**CS 3310 Design and Analysis of Algorithms**

**Project #2**

(Total: 100 points)

**Student Name: \_\_\_\_John Dang\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date: \_05/06/2023\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

***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 kth 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 kth 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 kth 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(n2). 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 kth smallest element in the list using the O (n log n) Mergesort sorting method.
* Algorithm 2: find the kth smallest element in the list using partition procedure of Quicksort recursively.
* Algorithm 3: find the kth 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)**

* 1. 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) | Worse case would be when the order of the array is sorted in an alternating order |
| Algorithm 2 | O(n^2) | Worse case is when the largest or smallest value is chosen as the pivot or when it’s reversely sorted |
| Algorithm 3 | O(n) | We need to make sure that the pivot v is chosen such that at least some fraction of the elements will be smaller than v and at least some other fraction of elements will be greater than v |

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

Algorithm 1

MS(left, right) {

If(left < right) {

Mid = (left + right) / 2;

MS(left, mid)

MS(mid+1, right)

Merge(left, mid, right);

}

}

Algorithm 2

Quicksort(A, n, k) {

M = 1;

J = n;

For loop() {

Partition(m,j,pivotposition);

If(k = pivotposition) {

Return A(k);

} else if(k < pivotposition) {

J = pivotposition – 1;

} else {

M = pivotposition + 1;

k = k – pivotposition;

}

}

}

Algorithm 3

MediansOfMedians(A,n,k) {

Begin:

If(n <= r) {

Sort A and return the kth element;

}

Divide A into floor(n/r) subset of size r each, ignore excess elements

Let M = {m1, m2, …, m(n/r)} be the set of medians in the floor(n/r). subsets

V = Select2(M, floor(n/r), ceiling(floor(n/r) / 2));

Use “Partition” to partition A using v as the pivot;

If(k = pivotposition) {

Return (v);

} else if(k < pivotposition) {

Let S be the set of elements

A(1,…,pivotposition-1),

Return MeanOfMedians(S,pivotposition-1,k);

} else {

Let R be the set of element

A(pivotposition+1,…,n),

Return MediansOfMedians(R,n-pivotposition,k-pivotposition);

}

}

}

* 1. 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 | Algorithm 1  (√ if CORRECT output from your program) | Algorithm 2  (√ if CORRECT output from your program) | Algorithm 3  (√ if CORRECT output from your program) |
| 1 | A = {10, 2, 5, 6 }  Kth = 1 | MS: 2  QS & MM: 5 | √ | √ | √ |
| 2 | A = {4, 10, 3, 9 }  Kth = 1 | MS: 10  QS & MM: 2 | √ | √ | √ |
| 3 | A = {3, 2, 6, 1}  Kth = 1 | MS: 1  QS & MM: 2 | √ | √ | √ |
| 4 | A = {3, 1, 2, 0 }  Kth = 1 | MS: 0  QS & MM: 1 | √ | √ | √ |
| 5 | A = {10, 4, 2, 1}  Kth = 1 | MS: 1  QS & MM: 2 | √ | √ | √ |
| 6 | A ={ 20, 23, 11, 1}  Kth = 1 | MS: 1  QS & MM: 11 | √ | √ | √ |
| 7 | A = {4, 50, 12, 16}  Kth = 1 | MS: 4  QS & MM: 12 | √ | √ | √ |
| 8 | A = {5, 16, 14, 11}  Kth = 1 | MS: 5  QS & MM: 11 | √ | √ | √ |
| 9 | A = {45, 20, 1, 2}  Kth = 1 | MS: 1  QS & MM: 2 | √ | √ | √ |
| 10 | A = {52,46,11,10}  Kth = 1 | MS: 10  QS & MM: 11 | √ | √ |  |

* 1. 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 kth 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 = {x1, x2, x3 … xm} and each xi is the run time of the ith trial. Let xw be the largest time (worst case) and xb be the smallest time (best case). Then we have  
  
 Average Run Time =*

*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?

I will create a for loop with length M which is the size of the array and have an array that will initialize itself with a random number in the loop. Then, it will use the array to run the algorithms M times and record the total time and average time for all M cases.

2. What value of *m* you plan to use? >

**I will use the value of 10**

**Part 2: Implementation (35 points)**

1. 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>**

1. 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>**

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

<Insert - totally three screen shots, one for each program, here>

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)**

1. 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  (Algorithm 1)  Kth = 3 | Average Time  (Algorithm 2)  Kth = 3 | Average Tine  (Algorithm 3)  Kth = 3 |
| 10 | 3458 | 858 | 19625 |
| 100 | 9496 | 2621 | 19348 |
| 1000 | 53359 | 6374 | 23728 |
| 10000 | 227963 | 11745 | 290639 |
| 25000 | 599547 | 24786 | 1625469 |

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

<Insert - totally three graphs, one for each program, here>

<Insert - write a summary>

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.

<Insert - three-graphs-in-one graph here>

<Insert - about explaining the results>

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

<Insert - report the findings and explain>

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

<Insert - spec of your computing environment>

1. 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.

<Insert - write a conclusion about strength and constraints of your work here.>