickSort();

        sorterDsc.quickSort (listDsc, 0, listDsc.size () - 1);
        int comparison1Dsc = sorterDsc.comparisons;

        int comparisons2Dsc = QuickSortMedian.quickSort (
list2Dsc,0,listDsc.size () - 1 );
        QuickSortThree sorter2Dsc = new QuickSortThree();
        sorter2Dsc.sort ( listIntDsc );
        int comparisons3Dsc = sorter2Dsc.comparisons;

        System.out.print ("Comparisons counter of simple Quick sort: " +
```

```
comparison1Dsc +"\n" );
        System.out.print ("Comparisons counter of median Quick Sort: " +
comparisons2Dsc +"\n" );
        System.out.print ("Comparisons counter of 3Pivot Quick Sort: " +
comparisons3Dsc +"\n" );

    }



}
```

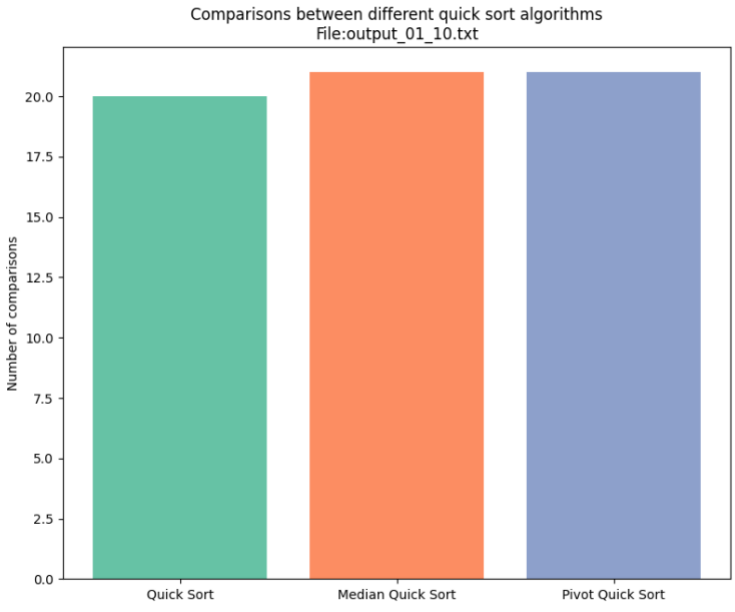
Отож, я розробила три алгоритми – quicksort, quicksort with median, quicksort 3 pivots та зробила дослідження ефективності та визначила різницю між алгоритмами. Отож, запустимо по-перше декілька файлів з консолі за допомогою терміналу:

```
mac@MacBook-Pro-mac task_03_data_examples % java Task input_01_10.txt
```

ip22_andreieva_03_output.txt

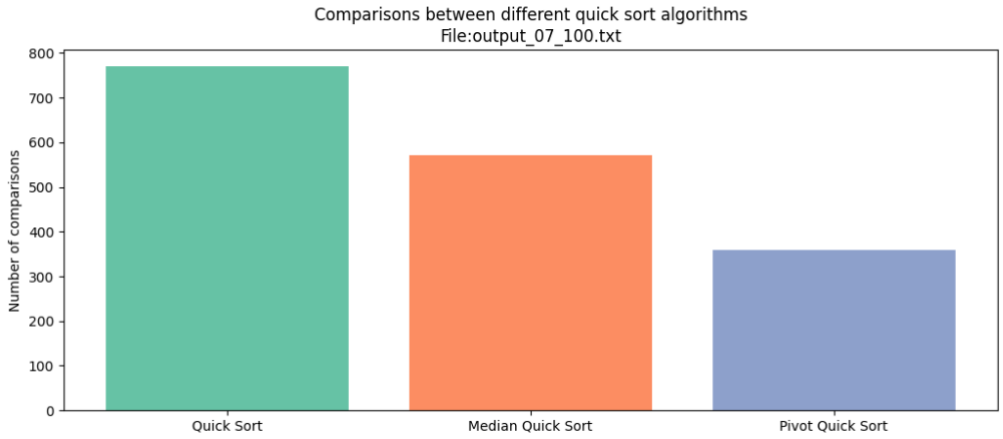
```
20 21 23
```

Побудуємо діаграму за порівняннями

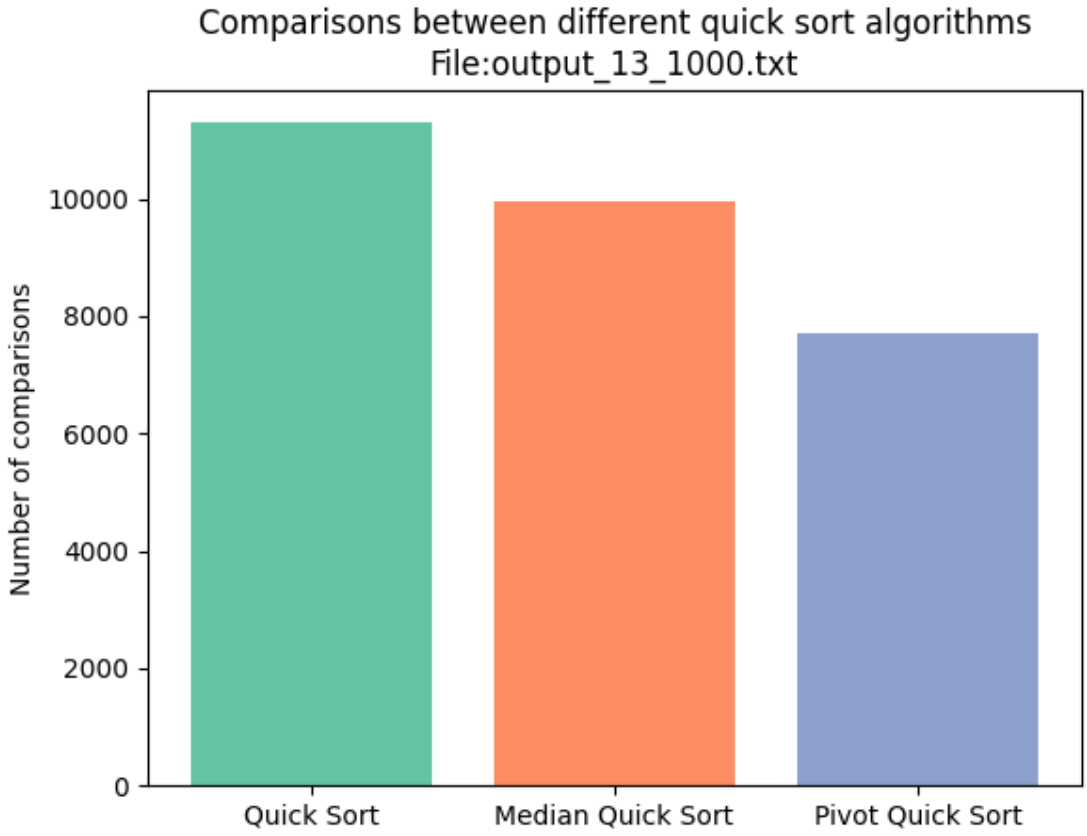


Діаграма порівнянь за заданим файлом(10 елементів)

Приведу приклад ще реалізації декількох файлів на 100 та 1000 елементів, які містяться у файлах(запуск виконується так само з консолі):



Діаграма порівнянь за заданим файлом(100 елементів)



Діаграма порівнянь за заданим файлом(1000 елементів)

Отож за діаграмами можна зробити висновок, що:
Швидкий алгоритм сортування є одним з найшвидших алгоритмів сортування в середньому випадку, але може працювати досить повільно у найгіршому випадку, коли вхідні дані вже відсортовані або майже відсортовані.
Швидкий алгоритм з опорним елементом - медіаною покращує роботу з відсортованими або майже відсортованими вхідними даними. Він

вибирає три елементи - перший, середній та останній елементи, порівнює їх і вибирає медіану як опорний елемент. Це дозволяє запобігти зацикленню на повільному поділі, коли на кожному кроці опорний елемент завжди вибирається максимальним або мінімальним елементом.

Швидкий алгоритм з трьома опорними елементами використовує три опорні елементи замість одного. Він розбиває масив на три частини з опорними елементами, меншими за них та більшими за них. Потім рекурсивно сортує кожну частину. Цей алгоритм може бути ефективнішим, ніж традиційний QuickSort, коли масив містить багато дублікатів.

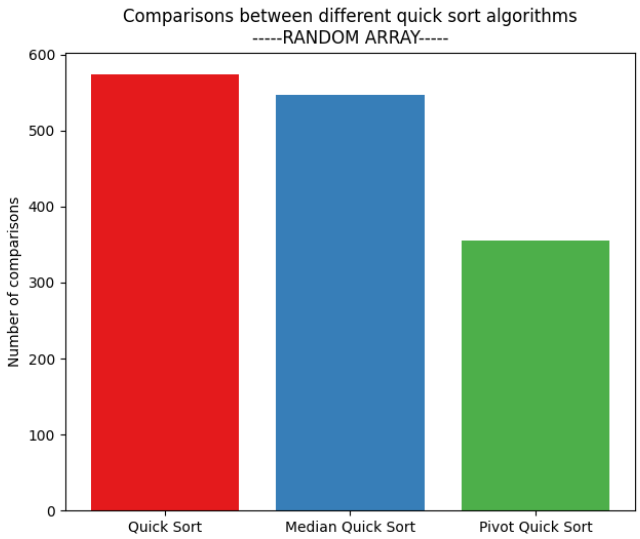
Також додатково я провела дослідження на випадково згенерованому масиві, спадаючому та зростаючому:

Результат

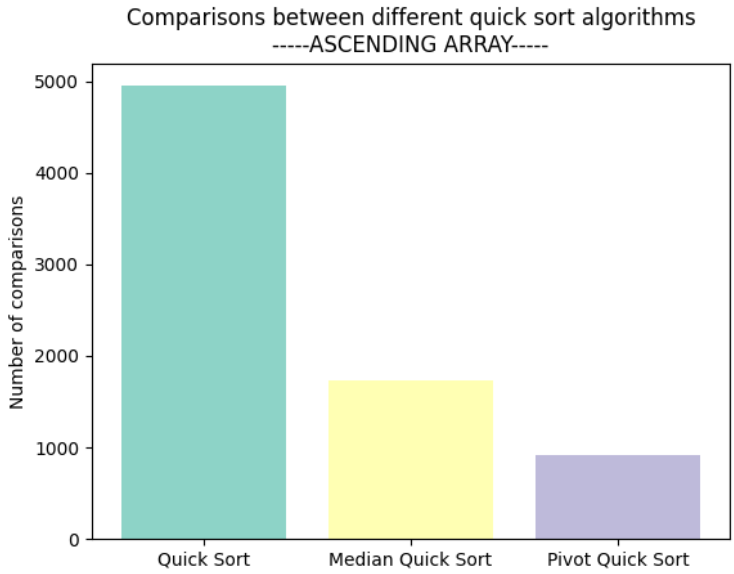
```
/Library/Java/JavaVirtualMachines/jdk-18.0.2.1.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt
 .jar=52053:/Applications/IntelliJ IDEA CE.app/Contents/bin -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath
 /Users/mac/Downloads/Labwork_3/target/classes Difficulties
-----RANDOM ARRAY-----
Comparisons counter of simple Quick sort: 693
Comparisons counter of median Quick Sort: 610
Comparisons counter of 3Pivot Quick Sort: 399
-----ASCENDING ARRAY-----
Comparisons counter of simple Quick sort: 4950
Comparisons counter of median Quick Sort: 1095
Comparisons counter of 3Pivot Quick Sort: 917
-----DESCENDING ARRAY-----
Comparisons counter of simple Quick sort: 4950
Comparisons counter of median Quick Sort: 1789
Comparisons counter of 3Pivot Quick Sort: 1749

Process finished with exit code 0
```

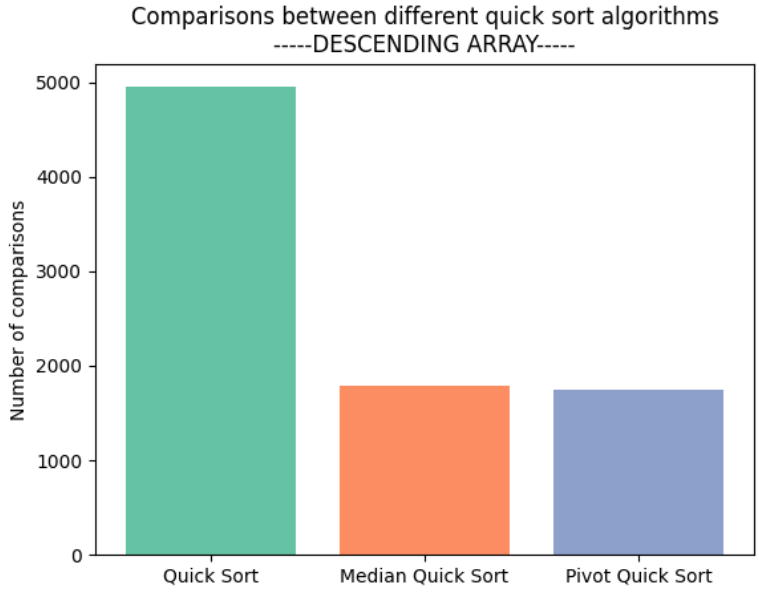
Реалізую це більш зрозуміло на графіках:



Random array (100 елементів)



Ascending array (100 елементів)



Descending array (100 елементів)

Загалом, алгоритм QuickSort з опорним елементом - медіаною і QuickSort з трьома опорними елементами можуть бути ефективнішими за звичайний QuickSort у деяких випадках. Проте, їх ефективність залежить від конкретного набору даних, тому найкраще вибирати алгоритм відповідно до конкретних потреб та властивостей вхідних даних.Хочу відмітити, що алгоритм quicksort є дуже своєрідним, так як відрізняється тим, що найгірший випадок виникає, коли процес розділення завжди вибирає

найбільший або найменший елемент як опору. Якщо ми розглянемо наведену вище стратегію розділення, де останній елемент завжди вибирається як опорний, найгірший випадок станеться, коли масив уже буде відсортовано в порядку зростання або зменшення. Тому ascending та descending будуть найгіршими випадками, а найкращим буде той випадок, коли береться середній елемент як pivot.

Випадок	Час виконання
Найкращий	O(n log n)
Середній	O(n log n)
Найгірший	O(n^2)

У найкращому випадку QuickSort може досягти часу виконання O(n log n), коли розділення масиву на підмасиви відбувається збалансовано, тобто масив ділиться на дві рівні частини.
У середньому випадку час виконання також дорівнює O(n log n), оскільки випадковість даних впливає на балансування підмасивів.
У найгіршому випадку, коли масив вже відсортований або відсортований у зворотному порядку, та якщо кожний раз вибирати елемент в кінці масиву як опорний елемент, час виконання може досягти O(n^2). Однак, зазвичай це виключний випадок, оскільки використання випадкового елемента для розділення зменшує ймовірність його виникнення.

Порівняння Quicksort та MergeSort

	Quicksort	MergeSort
Час роботи	В найгіршому випадку: $\Theta(n^2)$ • В середньому випадку: $\Theta(n \lg n)$	O(n log n), де n - розмір вхідного масиву
Додаткова пам'ять	Не потребує	Потребує
Елегантність	+	-

Метод декомпозиції для алгоритму

• Розділення
• Рекурсивний розв'язок

• Комбінування (не потребує швидке сортування)

Висновок

Отже, я дослідила алгоритм QuickSort та його варіації, розробила програму, дослідила всі три випадки роботи, порівняла варіації між собою та ілюструвала це у вигляді стовпчастих діаграм. Також додатково провела дослідження на основі рандомного, спадаючого та зростаючого масиву й порівняла quicksort з merge sort та зробила свої висновки.