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

Student Name: Roy Afaryan

Date: 5/18/2023

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(n log n)	An array sorted in **reverse** order.
Algorithm 2	$O(n^2)$	A sorted array.
Algorithm 3	$O(n^2)$	An array that would result in imbalanced partitions when choosing pivots.

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

Algorithm 1 Pseudocode:

```
function mergesort(array arr)

    # base case
    if arr length <= 1
        return arr
    # recursive call
    else
        mid = (left + right) / 2
        call MergeSort (left, mid)
        call MergeSort (mid+1, right)
        call Merge (left, mid, right)

function merge(array left, array right):

    array mergedArr

    while a and b are not empty
        if left[0] > right[0]
            append right[0] to the end of mergedArr
            remove right[0] from right
        else
            append left[0] to the end of mergedArr
            remove left[0] from left

    return mergedArr
```

Algorithm 2 Pseudocode:

```
function quicksort(arrary arr, int left, int right)

    if left < right
        pivot_position = call partition(arr, left, right)
        call quicksort(arr, left, pivot_position - 1)
        call quicksort(arr, pivot_position + 1, right)

function partition(array arr, int left, int right)

    # pivot is leftmost value
    pivot_value = arr[left]

    # initialize i and j values
    i = left + 1
    j = left + 1

    for j to right
        if arr[j] < pivot_value
            swap arr[i] and arr[j]
            i++

    pivot_position = i - 1
    swap arr[left] and arr[pivot_position]

    return pivot_position
```

Algorithm 3 Pseudocode:

```
function quicksort(arrary arr, int left, int right)

    if left < right
        pivot_position = call partition(arr, left, right)
        call quicksort(arr, left, pivot_position - 1)
        call quicksort(arr, pivot_position + 1, right)

function partition(array arr, int left, int right)

    # pivot is Median of Medians
    pivot_value = Median of Medians
    pivot_position = index of pivot value

    # Median of Medians is now pivot and leftmost value
    swap arr[pivot_position] and arr[left]

    # initialize i and j values
    i = left + 1
    j = left + 1

    for j to right
        if arr[j] < pivot_value
            swap arr[i] and arr[j]
            i++

    pivot_position = i - 1
    swap arr[left] and arr[pivot_position]

    return pivot_position
```

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	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	[60, 15, 50, 27, 12, 29, 80, 61]	27 is the 3rd smallest value in the sorted list.	√	√	√
2	[13, 45, 36, 78, 82]	45 is the 3rd smallest value in the sorted list.	√	√	√
3	[41, 54, 69, 71]	41 is the smallest value in the sorted list.	√	√	√
4	[55, 1, 53]	55 is the 3rd smallest value in the sorted list.	√	√	√
5	[95, 97, 50, 100, 23, 93, 93]	50 is the 2nd smallest value in the sorted list.	√	√	√
6	[67, 33, 20, 11, 44, 67, 51, 95, 33]	44 is the 5th smallest value in the sorted list.	√	√	√
7	[34, 11]	11 is the smallest value in the sorted list.	√	√	√
8	[97, 99, 88, 74, 88, 31, 82, 12, 26, 69, 100, 29, 28, 48, 7]	12 is the 2nd smallest value in the sorted list.	√	√	√
9	[83, 58, 55, 78, 60, 21, 42, 67, 14, 84, 29, 18, 28, 8, 84, 60, 69, 82, 62, 58, 40, 35]	8 is the smallest value in the sorted list.	√	√	√

10	[71, 62, 53, 34, 59, 21, 72, 53, 95, 71]	53 is the 4th smallest value in the sorted list.	√	√	√

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 $X = \{x_1, x_2, x_3 \dots x_m\}$ 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?

   I generated and structured the randomly generated inputs using the random library in python. Here is the example code of random input generation:

```
# Generate a random array of size 'size' with values between 1 and 100
arr1 = [random.randint(1, 100) for _ in range(size)]
arr2 = arr1.copy()
arr3 = arr1.copy()
```

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

   I plan on using an m value of 10.

Part 2: Implementation (35 points)

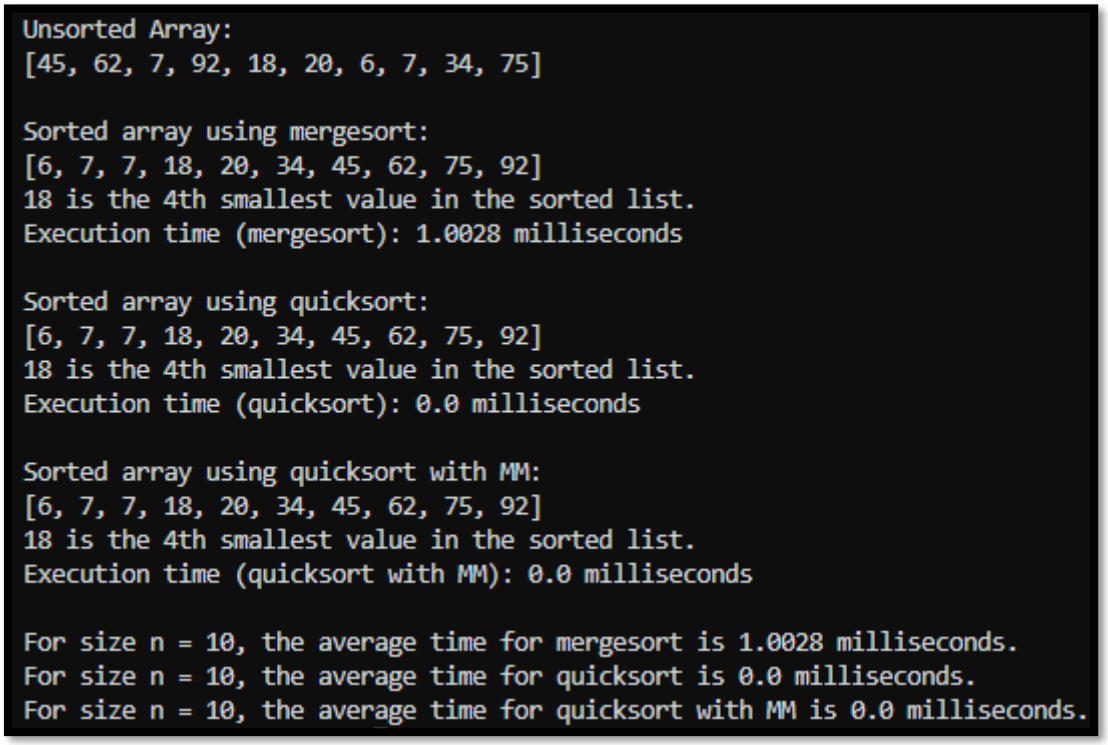
a. Code each program based on the design (pseudocode or flow chart) given in Part 1(b).

<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



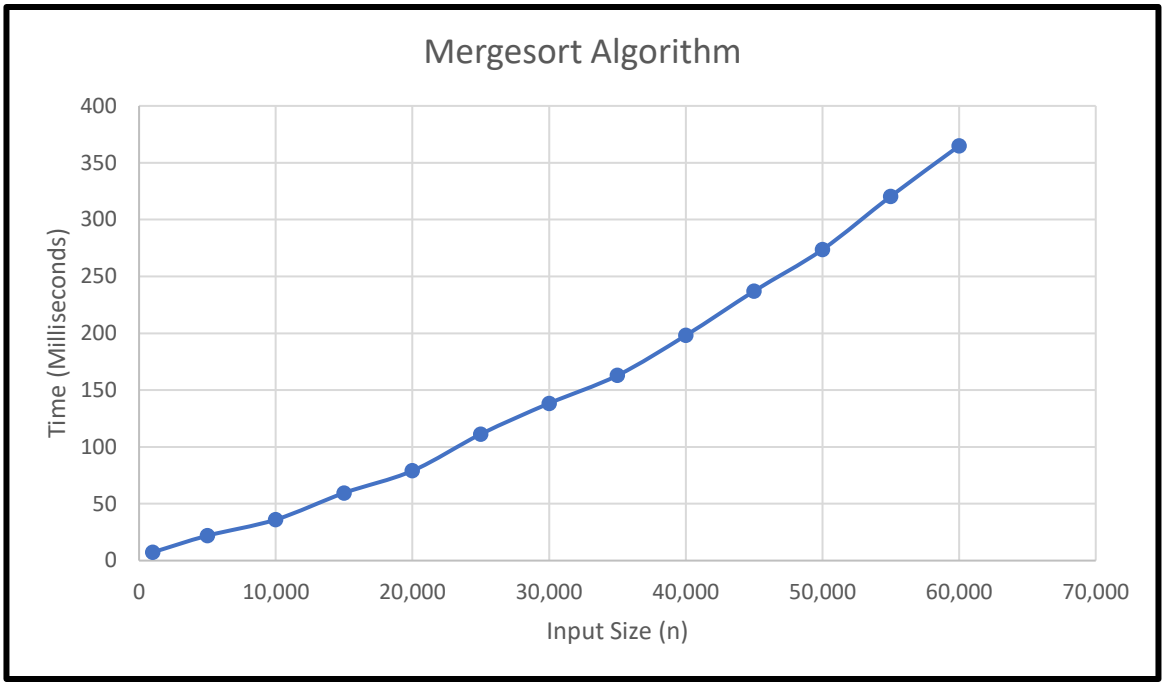
By now, three working programs for the three algorithms are created and ready for experimental study in the next part, Part 3.

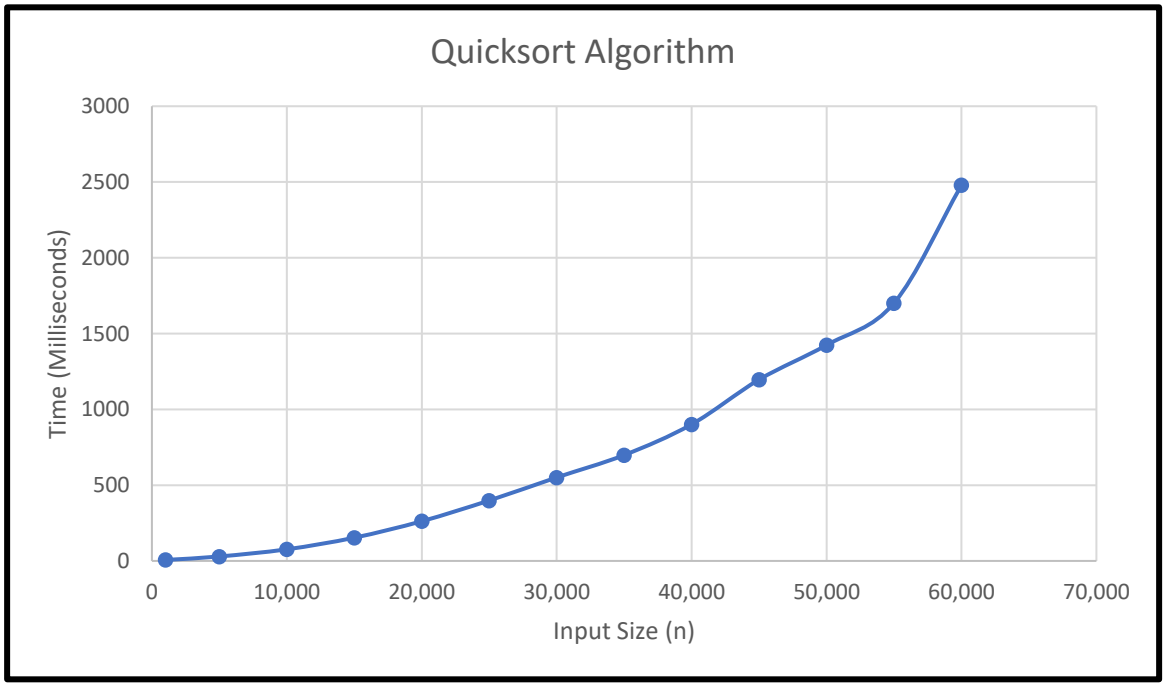
Part 3: Comparative Analysis (35 points)

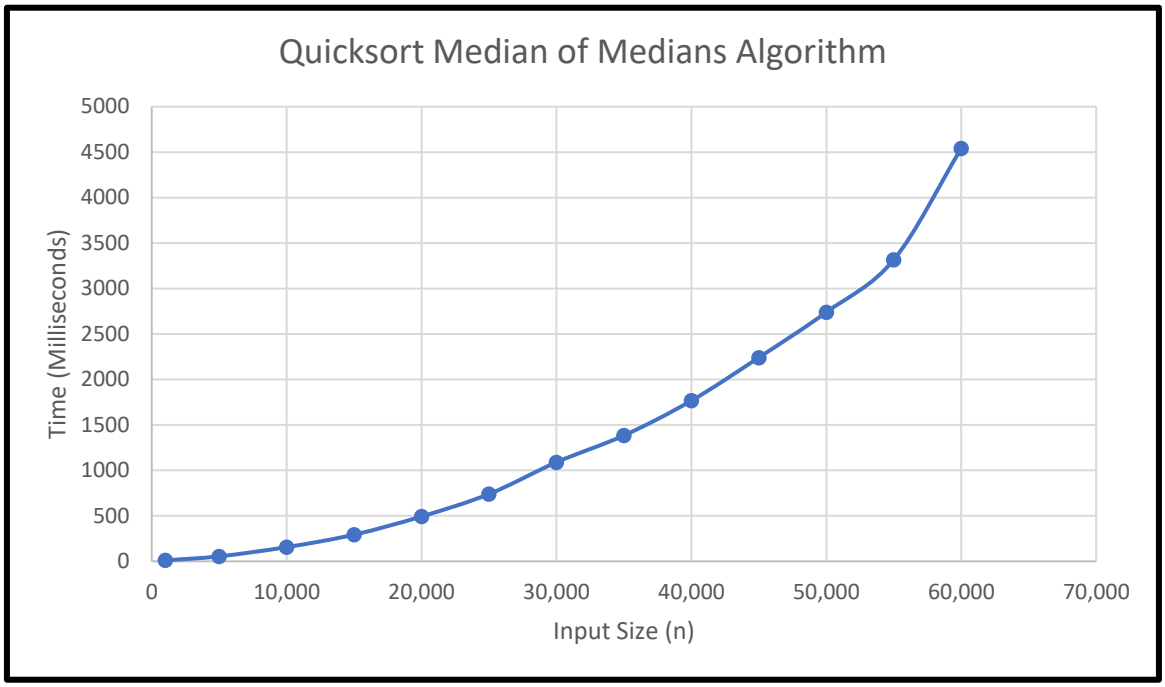
a. Run each program with the designed randomly generated input data given in Part 1(d). Generate a table for all the experimental results as follows.

Input Size n	Average time (Algorithm 1) (milliseconds)	Average Time (Algorithm 2) (milliseconds)	Average Tine (Algorithm 3) (milliseconds)
1,000	7.3	6.5	10.2
5,000	21.91	29.29	54.63
10,000	36.06	76.58	156.14
15,000	59.49	153.34	293.30
20,000	78.96	261.94	493.22
25,000	111.23	398.71	740.15
30,000	138.32	550.82	1089.56
35,000	162.88	698.51	1384.45
40,000	198.05	901.06	1766.70
45,000	237.05	1196.23	2240.93
50,000	273.58	1424.19	2739.12
55,000	320.48	1701.02	3316.84
60,000	364.78	2479.09	4540.65

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

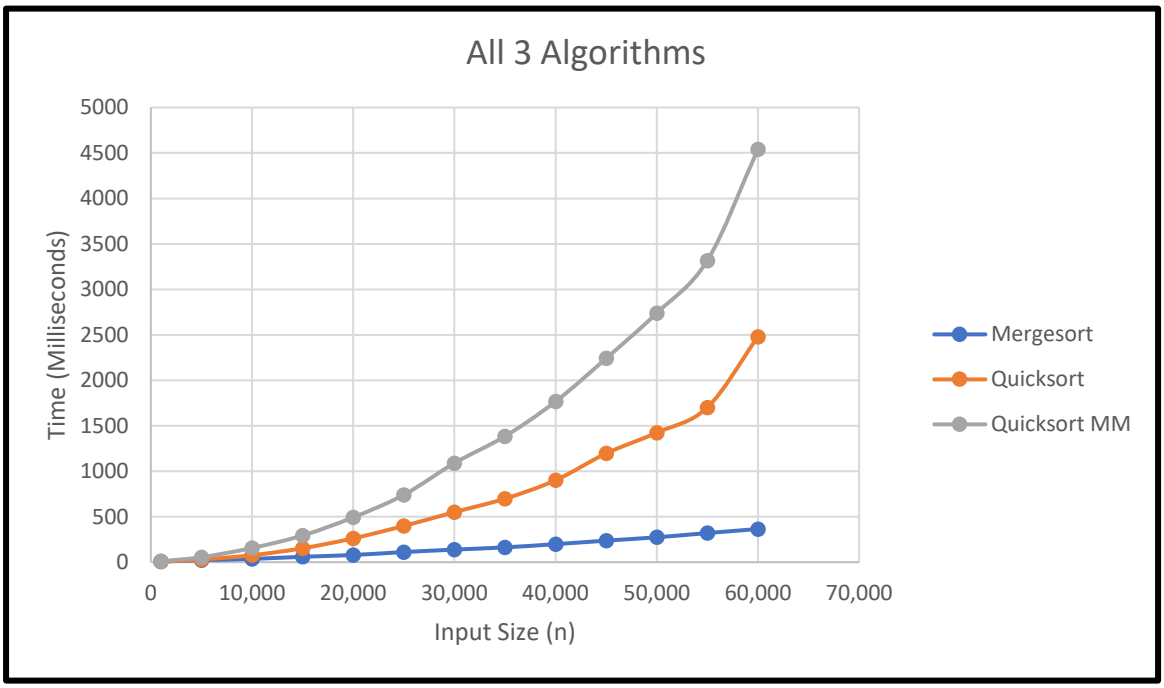






1. Mergesort – Slow growth rate that appears to match the theoretical worst-time complexity of O(n log n)
2. Quicksort – Faster growth than mergesort. Appears to match the theoretical worst-time complexity of $O(n^2)$
3. Quicksort – Faster growth than Quicksort. Appears to match the theoretical worst-time complexity of $O(n^2)$

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.



In terms of efficiency, the order of growth from slowest to fastest is as follows: Mergesort, Quicksort, and Quicksort MM. According to theoretical analysis, although Quicksort and Quicksort MM technically have the same worst-case time complexity, Quicksort MM should be growing slower or at least near the same rate. In this graph, it is very clear that Quicksort MM grows noticeably quicker than regular Quicksort.

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

As mentioned earlier, Quicksort MM grew the fastest despite the theoretical analysis saying it should grow relatively close to Quicksort. I believe a potential reason for these differences is inconsistent medians. To elaborate further, perhaps choosing the first element as the pivot is more consistent in producing balanced array splits.

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

Item	Value
OS Name	Microsoft Windows 10 Home
Version	10.0.19045 Build 19045
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-59B9J6V
System Manufacturer	Micro-Star International Co., Ltd.
System Model	MS-7B61
System Type	x64-based PC
System SKU	Default string
Processor	Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz, 3192 Mhz, 6 Core(s), 12 Logical Pr...
BIOS Version/Date	American Megatrends Inc. 1.30, 3/7/2018
SMBIOS Version	2.8
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Micro-Star International Co., Ltd.
BaseBoard Product	Z370 GAMING PLUS (MS-7B61)
BaseBoard Version	1.0
Platform Role	Desktop
Secure Boot State	Off
PCR7 Configuration	Binding Not Possible
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume4
Locale	United States
Hardware Abstraction Layer	Version = "10.0.19041.2728"
User Name	DESKTOP-59B9J6V\Roy
Time Zone	Pacific Daylight Time
Installed Physical Memory (RAM)	16.0 GB
Total Physical Memory	15.9 GB
Available Physical Memory	3.83 GB
Total Virtual Memory	25.4 GB
Available Virtual Memory	4.26 GB
Page File Space	9.50 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.

Completion of this project has exponentially increased my understanding of algorithms entirely. As mentioned in the previous project, one of my weakest links is divide-and-conquer algorithms due to their use of recursion. After designing and implementing these algorithms, I feel that I have a much better understanding of the concept of recursion. Another great takeaway from this project is the formation of testing parameters. I feel like the testing and debugging phase of this project has developed important skills that will help me succeed in an actual work environment. Looking back to project 1, I mentioned that a previous constraint I had was my choice in system. This time, I chose a much better system to generate results of unsorted list sizes of bigger n values; this allowed me to have a much better analysis. If I could do this project again, I would want to compare the recursive quicksort algorithm with an iterative quicksort algorithm to see the difference between loops and recursion in the sorting process. Overall, the concepts I learned throughout the entirety of this project have given me a deeper understanding of algorithms and the analysis of them, which ultimately are very key takeaways in my academic career.