Cemhan Kaan Özaltan

21902695

Section 3

Homework 1 Report

Question 1

a) In order to show that $f(n) = 5n^3 + 4n^2 + 10$ is $O(n^4)$, we must prove that there are positive constants c and n_0 such that $f(n) \leq c \cdot n^4$ when $n \geq n_0$. If $5n^3 + 4n^2 + 10 \leq c \cdot n^4$, then $\frac{5}{n} + \frac{4}{n^2} + \frac{10}{n^4} \leq c$. If we let n be 1, we get $5 + 4 + 10 = 19 \leq c$. Therefore, the given Big-O value holds for $n \geq n_0 = 1$ and $c \geq 19$.

b) Array: [24, 8, 51, 28, 20, 29, 21, 17, 38, 27]. We will sort this array using Insertion Sort and Bubble Sort. Both algorithms take the first index as 1.

Using Insertion Sort: The array is divided into sorted and unsorted subarrays where the array is sorted to the left of the wall. We iterate the array for j from 2 to n. In each of these iterations, we iterate the sorted array for i from j – 1 to 1, shift the items right until we find the item that is smaller than the key (arr[j]) or reach index 1, place the key to the right of that item or the beginning of the array if no item is smaller than the key and move the wall to the right.
Initial array: [24,| 8, 51, 28, 20, 29, 21, 17, 38, 27]
j = 2: [8, 24,| 51, 28, 20, 29, 21, 17, 38, 27] (key = 8, no element is smaller than 8)
j = 3: [8, 24, 51,| 28, 20, 29, 21, 17, 38, 27] (key = 51, 24 is smaller than 51)
j = 4: [8, 24, 28, 51,| 20, 29, 21, 17, 38, 27] (key = 28, 24 is smaller than 28)
j = 5: [8, 20, 24, 28, 51,| 29, 21, 17, 38, 27] (key = 20, 8 is smaller than 20)
j = 6: [8, 20, 24, 28, 29, 51,| 21, 17, 38, 27] (key = 29, 28 is smaller than 29)
j = 7: [8, 20, 21, 24, 28, 29, 51,| 17, 38, 27] (key = 21, 20 is smaller than 21)
j = 8: [8, 17, 20, 21, 24, 28, 29, 51,| 38, 27] (key = 17, 8 is smaller than 17)
j = 9: [8, 17, 20, 21, 24, 28, 29, 38, 51,| 27] (key = 38, 29 is smaller than 38)
j = 10: [8, 17, 20, 21, 24, 27, 28, 29, 38, 51,|] (key = 27, 24 is smaller than 27)
j reached n and the array is sorted so the algorithm terminates.

Using Bubble Sort: The array is divided into sorted and unsorted subarrays where the array is sorted to the right of the wall. We iterate the array for pass from 1 to n or until it is sorted. In each of these iterations, we iterate the unsorted part from index to pass – index, compare arr[i] and arr[i + 1] and swap them if arr[i] is larger, have the largest

element in the unsorted subarray bubbled to its place in the sorted subarray and move the wall to the left.
Initial array: [24, 8, 51, 28, 20, 29, 21, 17, 38, 27||]
pass = 1: [8, 24, 28, 20, 29, 21, 17, 38, 27,| 51]
pass = 2: [8, 24, 20, 28, 21, 17, 29, 27,| 38, 51]
pass = 3: [8, 20, 24, 21, 17, 28, 27,| 29, 38, 51]
pass = 4: [8, 20, 21, 17, 24, 27,| 28, 29, 38, 51]
pass = 5: [8, 20, 17, 21, 24,| 27, 28, 29, 38, 51]
pass = 6: [8, 17, 20, 21,| 24, 27, 28, 29, 38, 51]
The array is sorted despite pass not having reached n so the algorithm terminates.

Question 2

b) Output of the main function:

```
Command Prompt
Microsoft Windows [Version 10.0.18363.1379]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Kaan>cd Desktop

C:\Users\Kaan\Desktop>cd HW1

C:\Users\Kaan\Desktop\HW1>g++ sorting.cpp main.cpp -o hw1

C:\Users\Kaan\Desktop\HW1>hw1
Array: 12, 7, 11, 18, 19, 9, 6, 14, 21, 3, 17, 20, 5, 12, 14, 8,

Selection Sort
compCount: 120
moveCount: 45
Array after sorting: 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

Merge Sort
compCount: 46
moveCount: 128
Array after sorting: 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

Quick Sort
compCount: 45
moveCount: 93
Array after sorting: 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,

Radix Sort
Array after sorting: 3, 5, 6, 7, 8, 9, 11, 12, 12, 14, 14, 17, 18, 19, 20, 21,
C:\Users\Kaan\Desktop\HW1>
```

c) Output of the performanceAnalysis function:

Analysis of Selection Sort
Random Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	36 ms	17997000	17997
10000	95 ms	49995000	29997
14000	184 ms	97993000	41997
18000	303 ms	161991000	53997
22000	452 ms	241989000	65997
26000	634 ms	337987000	77997
30000	841 ms	449985000	89997

Ascending Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	34 ms	17997000	17997

10000	96 ms	49995000	29997
14000	190 ms	97993000	41997
18000	311 ms	161991000	53997
22000	467 ms	241989000	65997
26000	651 ms	337987000	77997
30000	867 ms	449985000	89997

Descending Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	35 ms	17997000	17997
10000	95 ms	49995000	29997
14000	186 ms	97993000	41997
18000	307 ms	161991000	53997
22000	459 ms	241989000	65997
26000	641 ms	337987000	77997
30000	853 ms	449985000	89997

Analysis of Merge Sort

Random Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	0 ms	67790	151616
10000	1 ms	120364	267232
14000	2 ms	175394	387232
18000	3 ms	231972	510464
22000	4 ms	290039	638464
26000	5 ms	348929	766464
30000	7 ms	408545	894464

Ascending Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	0 ms	39152	151616
10000	1 ms	69008	267232
14000	1 ms	99360	387232
18000	2 ms	130592	510464
22000	2 ms	165024	638464
26000	4 ms	197072	766464
30000	5 ms	227728	894464

Descending Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	1 ms	36656	151616
10000	1 ms	64608	267232
14000	2 ms	94256	387232
18000	3 ms	124640	510464
22000	3 ms	154208	638464
26000	4 ms	186160	766464
30000	5 ms	219504	894464

Analysis of Quick Sort

Random Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	0 ms	88490	144894
10000	1 ms	157420	260613
14000	2 ms	236824	374595
18000	1 ms	286829	474828
22000	2 ms	358192	586365
26000	3 ms	481591	727044
30000	3 ms	525463	801309

Ascending Arrays:

Array Size	Elapsed Time	compCount	moveCount

6000	29 ms	17997000	17997
10000	79 ms	49995000	29997
14000	154 ms	97993000	41997
18000	254 ms	161991000	53997
22000	379 ms	241989000	65997
26000	531 ms	337987000	77997
30000	704 ms	449985000	89997

Descending Arrays:

Array Size	Elapsed Time	compCount	moveCount
6000	46 ms	17997000	27017997
10000	126 ms	49995000	75029997
14000	249 ms	97993000	147041997
18000	411 ms	161991000	243053997
22000	615 ms	241989000	363065997
26000	859 ms	337987000	507077997
30000	1143 ms	449985000	675089997

Analysis of Radix Sort

Random Arrays:

Array Size	Elapsed Time
6000	0 ms
10000	1 ms
14000	1 ms
18000	1 ms
22000	1 ms
26000	1 ms
30000	1 ms

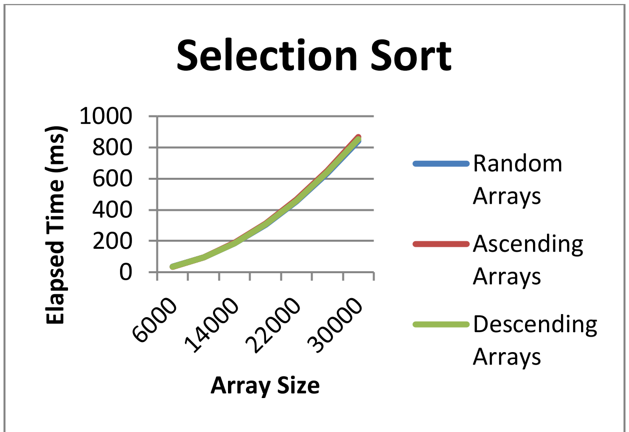
Ascending Arrays:

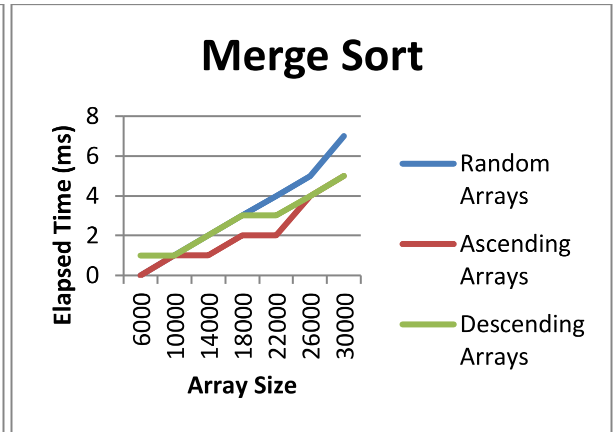
Array Size	Elapsed Time
6000	0 ms
10000	0 ms
14000	0 ms
18000	1 ms
22000	1 ms
26000	1 ms
30000	1 ms

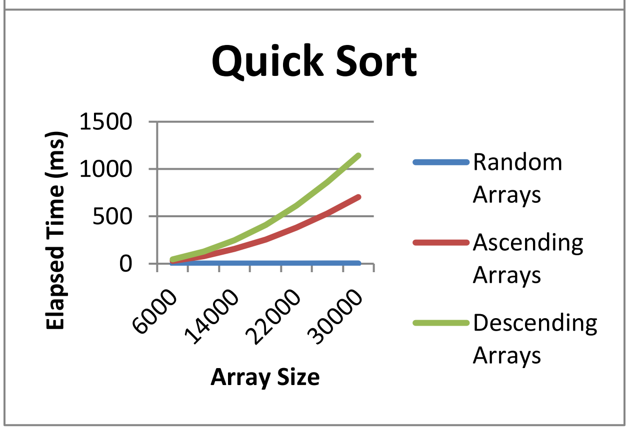
Descending Arrays:

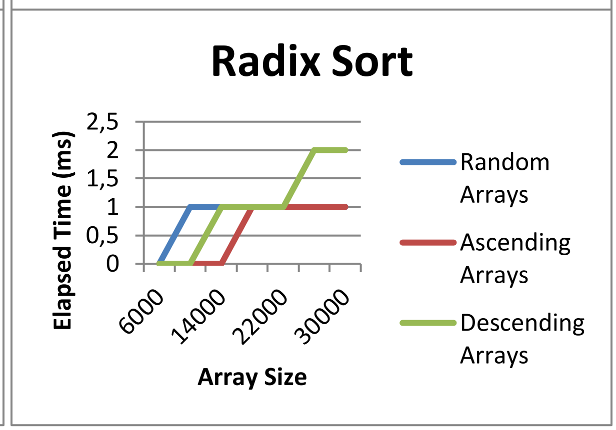
Array Size	Elapsed Time
6000	0 ms
10000	0 ms
14000	1 ms
18000	1 ms
22000	1 ms
26000	2 ms
30000	2 ms

Question 3









For selection sort, we see that the elapsed time increases exponentially with random arrays. This is because of the nested loops inside the algorithm causing it to be O(n^2). This is also valid for each case presented in the graph and this algorithm does not change its behavior for random, ascending or descending arrays and has the same best, worst and average case complexities which is O(n^2). This is because the unsorted subarray is traversed size – 1 times and the key comparison and data move counts have identical growth rates in each case.

For merge sort, we see that the elapsed time grows slightly larger than linearly with random arrays. Due to merge sort splitting the array into two halves and linearly merging them, it is O(nlogn). This is also valid for each case presented in the graph and this algorithm does not change its behavior for random, ascending or descending arrays and has the same best, worst and average case compexities which is O(nlogn). This is because the array is split into two halves and then merged and the key comparison and data move counts have identical growth rates in each case.

For quick sort, we see that the elapsed time increases slightly larger than linearly with random arrays. Due to quick sort partitioning the array into 2 subarrays and linearly merging them, it is O(nlogn). However this is not valid for ascending and descending arrays due to the algorithm taking the first item as pivot and not being able to partition equally since it is either the smallest or the largest item. Therefore, the best and average cases are with random arrays and are O(nlogn) and the worst case is with ascending or descending arrays and is O(n^2). The worst cases have higher comparison and move counts and the descending case has slightly higher move counts and elapsed times due to having more swap operations.

For radix sort, we see a linear increase in elapsed time with random arrays. Since radix sort makes 2 * size * digit moves, it is O(n), unless the digit count is greater than or equal to size, which makes it O(n^2). This is also valid for each case presented in the graph and this algorithm does not change its behavior for random, ascending or descending arrays and has the same best, worst and average case compexities which is O(n). This is because the array is traversed as many times as the digit and data move count has identical growth rates in each case while the algorithm does not use comparisons.