Ayda Yurtoğlu
21903153
CS202 Section 1

Question 1:

(a) Show $f(n) = 5n^3 + 4n^2 + 10$ is $O(n^4)$.

There should be appropriate c and n_0 that satisfy the equation:

$5n^3 + 4n^2 + 10 <= cn^4$ for all $n >= n_0$

If we pick $c = 19$ and $n_0 = 1$, then this equation is satisfied. Then, by the definition of Big-O, this statement is true.

(b) Trace the sorting of the array in ascending order: [24, 8, 51, 28, 20, 29, 21, 17, 38, 27]

1- Insertion sort

Divide the array into two parts: sorted and unsorted. Take [24] as the sorted part, the rest is unsorted. Then, loop from i=1 to i=9.

For i=1, the current element is 8. It is smaller than 24, so we insert 8 before 24. The logic is to pick an element from the unsorted array and find the right location in the sorted array to insert this element, by comparing the elements of the sorted array with this element.
Current Array: [8, 24, 51, 28, 20, 29, 21, 17, 38, 27]

For i=2, the current element = 51 and it is greater than all elements in the sorted array, so it will remain at the same place.
Current Array: [8, 24, 51, 28, 20, 29, 21, 17, 38, 27]

For i=3, the current element = 28 and it is greater 24 and smaller than 51. It will be inserted in between these elements.
Current Array: [8, 24, 28, 51, 20, 29, 21, 17, 38, 27]

For i=4, the current element = 20 and it is greater 8 and smaller than 24. It will be inserted in between these elements.
Current Array: [8, 20, 24, 28, 51, 29, 21, 17, 38, 27]

For i=5, the current element = 29 and it is greater 28 and smaller than 51. It will be inserted in between these elements.
Current Array: [8, 20, 24, 28, 29, 51, 21, 17, 38, 27]

For i=6, the current element = 21 and it is greater 20 and smaller than 24. It will be inserted in between these elements.
Current Array: [8, 20, 21, 24, 28, 29, 51, 17, 38, 27]

For i=7, the current element = 17 and it is greater 8 and smaller than 20. It will be inserted in between these elements.
Current Array: [8, 17, 20, 21, 24, 28, 29, 51, 38, 27]

For i=8, the current element = 38 and it is greater 29 and smaller than 51. It will be inserted in between these elements.
Current Array: [8, 17, 20, 21, 24, 28, 29, 38, 51, 27]

For i=9, the current element = 27 and it is greater 24 and smaller than 28. It will be inserted in between these elements.
Current Array: [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]

Now, the unsorted part of the array has no elements left and we obtained the sorted array in ascending order.

2- Bubble sort

We again start by diving the array into two parts: sorted and unsorted. Then, we will swap the adjacent elements starting from indexes 0 and 1 by comparing these elements. We will loop from i=0 to i=9 (unsorted array size-1). We will go through the array from the beginning until there is no need for swapping, in other words, until the array is sorted. There will be (array size-1) number of passes and at the end of each pass, the greatest element in the unsorted array will be put at the last index of the unsorted array.
Current array: [24, 8, 51, 28, 20, 29, 21, 17, 38, 27]

Pass 1:
For i=0, the pair is (24,8) and we will swap between them, since 24 > 8.
Current array: [8, 24, 51, 28, 20, 29, 21, 17, 38, 27]

For i=1, the pair is (24,51) and we will not swap them, because they are in the right order.
Current array: [8, 24, 51, 28, 20, 29, 21, 17, 38, 27]

For i=2, the pair is (51,28) and we will swap them.
Current array: [8, 24, 28, 51, 20, 29, 21, 17, 38, 27]

Since 51 is greatest element in the whole array, it will be at the last index of this array when i=9. The sorted part of the array only includes 51 now. We need to perform 8 more passes to obtain the full sorted array. I will not show the details of each pass, since I showed it in the first one.
Current array: [8, 24, 28, 20, 29, 21, 17, 38, 27, 51]

Pass 2: The greatest element in the unsorted array is 38. There will be switches in the unsorted array, because there are pairs in between that should be switched. (For example, the pair (24,20) is switched, but neither of the elements are the greatest element of the unsorted array.)
Current array: [8, 24, 20, 28, 21, 17, 29, 27, 38, 51]

Pass 3: The greatest element in the unsorted array is 29.

Current array: [8, 20, 24, 21, 17, 28, 27, 29, 38, 51]

Pass 4: The greatest element in the unsorted array is 28.
Current array: [8, 20, 21, 17, 24, 27, 28, 29, 38, 51]

Pass 5: The greatest element in the unsorted array is 27.
Current array: [8, 20, 17, 21, 24, 27, 28, 29, 38, 51]

Pass 6: The greatest element in the unsorted array is 24. At this point, the array is fully sorted. Bubble sort will continue checking the pairs, but will not perform any swap operations. So, the array will remain the same for the remaining passes.
Current array: [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]

Pass 7:
Current array: [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]

Pass 8:
Current array: [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]

Pass 9:
Current array: [8, 17, 20, 21, 24, 27, 28, 29, 38, 51]
The algorithm stops, we have obtained the sorted array.

Question 2:

```
[ayda.yurtoglu@dijkstra cs202hw1]$ ./hw1
Selection sort:
Number of comparisons: 120
Number of data moves: 45
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }

Merge sort:
Number of comparisons: 46
Number of data moves: 128
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }

Quick:
Number of comparisons: 45
Number of data moves: 102
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }

Radix sort:
{ 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21 }

[ayda.yurtoglu@dijkstra cs202hw1]$
```

```
RANDOM ARRAYS
-----------------------------------------------------
Analysis of Selection Sort
Array Size         Elapsed Time         compCount         moveCount
6000              80 ms              17997000              17997
10000              220 ms              49995000              29997
14000              440 ms              97993000              41997
18000              720 ms              161991000              53997
22000              1080 ms              241989000              65997
26000              1510 ms              337987000              77997
30000              2000 ms              449985000              89997

-----------------------------------------------------
Analysis of Merge Sort
Array Size         Elapsed Time         compCount         moveCount
6000              0 ms              67944              151616
10000              10 ms              120481              267232
14000              0 ms              175324              387232
18000              10 ms              232021              510464
22000              10 ms              290153              638464
26000              10 ms              348883              766464
30000              10 ms              408571              894464

-----------------------------------------------------
Analysis of Quick Sort
Array Size         Elapsed Time         compCount         moveCount
6000              0 ms              93621              158664
10000              0 ms              156927              262737
14000              10 ms              220809              326196
18000              0 ms              302232              503347
22000              0 ms              379335              653469
26000              0 ms              438508              699248
30000              10 ms              509046              788361

-----------------------------------------------------
Analysis of Radix Sort
Array Size         Elapsed Time
6000              0 ms
10000              0 ms
14000              0 ms
18000              10 ms
22000              10 ms
26000              10 ms
30000              10 ms
```

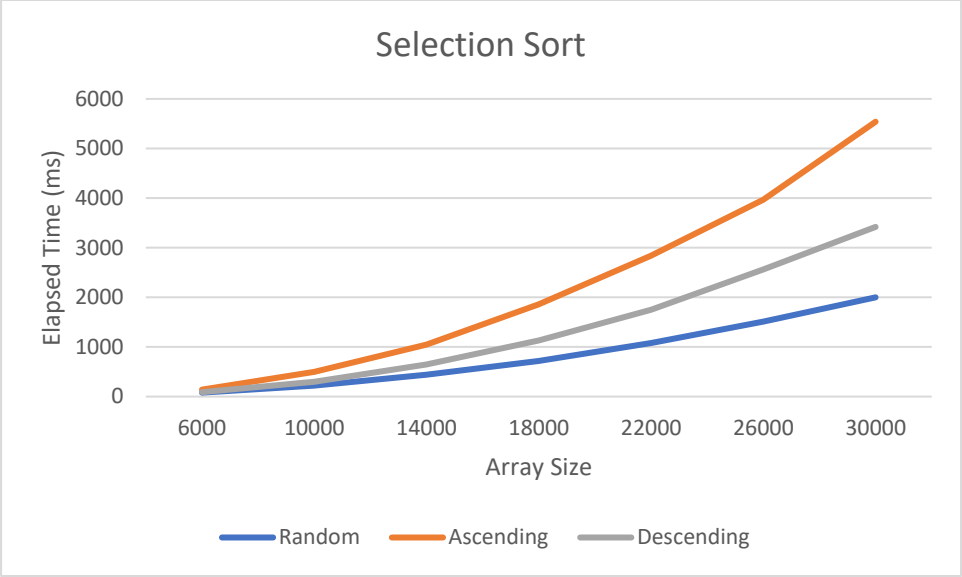
```
ASCENDING ARRAYS
-----------------------------------------------------
Analysis of Selection Sort
Array Size         Elapsed Time         compCount         moveCount
6000              140 ms             17997000              17997
10000              500 ms              49995000              29997
14000              1050 ms              97993000              41997
18000              1860 ms              161991000              53997
22000              2840 ms              241989000              65997
26000              3970 ms              337987000              77997
30000              5540 ms              449985000              89997

-----------------------------------------------------
Analysis of Merge Sort
Array Size         Elapsed Time         compCount         moveCount
6000              0 ms             39984               151616
10000              0 ms             70497               267232
14000              0 ms             101699               387232
18000              0 ms             133904               510464
22000              0 ms             168080               638464
26000              0 ms             200771               766464
30000              10 ms              233274               894464

-----------------------------------------------------
Analysis of Quick Sort
Array Size         Elapsed Time         compCount         moveCount
6000              80 ms             17997000              23996
10000              220 ms              49995000              39996
14000              410 ms              97993000              55996
18000              690 ms              161991000              71996
22000              1020 ms              241989000              87996
26000              1430 ms              337987000              103996
30000              1900 ms              449985000              119996

-----------------------------------------------------
Analysis of Radix Sort
Array Size         Elapsed Time
6000              0 ms
10000              0 ms
14000              10 ms
18000              0 ms
22000              10 ms
26000              10 ms
30000              10 ms
```

```
DESCENDING ARRAYS
-----------------------------------------------------
Analysis of Selection Sort
Array Size         Elapsed Time         compCount         moveCount
6000              90 ms              17997000              17997
10000              300 ms              49995000              29997
14000              650 ms              97993000              41997
18000              1130 ms              161991000              53997
22000              1750 ms              241989000              65997
26000              2560 ms              337987000              77997
30000              3420 ms              449985000              89997

-----------------------------------------------------
Analysis of Merge Sort
Array Size         Elapsed Time         compCount         moveCount
6000              0 ms              36656              151616
10000              0 ms              64608              267232
14000              0 ms              94256              387232
18000              0 ms              124640              510464
22000              10 ms              154208              638464
26000              0 ms              186160              766464
30000              10 ms              219504              894464

-----------------------------------------------------
Analysis of Quick Sort
Array Size         Elapsed Time         compCount         moveCount
6000              120 ms              14088243              21153226
10000              340 ms              39054266              58616386
14000              670 ms              76522927              114833336
18000              1090 ms              124669785              187066919
22000              1630 ms              186425116              279713558
26000              2300 ms              263110172              394755781
30000              3060 ms              350085794              525233545

-----------------------------------------------------
Analysis of Radix Sort
Array Size         Elapsed Time
6000              10 ms
10000              10 ms
14000              0 ms
18000              10 ms
22000              0 ms
26000              10 ms
30000              10 ms
```

Question 3:

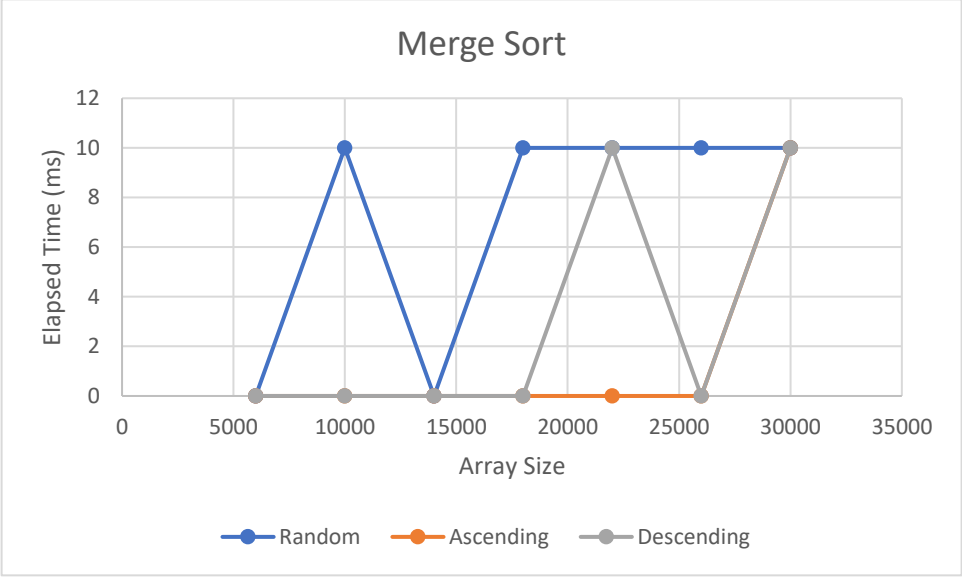
Selection Sort:

Time complexity: $O(n^2)$



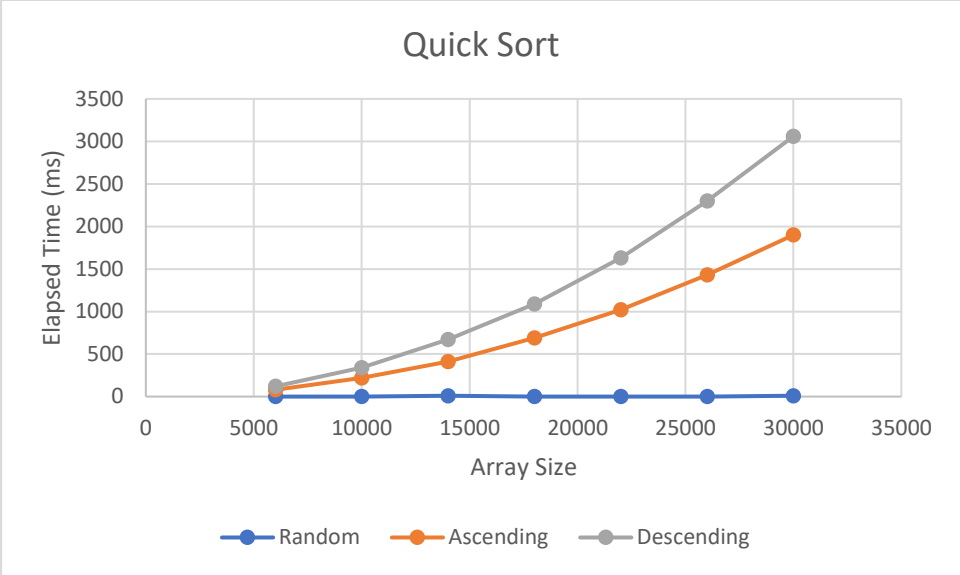
As the array size increased, the time to sort also increased with time complexity $O(n^2)$ for all cases. However, the elapsed time was the greatest in the ascending scenario, when the array was already sorted. This was different from what I expected, because the best, worst and average cases should be the same in theory. My empirical results suggest that when the array is random the time it took to sort is the smallest. I think this is just a coincide and it doesn't matter whether the array is actually sorted or not. Also, the time we measure is very small and although the graph suggests that there is a huge difference, the results is coincidental and might be the same when I run the program again.

Merge Sort:



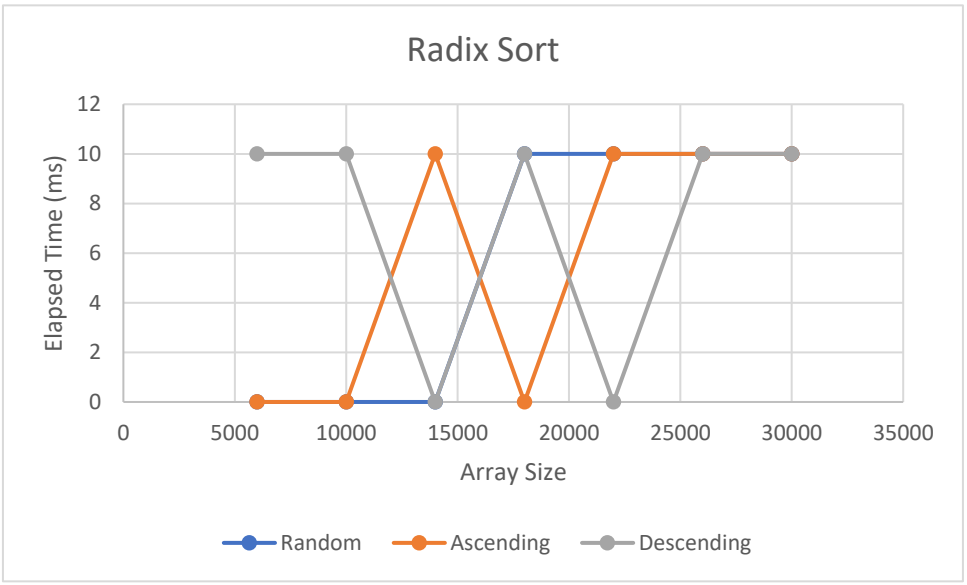
Time complexity is O(n*logn) for all cases of merge sort. Because of Dijkstra's rounding, the results were interesting in merge sort. If it didn't round up, I would have gathered more precise information about elapsed time. In merge sort, theoretically the worst case and the average case should be the same in terms of time complexity. This is what I observed in my results, because the elapsed time were almost the same for all three cases. So, it didn't matter whether the array was sorted at first or not. Since the elapsed time didn't change drastically as it did in selection sort, we can say that merge sort is a very efficient sorting function.

Quick Sort:



For quick sort's average case, the time complexity is O(n*logn). For the worst case, it is $O(n^2)$. When we select the first element as pivot, worst case happens when the array is already sorted. This is because the function will keep diving the array into two sub arrays of sizes 0 and (size-1) instead of size/2 and size/2 (like in the average case). My results suggest that when the array was sorted in descending order, the time the function required was the greatest. When the array was random, it took very small amount of time to sort. We can also observe that for both ascending and descending arrays, the time complexity is $O(n^2)$ from the graph. This coincides with the theory for quick sort. So, the average case would be when the array is random.

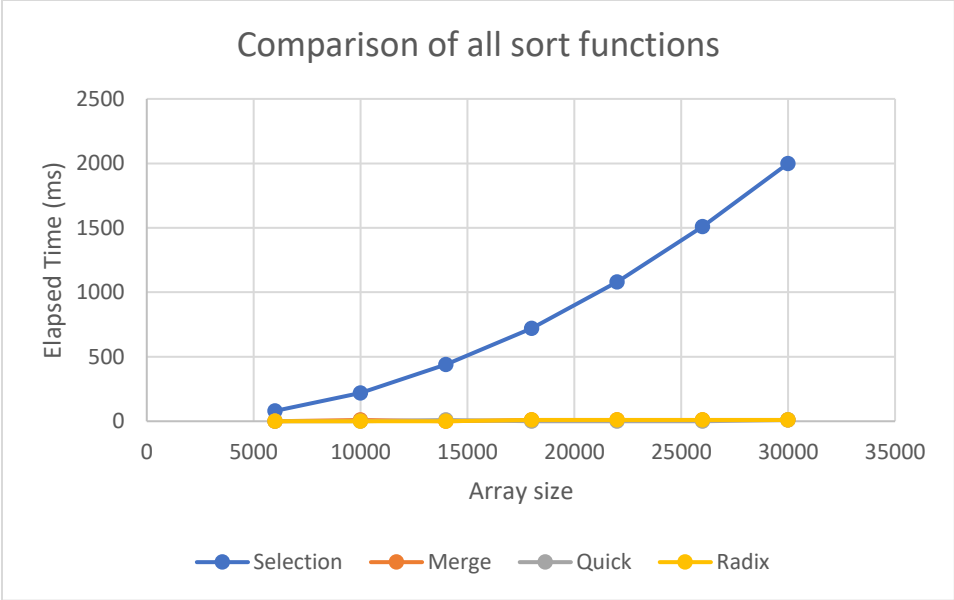
Radix Sort:



For all cases of radix sort, the time complexity is O(n). In theory, it shouldn't matter whether the arrays are sorted or not and my results suggest this. Since radix sort is very efficient, Dijkstra rounded up the elapsed time just like it did with merge sort, so results are not as precise as the selection sort. They are either 0 or 10. Still, we can understand that the scenarios don't matter when it comes to time complexity. The duration is random and is not dependent on the scenario. If we used larger array sizes or Dijkstra didn't round up, we would also see that as the array size increased the time also increased with O(n) complexity.

Comparing all functions:

I used MS Excel to compare the results of all sorting functions and their required time to sort random arrays. Since the time for selections sort was too large than the others, we can't see the red and the grey lines, because they are behind the yellow line. So, I created another graph that only compares merge sort, quick sort and radix sort.



Starting with the first graph, we can say that selection sort is the least efficient one. This is also the case when we compare the time complexities of these functions.

$O(n^2) > O(n*logn) > O(n)$

If we compare the other functions, we see that Dijkstra again rounded the results because they were so small. So, all of these functions are very efficient and the results we see here are random. Still, in theory radix sort should be more efficient than merge sort and quick sort due to its time complexity. We might have seen this result here if we have used greater array sizes to see the difference. We are able to see the difference of selection sort and the others because $O(n^2)$ grows very rapidly. This is not the case for other functions.