Zeyad Ahmed Ibrahim Zidan 19015709

ALGORITHMS

Programming Assignment #1 Report

I am starting this report with the analysis of problem 1 as well as a graph comparing the 3 methods together.

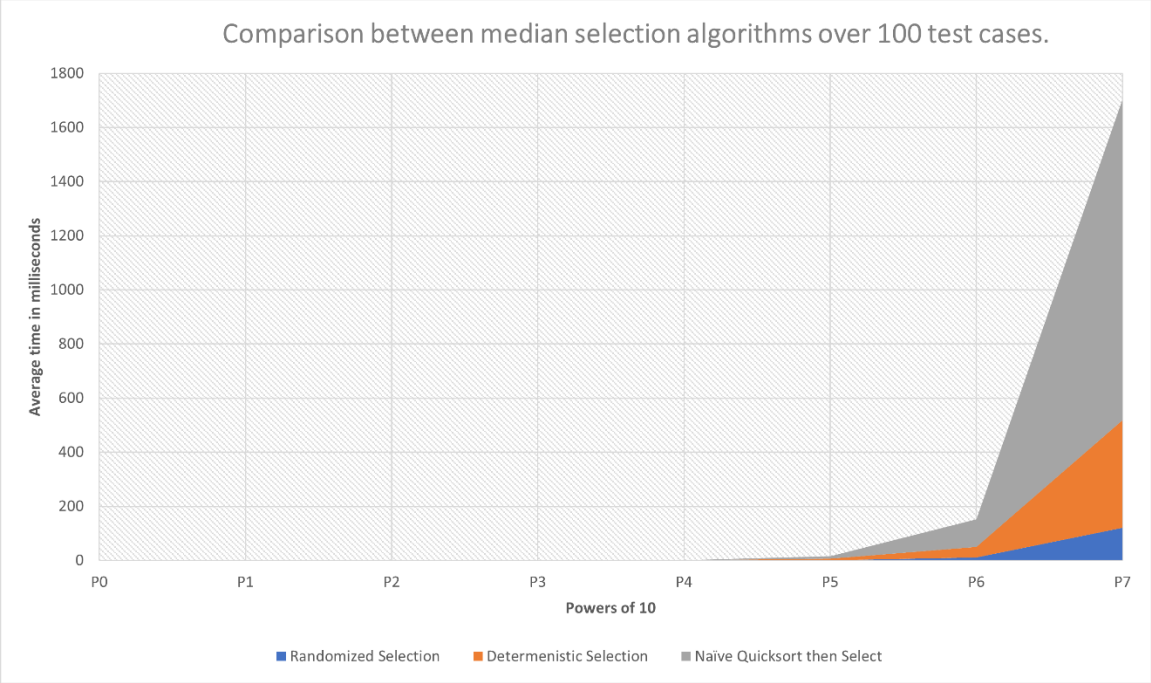
The table below shows the time it takes (in milliseconds) for each method to compute the median in an array of size n to the power P, where P has the values P0 to P7 (*n^7 as stated in the requirements).*

Please note that these results are the average of running 100 test cases on the algorithms.

	P0	P1	P2	P3	P4	P5	P6	P7
Randomized Selection Time (ms)	0	0	0	0	0	1	11	120
Determenistic Selection Time (ms)	0	0	0	0	0	5	39	398
Naïve Quicksort then Select Time (ms)	0	0	0	0	0	9	102	1185

Randomized Quick Selection Algorithm outperforms the **Deterministic Selection Algorithm** and the latter two outperform the **Naïve Quicksort Algorithm**.

The graph below is another representation of the table provided above.



Notes:

- Deterministic Selection is said to be better than Randomized Quicksort Selection and runs in linear time.
- Analysis of the algorithms from the previous section shows us that Randomized Quicksort Selection outperformed the Deterministic Selection, **which proves that Randomized Quicksort Selection is INDEED practically better than Deterministic Selection**.

Now we have come into a conclusion with our analysis, let's go through problems statement, challenges, approaches, and code structure I have used.

PROBLEM STATEMENT

Randomized Quick Selection Algorithm

Challenges:

- I have studied the quicksort algorithm, but as much as it might sound weird, I have never implemented the quicksort algorithm before.
- The algorithm seemed hard to implement at first.
- Random selection of pivots and dealing with each one.

Approaches:

- As a practice I implemented quicksort algorithm in a class. It is not generic, but it suits the course of actions applied on integers.
- After playing around with quicksort and getting quite good knowledge about its mechanisms, randomized quicksort seemed easier a lot.
- The random selection though needed a correct indexing which took me some time to find it.

Data Structures Used:

- Typical Java Arrays
- Nothing special was used in implementing this algorithm.

Class Structure:

- A single class called RSelector.
- The class has a method called rSelect, which takes an array of integers as a parameter and specifying the lower and upper bounds, the method returns the ith smallest number in the array.
- Thanks to **Partitioner** class (will talk about it later), the code is so straight forward.

Pseudocode:

- Randomly select an element to be a pivot **p**.
- Partition around **p** and return the new index of **p (PI)**.
- If **PI** is equal to **i**, return **p**.

- Else if **PI < i**, recursively call the function on the right sub problem.
- Else if **PI > i,** recursively call the function on the left sub problem.

Deterministic Selection Algorithm

Challenges:

- I did not quite understand the algorithm well at first sight. I thought I would prepare a list of medians then just get the median of this array (Median of medians they say).
- Coming to implementation I figured out that this might not be how the algorithm works, so I revisited some lectures and slides to study the algorithm more.
- Indexing was painful to be honest.
- Although I did not understand the algorithm at first, it was easier for me to implement than the randomized selection algorithm.

Approaches:

- Observing and learning from the lectures and slides, I came up with a way or a pseudocode to implement the algorithm.
- Once again, I debugged each line of the code (as happened in randomized quick selection algorithm) to correct the indexing errors.

Data Structures Used:

- Typical Java Arrays
- Nothing special was used in implementing the algorithm.

Class Structure:

- A single class called DSelector.
- The class has a method called dSelect, which takes an integer array as a parameter and by specifying lower and upper bounds, the function can recursively compute the median of the array.
- Thanks again to **Partitioner** class, the code is straight forward.
- The class has another method called selectMedian, which takes the same parameters as dSelect.

Pseudocode:

- Function selectMedian (int[] array, lower, upper) returns int
    - If lower > upper → return array[upper] *// Base Case*
    - Let pivot = dSelect(array, lower, upper)
    - Partition around pivot.
    - If **pivotIndex = array size / 2** → return pivot *// Median found*
    - If **pivotIndex < array size / 2** → recursive call on right sub problem.
    - If **pivotIndex > array size / 2** → recursive call on left sub problem.
- End of Function selectMedian.

- Function dSelect (int[] array, lower, upper) returns int

    - If array size < 5 → Special Handle *// Base Case*
    - Break the array into groups of 5, then sort the array
        - *// Here I have used the naïve quicksort I have implemented as practice.*
    - Establish an array of medians.
    - Recursively call dSelect on the medians array with the new lower and upper bounds.
- End of Function dSelect.

Maximum Side Length Problem

Challenges:

- The problem might seem like just another copy of the closest pair of points problem, but it is not.
- Determining the maximum side length has come to being tricky at the points where I would select the best side length to continue the problem with.
- Another indexing challenge and yes, I know by now I have a problem in indexing.

Approaches:

- Using the min and max functions provided by Java Math Library, selecting, keeping, and determining the best side length was no further a problem.
- Debugging saved another day in a junior programmer life, as I used debugging to solve my indexing problems.

Data Structures Used:

- ArrayList, Comparator, and List provided by Java Utilities Library.
- Point provided by Java AWT Library.

Class Structure:

- A single class called MaxLenSolver.
- 4 methods are implemented in the class.
- Method solve which takes the array of points, prepares it to 2 sorted arrays with respect to X axis and Y axis, and sends them to closestPair method.
- Method closestPair finds the maximum side length between the closest pair of points.
- Method closestSplitPair finds the maximum side length between the closest **SPLIT** pair of points given a delta parameter.
- Method cloneArray clones an ArrayList to be processed.

Pseudocode:

Closest Pair of Points Algorithm

- Given an array of points, P, the following algorithm finds the maximum side length of a square that can be drawn around each point.

- Sort P in 2 arrays with respect to X axis and Y axis into P_x and P_y.
- Base Case (less than 4 elements in the array):
    - Brute force on the elements to get the maximum side length.

- Divide and Conquer Approach:
    - Let P_x be split into L_x and R_x. where L and R are the left and right sub problems to be processed next.
    - Sort Lx in 2 arrays with respect to X axis and Y axis just like P.
    - Repeat for Rx
    - Let leftMaxLen be the value returned by recursively calling the closestPair method on left subproblem (Lx_x, Lx_y).
    - Repeat for the right subproblem → rightMaxLen.
    - Let **DELTA** = min(leftMaxLen, rightMaxLen)
    - Call closestSplitPair on Px and Py and give **DELTA** as a parameter.

Closest Split Pair of Points Algorithm

- Let x dash be the median of Px.
- From Py, copy all the elements that has its x coordinates between (x` +/- **DELTA**) to a new array **spaceY**.
- Since Py is sorted with respect to Y axis, the new array would be sorted as well.
- Let bestDelta = current **DELTA**.
- Brute force on the elements of **spaceY** up to 8 elements only.
- Compute the new bestDelta and return it.

Finding the best delta on the left, right, split planes, compute the best among these 3 and return it using Min(Min(Left, Split), Min(Right, Split)).

CODE STRUCTURE

Classes

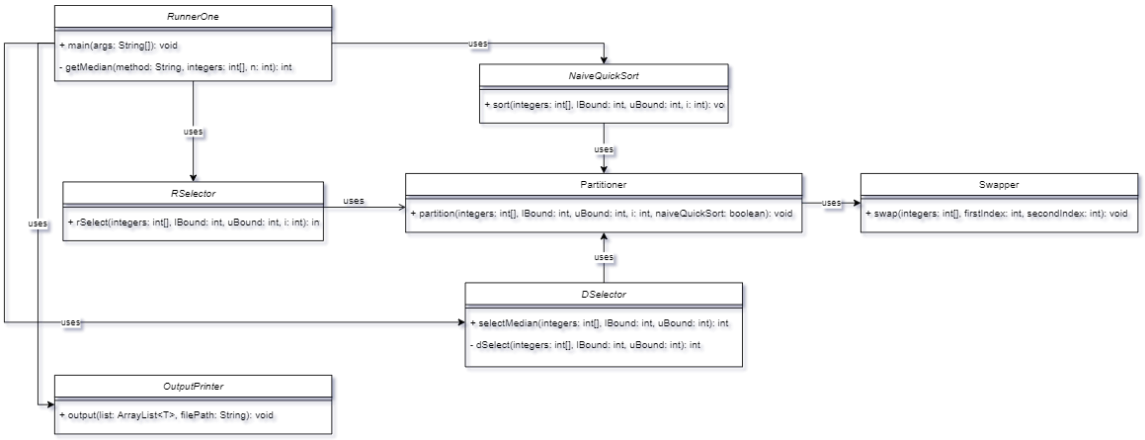
- Main Classes
    - RSelector
    - DSelector
    - NaiveQuickSort
    - MaxLenSolver
- Helper Classes
    - Partitioner → Used to partition around a pivot in the first 3 main classes.
    - Swapper → Used to swap elements in arrays.
    - RunnerOne → Contains main method for problem 1.
    - RunnerTwo → Contains main method for problem 2.
    - OutputPrinter → Prints the outcomes of runners into files.

Design Patterns Used

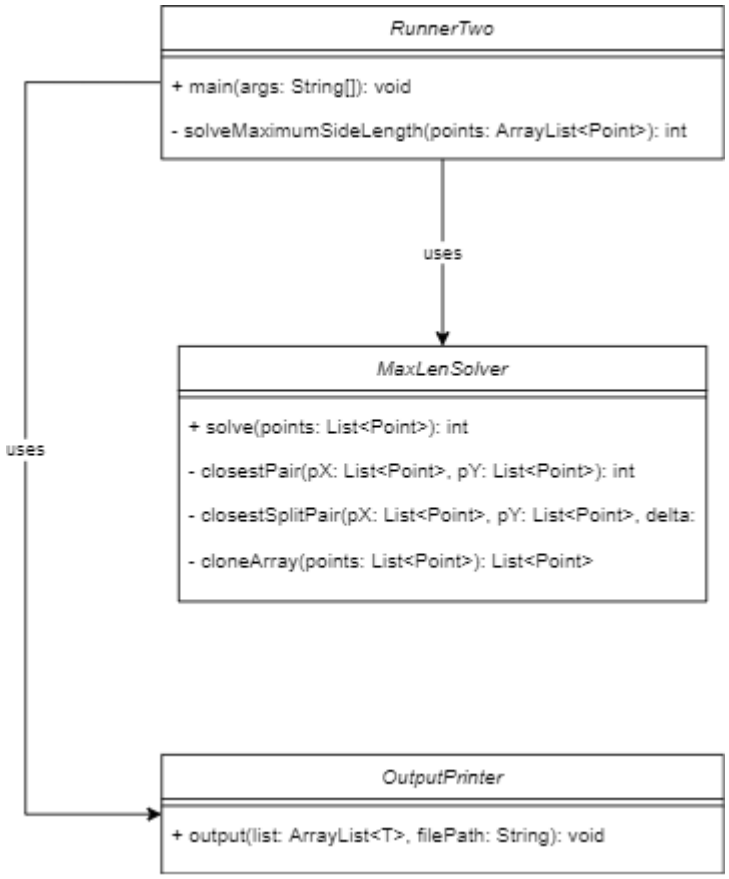
- Used dependency injection which helped refactoring the code into some kind of clean code by removing duplicates and having each class do a dedicated job to another.

UML DIAGRAM

A UML diagram for problem 1 is provided below to describe the relations between the classes used for problem 1 implementation.



Another one is shown below for problem 2 describing the relations between the classes used for problem 2 implementation.



Assignment is reported by student,

Zeyad Ahmed Ibrahim Zidan 19015709

"I acknowledge that I am aware of the academic integrity guidelines of this course, and that I worked on this assignment independently without any unauthorized help."

- Zeyad Zidan, 19015709