CS 3310 Design and Analysis of Algorithms
Project #2
(Total: 100 points)

Student Name: Loc Nguyen

Date: 05/16/2023

Important:
- *Please read this document completely before you start coding.*
- Also, please read the submission instructions (provided at the end of the project description) carefully before submitting the project.

Project #2 Description:

Given a list of n numbers, the Selection Problem is to find the k^{th} smallest element in the list. The first algorithm (Algorithm 1) is the most straightforward one, i.e. to sort the list and then return the k^{th} smallest element. It takes O(n log n) amount time. The second algorithm (Algorithm 2) is to apply the procedure Partition used in Quicksort. The procedure partitions an array so that all elements smaller than some pivot item come before it in the array and all elements larger than that pivot item come after it. The slot at which the pivot item is located is called the pivotposition. We can solve the Selection Problem by partitioning until the pivot item is at the k^{th} slot. We do this by recursively partitioning the left subarray if k is less than pivotposition, and by recursively partitioning the right subarray if k is greater than pivotposition. When k = pivotposition, we're done. The best case complexity of this algorithm is O(n) while the worst case is $O(n^2)$. The third algorithm (Algorithm 3) is to apply the Partition algorithm with the mm rule and it's theoretical worst case complexity is O(n).

Program the following three algorithms that we covered in the class:

- Algorithm 1: find the k^{th} smallest element in the list using the O (n log n) Mergesort sorting method.
- Algorithm 2: find the k^{th} smallest element in the list using partition procedure of Quicksort recursively.
- Algorithm 3: find the k^{th} smallest element in the list using partition procedure of Quicksort recursively via Medians of Medians (mm).

You can use either Java, or C++ for the implementation. The objective of this project is to help student understand how above three algorithms operates and their difference in run-time complexity (average-case scenario). The project will be divided into three phases to help you to accomplish above tasks. They are Part 1: Design and Theoretical analysis, Part 2: Implementation, and Part 3: Comparative Analysis.

After the completion of all three parts, Part 1, Part 2 and Part 3, submit (upload) the following files to Canvas:

- ***three program files of three algorithms or one program file including all three algorithms (your choice of programming language with proper documentation)***
- ***this document in pdf format (complete it with all the*** <Insert > ***answers)***

Part 1: Design & Theoretical Analysis (30 points)

a. Complete the following table for theoretical worst-case complexity of each algorithm. Also need to describe how the worst-case input of each algorithm should be.

Algorithm	theoretical worst-case complexity	describe the worst-case input
Algorithm 1	O(nlogn)	If elements in left and right array are already sorted and have alternate elements, such as [1,3,5,7] or [2,4,6,8], every element will need to be compared at least once.
Algorithm 2	$O(n^2)$	If the pivot creates an unbalanced array because it's the biggest or smallest element, this is the worst-case input.
Algorithm 3	O(nlogn)	If the pivot creates an unbalanced array because it's the biggest or smallest element, this is the worst-case input. However, after the first pick, finding the median is guaranteed and will divide the array "evenly", guaranteeing same performance as merge sort

b. Design the program by providing pseudocode or flowchart for each sorting algorithm.

Algorithm 1:

```
Def mergesort (array a):
    If (n == 1):
        Return a
    arrayOne = a[0] …. A[n/2]
    arrayTwo = a[n/2 +1] … a[n]

    arrayOne = mergesort(arrayOne)
    arrayTwo = mergesort(arrayTwo)

    return merge(arrayOne, arrayTwo)

Def merge (array a, array b)
    Array c = []
    While (a and b have elements)
        If(a[0] > b[0])
            Add b[0] to end of c
            Remove b[0] from b
        Else:
```

```
                    Add a[0] to the end of c
                    Remove a[0] from a


        While (a has elements)
                Add a[0] to end of c
                Remove a[0] from a
        While (b has elements)
                Add b[0] to end of c
                Remove b[0] from b

        Return c

Def findkth(array c, kthSmallest element):
        Return c[kthSmallest element -1]
```

Algorithm 2:

```
Def quicksort(array):
If len(array <=1):
        Return array
Else:
        #pick random element from array as pivot
        Pivot = random.choice(array)
        #sort array based on pivot using Python list comprehension
        leftArr = [x for x in array[1:] if x < pivot]
        rightArr = [x for x in array[1:] if x >= pivot]
        return quicksort(leftArr) + [pivot] + quicksort(rightArr)
```

Algorithm 3:

```
def findMedian(a, p, r):
        L = []
        for i in range(p, r+1):
                L.append(a[i])
                L.sort()
        return L[(r-p+1)//2]
Def quicksortMM(array):
If len(array <=1):
        Return array
Else:
        #pick random element from array as pivot
        Pivot = findMedian(array,0,len(array)-1)
        #sort array based on pivot using Python list comprehension
        leftArr = [x for x in array[1:] if x < pivot]
        rightArr = [x for x in array[1:] if x >= pivot]
        return quicksort(leftArr) + [pivot] + quicksort(rightArr)
```

c. Design the program correctness testing cases. Design at least 10 testing cases to test your program and give the expected output of the program for each case. We prepare for correctness testing of each of the three programs later generated in Part 2.

Testing case #	Input	Expected output	Merge Sort (√ if CORRECT output from your program)	Quicksort (√ if CORRECT output from your program)	Quicksort using Median of Median (√ if CORRECT output from your program)
1	Input Array: [27, 42, 42, 18, 130, 39, 128, 113, 16, 108, 154, 125, 66, 122, 114, 93]	Sorted Array: [16, 18, 27, 39, 42, 42, 66, 93, 108, 113, 114, 122, 125, 128, 130, 154] The kth (k = 2) smallest element: 18	√	√	√
2	Input Array: [64, 151, 147, 47, 80, 129, 6, 115, 132, 84, 110, 103, 109, 79, 106, 53]	Sorted Array: [6, 47, 53, 64, 79, 80, 84, 103, 106, 109, 110, 115, 129, 132, 147, 151] The kth (k = 2) smallest element: 47	√	√	√
3	Input Array: [0, 17, 107, 49, 116, 86, 103, 89, 3, 110, 89, 38, 106, 29, 143, 11]	Sorted Array: [0, 3, 11, 17, 29, 38, 49, 86, 89, 89, 103, 106, 107, 110, 116, 143] The kth (k = 2) smallest element: 3	√	√	√
4	Input Array: [35, 71, 80, 156, 138, 134, 48, 57, 15, 15, 87, 93, 66, 85, 31, 143]	Sorted Array: [15, 15, 31, 35, 48, 57, 66, 71, 80, 85, 87, 93, 134, 138, 143, 156] The kth (k = 2)	√	√	√

		smallest element: 15			
5	Input Array: [51, 130, 83, 98, 109, 10, 86, 101, 136, 58, 81, 53, 153, 51, 149, 103]	Sorted Array: [10, 51, 51, 53, 58, 81, 83, 86, 98, 101, 103, 109, 130, 136, 149, 153] The kth (k = 2) smallest element: 5	√	√	√
6	Input Array: [70, 106, 61, 11, 114, 159, 96, 24, 143, 90, 133, 125, 15, 60, 139, 13]	Sorted Array: [11, 13, 15, 24, 60, 61, 70, 90, 96, 106, 114, 125, 133, 139, 143, 159] The kth (k = 2) smallest element: 13	√	√	√
7	Input Array: [54, 102, 104, 66, 40, 1, 17, 67, 140, 92, 159, 124, 128, 121, 21, 100]	Sorted Array: [1, 17, 21, 40, 54, 66, 67, 92, 100, 102, 104, 121, 124, 128, 140, 159] The kth (k = 2) smallest element: 17	√	√	√
8	Input Array: [156, 125, 125, 25, 117, 123, 143, 115, 71, 142, 152, 43, 59, 37, 91, 132]	Sorted Array: [25, 37, 43, 59, 71, 91, 115, 117, 123, 125, 125, 132, 142, 143, 152, 156] The kth (k = 2) smallest element: 37	√	√	√
9	Input Array: [29, 25, 14, 138, 122, 33, 121, 133, 89, 11, 40, 3, 15, 64, 13, 105]	Sorted Array: [3, 11, 13, 14, 15, 25, 29, 33, 40, 64, 89, 105, 121, 122, 133, 138] The kth (k = 2) smallest element: 11	√	√	√
10	Input Array: [133, 12, 41, 32, 95, 50,	Sorted Array: [2, 6, 7, 12, 17, 32,	√	√	√

	144, 146, 6, 33, 17, 117, 149, 2, 101, 7]	33, 41, 50, 95, 101, 117, 133, 144, 146, 149] The kth (k = 2) smallest element: 6			

d. Design testing strategy for the programs. Discuss about how to generate and structure the randomly generated inputs for experimental study in Part 3.
Hint 1: The project will stop at the largest input size n that your computer environment can handle. It is the easiest to use a random generator to help generate numbers for the input data sets. However, student should store all data sets and use the same data sets to compare the performance of all three Matric Multiplication algorithms.
Hint 2: Note that even when running the same data set for the same Selection k^{th} program multiple times, it's common that they have different run times as workloads of your computer could be very different at different moments. So it is desirable to run each data set multiple times and get the average run time to reflect its performance. The average run time of each input data set can be calculated after an experiment is conducted in m trails; but the result should exclude the best and worst run. Let X denotes the set which contains the m run times of the m trails, where $X = \{x_1, x_2, x_3 \ldots x_m\}$ and each x_i is the run time of the i^{th} trial. Let x_w be the largest time (worst case) and x_b be the smallest time (best case). Then we have

$$Average\ Run\ Time = \frac{\sum_{i=1}^{m} Xi - Xw - Xb}{m-2}$$

The student should think about and decide how many trials (the value of m) you will use in your experiment. Note that the common choice of the m value is at least 10.

<Insert answers here – on

1. How you generate and structure the randomly generated inputs?
   Using a random array value generator
2. What value of *m* you plan to use?
   I'm planning to use m of 10

Part 2: Implementation (35 points)

a. Code each program based on the design (pseudocode or flow chart) given in Part 1(b).

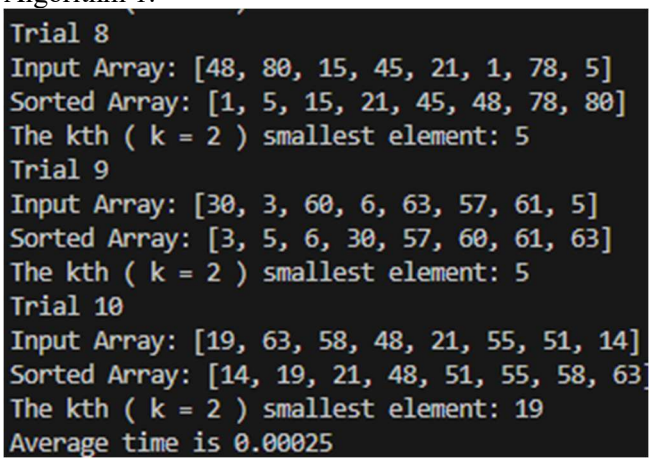
<Generate three programs with proper documentation and store them in three files. Note: They are required to be submitted to Canvas as described in the submission instructions>
<No insert here>

b. Test you program using the designed testing input data given in the table in Part 1(c), Make sure each program generates the correct answer by marking a "√" if it is correct for each testing case for each program column in the table. Repeat the process of debugging if necessary.

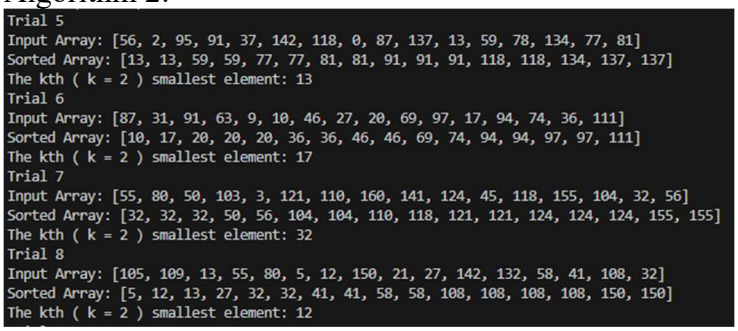
<Complete the testing with testing cases in the table @Part 1(c)>
<No insert here>

c. For each program, capture a screen shot of the execution (Compile&Run) using one testing case to show how this program works properly

Algorithm 1:



Algorithm 2:



Algorithm 3:

```
Trial 5
Input Array: [105, 124, 120, 149, 4, 59, 96, 104, 159, 94, 121, 61, 48, 85, 44, 7]
Sorted Array: [7, 44, 44, 59, 61, 85, 85, 94, 96, 96, 104, 104, 121, 121, 149, 159]
The kth ( k = 2 ) smallest element: 44
Trial 6
Input Array: [131, 73, 107, 107, 138, 40, 28, 95, 146, 106, 106, 136, 37, 37, 23, 56]
Sorted Array: [23, 37, 37, 37, 40, 56, 56, 95, 106, 106, 106, 107, 136, 138, 146, 146]
The kth ( k = 2 ) smallest element: 37
```

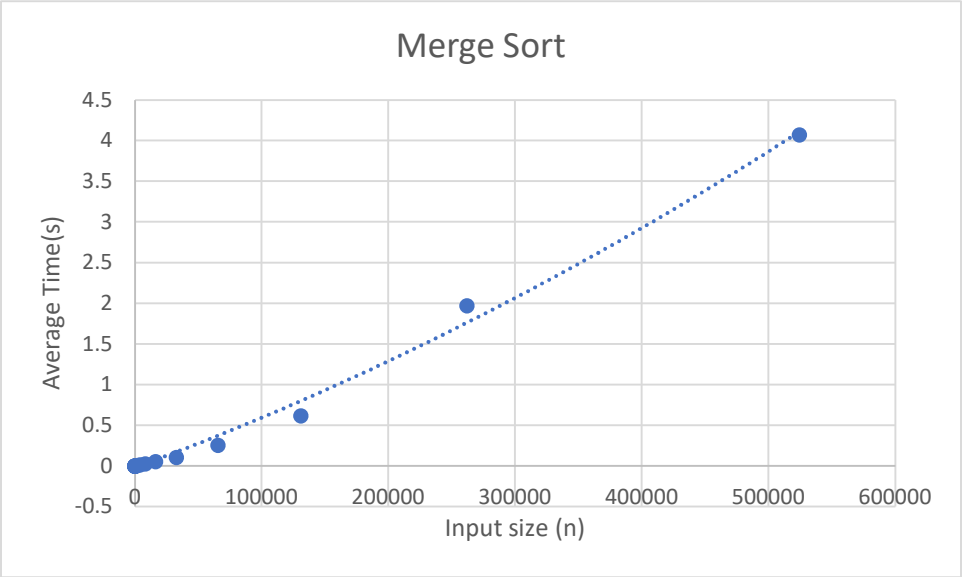
By now, three working programs for the three algorithms are created and ready for experimental study in the next part, Part 3.

Part 3: Comparative Analysis (35 points)

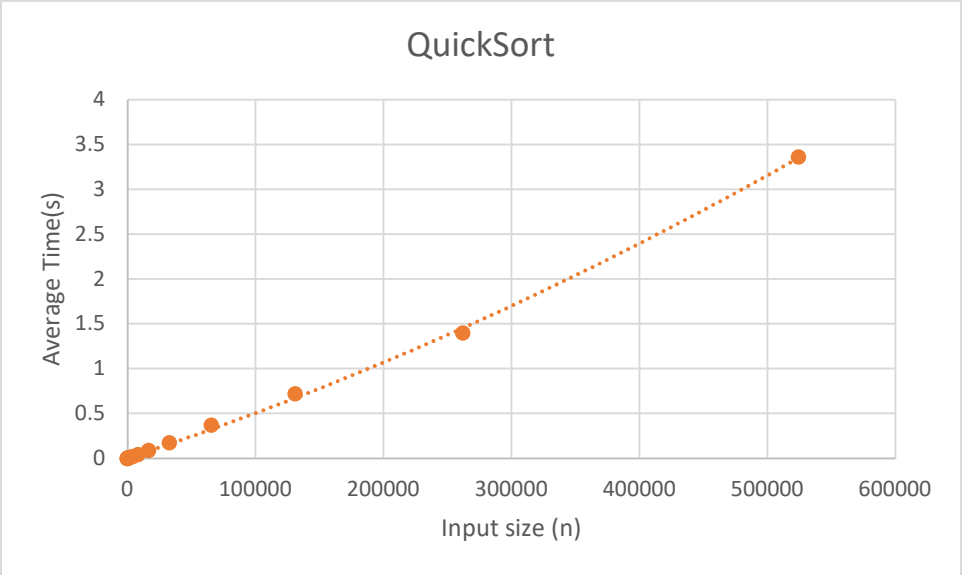
a. Run each program with the designed randomly generated input data given in Part 1(d). Generate a table for all the experimental results as follows.

Input Size n	Average time (Merge Sort)	Average Time (Quick Sort)	Average Time (Quicksort with Median of Median)
2	0.00000	0.00000	0.00000
4	0.00012	0.00000	0.00000
8	0.00025	0.00000	0.00025
16	0.00025	0.00000	0.00025
32	0.00025	0.00000	0.00038
64	0.00037	0.00000	0.00062
128	0.00050	0.00105	0.00087
256	0.00125	0.00103	0.00117
512	0.00238	0.00105	0.00148
1024	0.00268	0.00409	0.00540
2048	0.00574	0.01135	0.00693
4096	0.01586	0.01939	0.01345
8192	0.02678	0.04135	0.02939
2^14	0.05364	0.08621	0.05667
2^15	0.10669	0.17375	0.14618
2^16	0.25350	0.36834	0.31426
2^17	0.61632	0.72103	0.68729
2^18	1.97183	1.39872	1.47073
2^19	4.07044	3.36156	3.30185

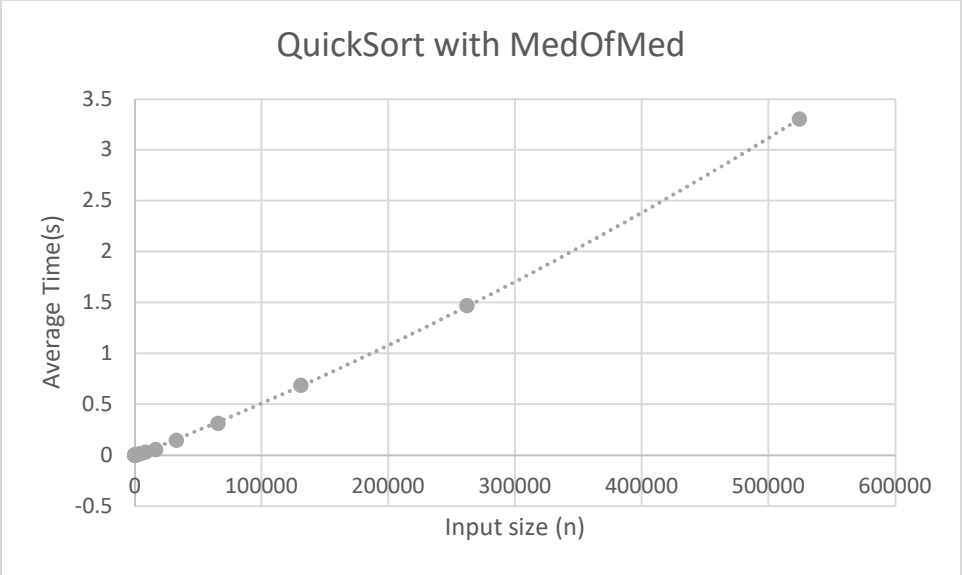
b. Plot a graph of each algorithm and summarize the performance of each algorithm based on its own graph.



Merge sort has quasilinear time and has baseline execution time

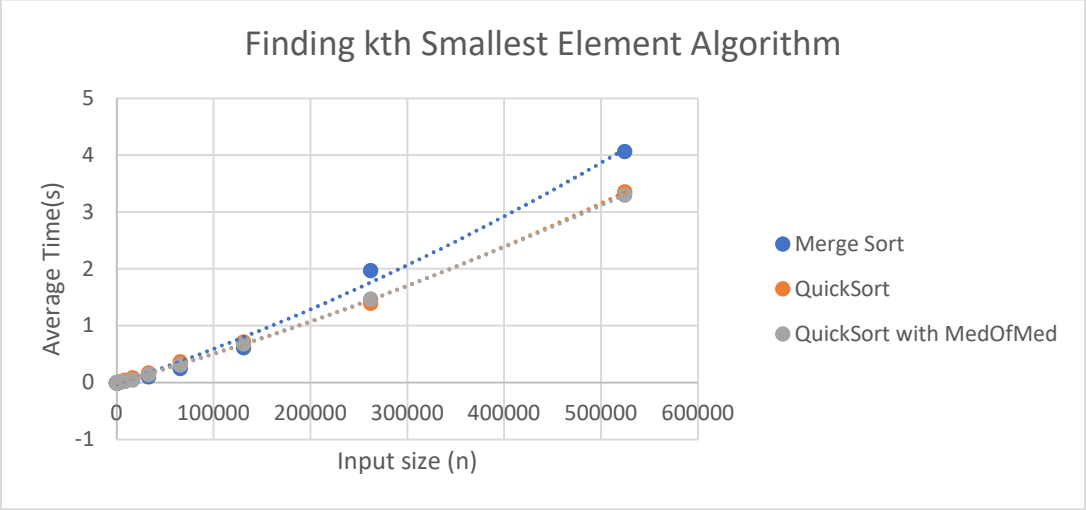


Quicksort has worst-case quadratic time and slowest execution time.



Quicksort using Median of Median optimizes quicksort and have worst case of O(nlogn), mimicking merge sort performance

Plot all three graphs on the same graph and compare the performance of all three algorithms. Explain the reasons for which algorithm is faster than others.



Merge sort is the slowest due to a lot of overhead. Quick sort standalone is slower than quicksort using median of median

c. Compare the theoretical results in Part 1(a) and empirical results here. Explain the possible factors that cause the difference.

Though time complexity application shows that merge sort is slower, in theory, it should be a baseline in the worst case. Quicksort using median of median is the fastest algorithm out of the three analyzed.

d. Give a spec of your computing environment, e.g. computer model, OS, hardware/software info, processor model and speed, memory size, ...

OS Name Microsoft Windows 10 Pro
AMD Ryzen 5 3600 6-Core Processor, 3593 Mhz, 6 Core(s), 12 Logical Processor(s)
Installed Physical Memory (RAM) 24.0 GB

e. Conclude your report with the strength and constraints of your work. At least 200 words. Note: It is reflection of this project. Ask yourself if you have a chance to re-do this project again, what you will do differently (e.g. your computing environment, hardware/software, programing language, data structure, data set generation, …) in order to design a better performance evaluation experiment.

During testing, I have to continually refine the output process from having to print out the array for double checking and streamlining it later by printing only the trial number the machine is running and the average time. The difference is noticeable because the processor don't have to spend precious memory outputting huge display of array 10 times which slow down and bias the true computational time of the algorithms. I would like to do this project in another language such as Java or C++ where they both might have faster execution time than Python, an interpreted language vs Java compilation.

The data set generation changes throughout the development process. At first, data were hardcoded into arrays when array size is small (2) but as the size grows toward 64 or 512, I decided to use a random number generator module in Python to generate a random number from 0-9 as a value for the array to speed up the process. I utilize functions over data structure in this project because it was simpler, but I would like to try approaching it from an object-oriented programming perspective. Finally, the timing device must be properly placed within the code so that it can accurately capture the true performance of the algorithm, ignoring execution of function definition.