2019-2020 SPRING CS202 HW1

Rüzgar Ayan 21801984 Section 2

Question 1:

a)

For the function $f(n) = 20n^4 + 20n^3 + 5$, we want to find c and n_0 values that makes its upper bound $O(n^5)$. Then,

$$20n^4 + 20n^3 + 5 \leq cn^5 \qquad for\ n \geq n_0$$

Divide both sides by n^5,

$$\frac{20}{n} + \frac{20}{n^2} + \frac{5}{n^5} \leq c \qquad for\ n \geq n_0$$

As n increases, the left hand side will always decrease. Then, taking $n_0 = 1$,

$$\frac{20}{n} + \frac{20}{n^2} + \frac{5}{n^5} \leq \frac{20}{1} + \frac{20}{1^2} + \frac{5}{1^5} = 45 \leq c \qquad for\ n > 1$$

Then, one possible choice for this pair is $n_0 = 1\ and\ c = 45$.

b) Selection Sort

YELLOW background shows the already sorted portion of the array.

GREEN elements show the latest swapped elements.

18	4	47	24	15	24	17	11	31	23

18	4	**23**	24	15	24	17	11	31	**47**
18	4	23	24	15	24	17	11	**31**	**47**
18	4	23	**11**	15	24	17	**24**	**31**	**47**
18	4	23	11	15	**17**	**24**	**24**	**31**	**47**
18	4	**17**	11	15	**23**	**24**	**24**	**31**	**47**
15	4	17	11	**18**	**23**	**24**	**24**	**31**	**47**
15	4	**11**	**17**	**18**	**23**	**24**	**24**	**31**	**47**
11	4	**15**	**17**	**18**	**23**	**24**	**24**	**31**	**47**
4	**11**	**15**	**17**	**18**	**23**	**24**	**24**	**31**	**47**

Bubble Sort

YELLOW background shows the already sorted portion of the array.

BLUE background elements show the latest swapped elements.

1st PASS

18	4	47	24	15	24	17	11	31	23

4	18	47	24	15	24	17	11	31	23
4	18	47	24	15	24	17	11	31	23
4	18	24	47	15	24	17	11	31	23
4	18	24	15	47	24	17	11	31	23
4	18	24	15	24	47	17	11	31	23
4	18	24	15	24	17	47	11	31	23
4	18	24	15	24	17	11	47	31	23
4	18	24	15	24	17	11	31	47	23
4	18	24	15	24	17	11	31	23	47

2nd PASS

4	18	24	15	24	17	11	31	23	47

4	18	24	15	24	17	11	31	23	47
4	18	24	15	24	17	11	31	23	47
4	18	15	24	24	17	11	31	23	47
4	18	15	24	24	17	11	31	23	47
4	18	15	24	17	24	11	31	23	47
4	18	15	24	17	11	24	31	23	47

4	18	15	24	17	11	24	31	23	47
4	18	15	24	17	11	24	23	31	47

3^{rd} PASS

4	18	15	24	17	11	24	23	31	47

4	18	15	24	17	11	24	23	31	47
4	15	18	24	17	11	24	23	31	47
4	15	18	24	17	11	24	23	31	47
4	15	18	17	24	11	24	23	31	47
4	15	18	17	11	24	24	23	31	47
4	15	18	17	11	24	24	23	31	47
4	15	18	17	11	24	23	24	31	47

4^{rd} PASS

4	15	18	17	11	24	23	24	31	47

4	15	18	17	11	24	23	24	31	47
4	15	18	17	11	24	23	24	31	47
4	15	17	18	11	24	23	24	31	47
4	15	17	11	18	24	23	24	31	47
4	15	17	11	18	24	23	24	31	47
4	15	17	11	18	23	24	24	31	47

5^{th} PASS

4	15	17	11	18	23	24	24	31	47

4	15	17	11	18	23	24	24	31	47
4	15	17	11	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47
4	15	11	17	18	23	24	24	31	47

6th PASS

4	15	11	17	18	23	24	24	31	47

4	15	11	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47

7th PASS

4	11	15	17	18	23	24	24	31	47

4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47
4	11	15	17	18	23	24	24	31	47

No swaps are done in 7th pass, the array is fully sorted.

Result:

4	11	15	17	18	23	24	24	31	47

Question 2:

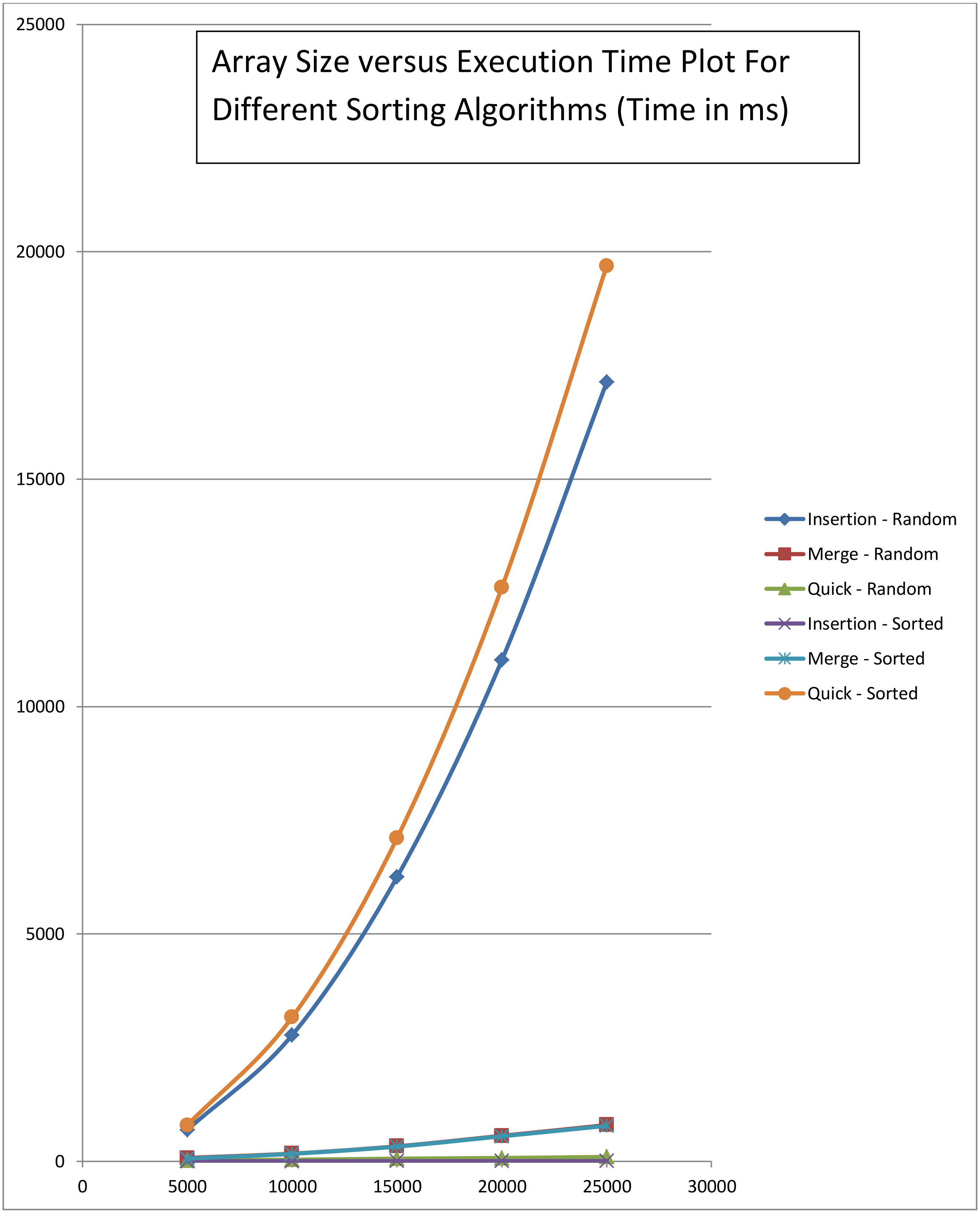
b)

```
For Insertion Sort, compCount = 71, moveCount = 89
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18

For Merge Sort, compCount = 46, moveCount = 128
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18

For Quicksort, compCount = 47, moveCount = 114
0       2       3       5       6       7       8       9       9       11      1
1       14      15      16      17      18
```

c)

```
Question 2, Part(c)

#######################################
Test with Random Arrays
#######################################
-----------------------------------------------------
Part c - Time analysis of Insertion Sort
Array Size          Time Elapsed        compCount           moveCount
5000                688                 6203592             6208600
10000               2775                25037796            25047803
15000               6249                56514250            56529266
20000               11021               99842616            99862628
25000               17133               155354752           155379762
-----------------------------------------------------
Part c - Time analysis of Merge Sort
Array Size          Time Elapsed        compCount           moveCount
5000                72                  55241               123616
10000               168                 120456              267232
15000               328                 189375              417232
20000               559                 260963              574464
25000               795                 334097              734464
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size          Time Elapsed        compCount           moveCount
5000                17                  72409               129295
10000               34                  154143              255569
15000               57                  249051              400908
20000               72                  326109              542117
25000               95                  418743              721190
#######################################
Test with Already Sorted Arrays
#######################################
-----------------------------------------------------
Part c - Time analysis of Insertion Sort
Array Size          Time Elapsed        compCount           moveCount
5000                0                   4999                9998
10000               1                   9999                19998
15000               2                   14999               29998
20000               3                   19999               39998
25000               4                   24999               49998
-----------------------------------------------------
Part c - Time analysis of Merge Sort
Array Size          Time Elapsed        compCount           moveCount
5000                58                  32004               123616
10000               163                 69008               267232
15000               322                 106364              417232
20000               549                 148016              574464
25000               778                 188476              734464
-----------------------------------------------------
Part c - Time analysis of Quick Sort
Array Size          Time Elapsed        compCount           moveCount
5000                794                 12497500            19996
10000               3167                49995000            39996
15000               7105                112492500           59996
20000               12618               199990000           79996
25000               19691               312487500           99996
```

d)Plot



Array Size versus Execution Time Plot For Different Sorting Algorithms (Time in ms)
25000
20000
15000
10000
5000
0
0
5000
10000
15000
20000
25000
30000
Insertion - Random
Merge - Random
Quick - Random
Insertion - Sorted
Merge - Sorted
Quick - Sorted

Analysis

The plot in the previous page shows the comparison of execution times for Insertion Sort, Merge Sort and Quicksort for random arrays and sorted arrays. And I have intentionally made the plot the whole page big to see the distinction between the curves more clearly. And when we look at the plot, what we first see is that there are 3 pairs of curves that are really close to each other.

First, we see **Quicksort with sorted arrays** and **Insertion Sort with random arrays** which are increasing faster than the all the other. This is what we expect since their time complexity for the given data is O(n^2) as we discussed in the class. Another point to note about **Quicksort with sorted arrays** is that it had O(n) space complexity in this case because of the stack filling with recursive calls. In my IDE, I had to increase the size of the stack frame in order to execute this program even for arrays of size 10000. This shows how ineffective Quicksort can be with sorted arrays, if the choosePivot function is chosen badly.

After that, we see **Merge Sort with random arrays** and **Merge Sort with sorted arrays** with a lower increasing rate. Again, this is what we expect from Merge Sort as we know its best, worst and average cases are all O(nlogn). The data shows that having a sorted array has almost no effect on Merge Sort's speed.

At the bottom, we see **Quicksort with random arrays** and **Insertion Sort with sorted arrays**. From the theoretical results, we know that the quicksort should be O(nlogn) and insertion sort should be O(n). The reason that they seem close in the graph is because of the scale of the graph. If we look at the results directly from the console output of the program, we can see that quicksort actually behaves like O(nlogn) and insertion sort behaves like O(n).

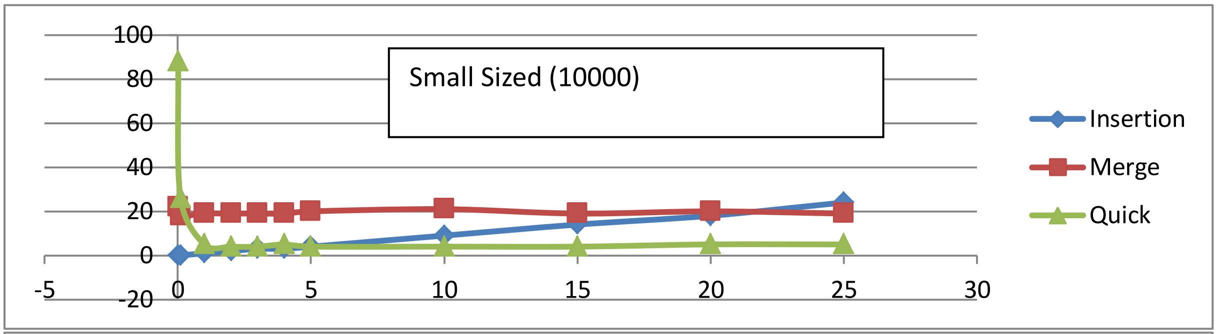
The reason why the curves for **Merge Sort** and the **Quicksort with random arrays** are not close although they have the same time complexity O(nlogn) is that the constant that comes before the nlogn is different for the two sorting algorithms. Quicksort is visibly faster than Merge Sort and it doesn't have extra space usage. The only disadvantage of quicksort is that we can observe in these tests is that it works in O(n^2) when the given array is sorted. But actually, if we had a better choosePivot function for quicksort such as choosing the median of a few random elements, it would work in O(nlogn) even with sorted arrays. As a conclusion from these tests, Insertion Sort is obviously the best for sorted arrays and Quicksort is much better for random arrays.

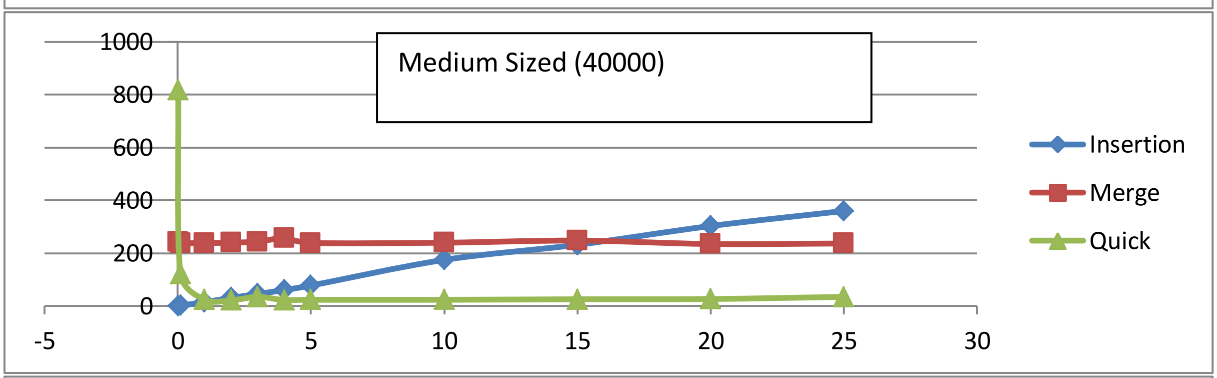
Question 3:

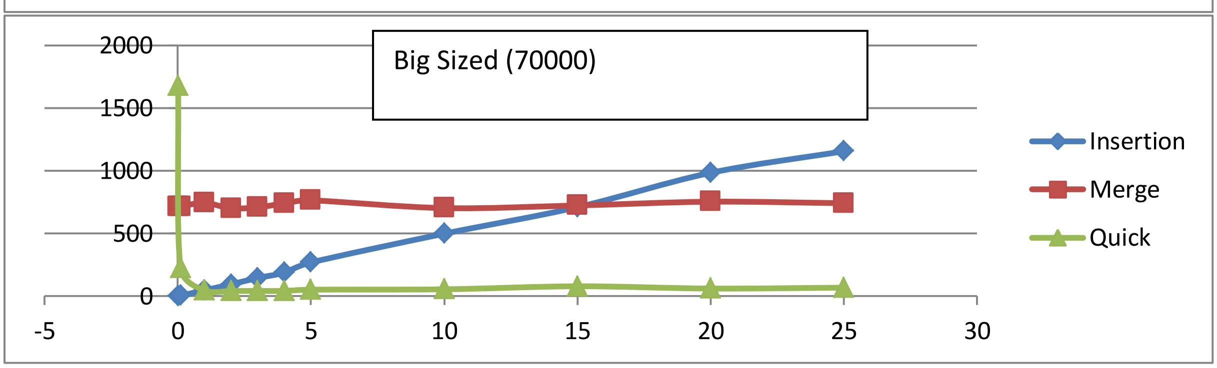
Tests and Data

For this part of the question, I have divided the tests into 3 three parts: Tests with small sized (10000), medium sized (40000) and big sized (70000) arrays. I couldn't go lower than 10000, because the execution times were becoming 0 milliseconds, decreasing the quality of the test. And I couldn't go higher than 100000, because it was starting take so much time to execute, therefore I have chosen 70000 for the big sized arrays.

With each of these three different array sizes, I then made 11 different tests, where the ratio of K to N as a percentage was changing from 0.01% to 25%. The following plots show the data I obtained where the y-axis is the execution time and the x-axis the ratio between K and N as a percentage.







Analysis

When we look into the plots, we can easily see how each of three sorting algorithms behaves for different values of the K/N ratio.

The execution time of **Insertion Sort** grows linearly as this ratio increases. This is what we would expect since **Insertion Sort** works only by swapping the elements that are adjacent to each other. Therefore, when K value increases, the number of required swaps increases too.

Similar to what we have seen in Question 2, the execution time of **Merge Sort** is not really affected by the distribution of data in the array. Its execution time stays the same for different values of K.

In Question 2, we have seen that **Quicksort** has time complexity O(n^2) when the array is sorted and O(nlogn) when it is random. Having a really small K is a similar case to having a sorted array, so we see that the execution time of **Quicksort** starts at the top when the ratio of K/N is 0.01%. But after that, for K/N = 0.1%, it quickly becomes better than **Merge Sort**.

Now, with this information about the sorting algorithms, I would choose **Insertion Sort** for very small values of K relative to N (K/N < 1%), as it performs very close to O(n). For larger values of K relative to N (K/N > 1%), I would choose **Quicksort** as it quickly becomes faster than the other two sorting algorithms.

As another possibility, if we don't know the distribution of the data, I would choose **Insertion Sort** for small sized arrays as it could be risky to choose **Quicksort**. For medium or larger sized arrays, I would choose **Merge Sort** because of the same risk again. For example, if we would try to sort a million element fully sorted array with **Quicksort**, it could take a lot of tame whereas the **Merge Sort** is a safe choice. Actually, **Merge Sort**'s time complexity is the same as **Quicksort** and they actually only differ by some constant in my test results, so it wouldn't be a really big problem. But, as I discussed in Question 2, **Quicksort** would be the best pick for almost all cases if we had a nice choosePivot function.

Note: Later, when I ran the program on the dijkstra server or in a computer in the university labs, **Merge Sort** was at the same speed with **Quicksort**, or sometimes even faster. This means that the selection of the ideal sorting algorithm might change with the specifications of the used computer. With the data I obtained, **Quicksort** becomes the best choice for most situations. But if I would be using the results I obtained from dijkstra server, **Merge Sort** would be a very good alternative to **Quicksort** with the possible disadvantage of having O(n) space complexity.