Ahmed Salih Cezayir 21802918 Section: 2

Question 1:

a)

If $f(n) = 5n^3+4n^2+10$ and $f(n) \in O(n^4)$, then there should be c and n_0 values such that, $5n^3+4n^2+10 <= c*n^4$ for $n > n_0$. For c and n_0, we can select 5 and 4, respectively. Then,

$$5n^3+4n^2+10 <= 5n^4$$

$$1+\frac{4}{5n}+\frac{2}{n^3} <= n.$$

For n values bigger than 4, this inequality is correct. Thus, by setting c=5 and n_0=4, we saw that, for $f(n) = 5n^3+4n^2+10$, $f(n) \in O(n^4)$.

b)

Insertion Sort:

Initial array:	[24, **8**, 51, 28, 20, 29, 21, 17, 38, 27]	Copy 8.
	[24, 24, 51, 28, 20, 29, 21, 17, 38, 27]	Shift 24.
	[8, 24, **51**, 28, 20, 29, 21, 17, 38, 27]	Insert 8; copy 51, insert 51 on top of itself.
	[8, 24, 51, **28**, 20, 29, 21, 17, 38, 27]	Copy 28.
	[8, 24, 51, 51, 20, 29, 21, 17, 38, 27]	Shift 51.
	[8, 24, 28, 51, **20**, 29, 21, 17, 38, 27]	Insert 28; copy 20.
	[8, 24, 24, 28, 51, 29, 21, 17, 38, 27]	Shift 24, 28, 51.
	[8, 20, 24, 28, 51, **29**, 21, 17, 38, 27]	Insert 20; copy 29.
	[8, 20, 24, 28, 51, 51, 21, 17, 38, 27]	Shift 51.
	[8, 20, 24, 28, 29, 51, **21**, 17, 38, 27]	Insert 29; copy 21.
	[8, 20, 24, 24, 28, 29, 51, 17, 38, 27]	Shift 51, 29, 28, 24.
	[8, 20, 21, 24, 28, 29, 51, **17**, 38, 27]	Insert 21; copy 17.
	[8, 20, 20, 21, 24, 28, 29, 51, 38, 27]	Shift 51, 29, 28, 24, 21, 20.
	[8, 17, 20, 21, 24, 28, 29, 51, **38**, 27]	Insert 17; copy 38.
	[8, 17, 20, 21, 24, 28, 29, 51, 51, 27]	Shift 51.
	[8, 17, 20, 21, 24, 28, 29, 38, 51, **27**]	Insert 38; copy 27.
	[8, 17, 20, 21, 24, 28, 28, 29, 38, 51]	Shift 51, 38, 29, 28.
Sorted array:	[8, 17, 20, 21, 24, 27, 28, 29, 38, 51]	Insert 27.

Bubble Sort:

Initial array:	[24, 8, 51, 28, 20, 29, 21, 17, 38, 27]	Start of pass 1.
	[8, 24, 51, 28, 20, 29, 21, 17, 38, 27]	
	[8, 24, 51, 28, 20, 29, 21, 17, 38, 27]	
	[8, 24, 28, 51, 20, 29, 21, 17, 38, 27]	
	[8, 24, 28, 20, 51, 29, 21, 17, 38, 27]	
	[8, 24, 28, 20, 29, 51, 21, 17, 38, 27]	
	[8, 24, 28, 20, 29, 21, 51, 17, 38, 27]	
	[8, 24, 28, 20, 29, 21, 17, 51, 38, 27]	
	[8, 24, 28, 20, 29, 21, 17, 38, 51, 27]	
	[8, 24, 28, 20, 29, 21, 17, 38, 27, 51]	
	[8, 24, 28, 20, 29, 21, 17, 38, 27, 51]	Start of pass 2.
	[8, 24, 28, 20, 29, 21, 17, 38, 27, 51]	
	[8, 24, 28, 20, 29, 21, 17, 38, 27, 51]	
	[8, 24, 20, 28, 29, 21, 17, 38, 27, 51]	
	[8, 24, 20, 28, 29, 21, 17, 38, 27, 51]	
	[8, 24, 20, 28, 21, 29, 17, 38, 27, 51]	
	[8, 24, 20, 28, 21, 17, 29, 38, 27, 51]	
	[8, 24, 20, 28, 21, 17, 29, 38, 27, 51]	
	[8, 24, 20, 28, 21, 17, 29, 27, 38, 51]	
	[8, 24, 20, 28, 21, 17, 29, 27, 38, 51]	Start of pass 3.
	[8, 24, 20, 28, 21, 17, 29, 27, 38, 51]	
	[8, 20, 24, 28, 21, 17, 29, 27, 38, 51]	
	[8, 20, 24, 28, 21, 17, 29, 27, 38, 51]	
	[8, 20, 24, 21, 28, 17, 29, 27, 38, 51]	
	[8, 20, 24, 21, 17, 28, 29, 27, 38, 51]	
	[8, 20, 24, 21, 17, 28, 29, 27, 38, 51]	
	[8, 20, 24, 21, 17, 28, 27, 29, 38, 51]	
	[8, 20, 24, 21, 17, 28, 27, 29, 38, 51]	Start of pass 4.
	[8, 20, 24, 21, 17, 28, 27, 29, 38, 51]	
	[8, 20, 24, 21, 17, 28, 27, 29, 38, 51]	

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

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

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

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

[8, 20, 21, 17, 24, 27, 28, 29, 38, 51] Start of pass 5.

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

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

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

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

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

[8, 20, 17, 21, 24, 27, 28, 29, 38, 51] Start of pass 6.

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

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

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

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

[8, 17, 20, 21, 24, 27, 28, 29, 38, 51] Start of pass 7.

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

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

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

[8, 17, 20, 21, 24, 27, 28, 29, 38, 51] Start of pass 8.

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

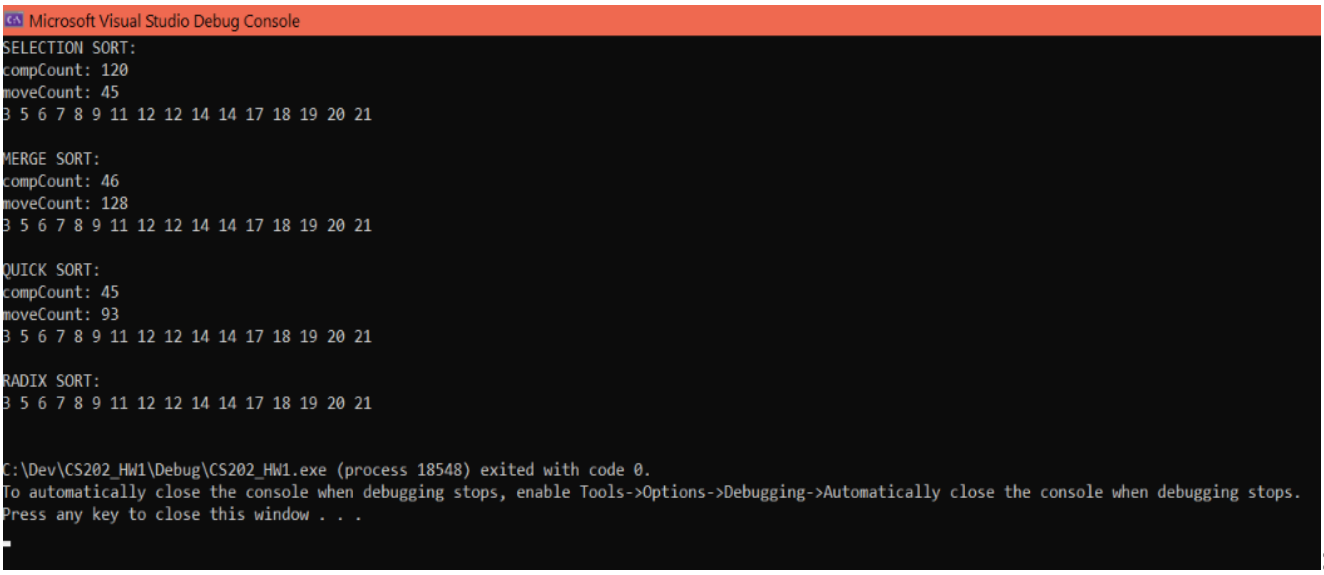
[8, 17, 20, 21, 24, 27, 28, 29, 38, 51]

[8, 17, 20, 21, 24, 27, 28, 29, 38, 51] Start of pass 9.

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

Question 2:

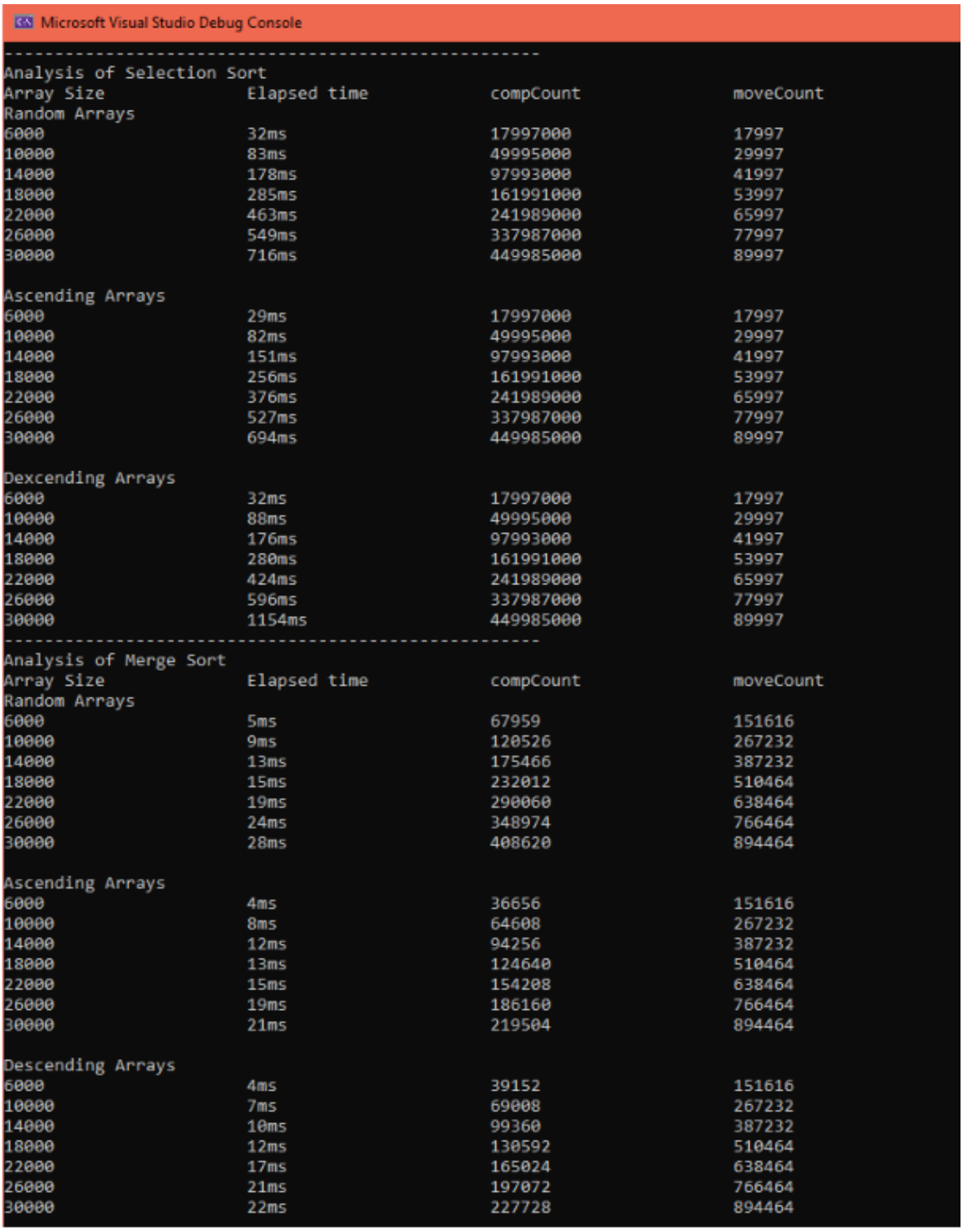
c)



:

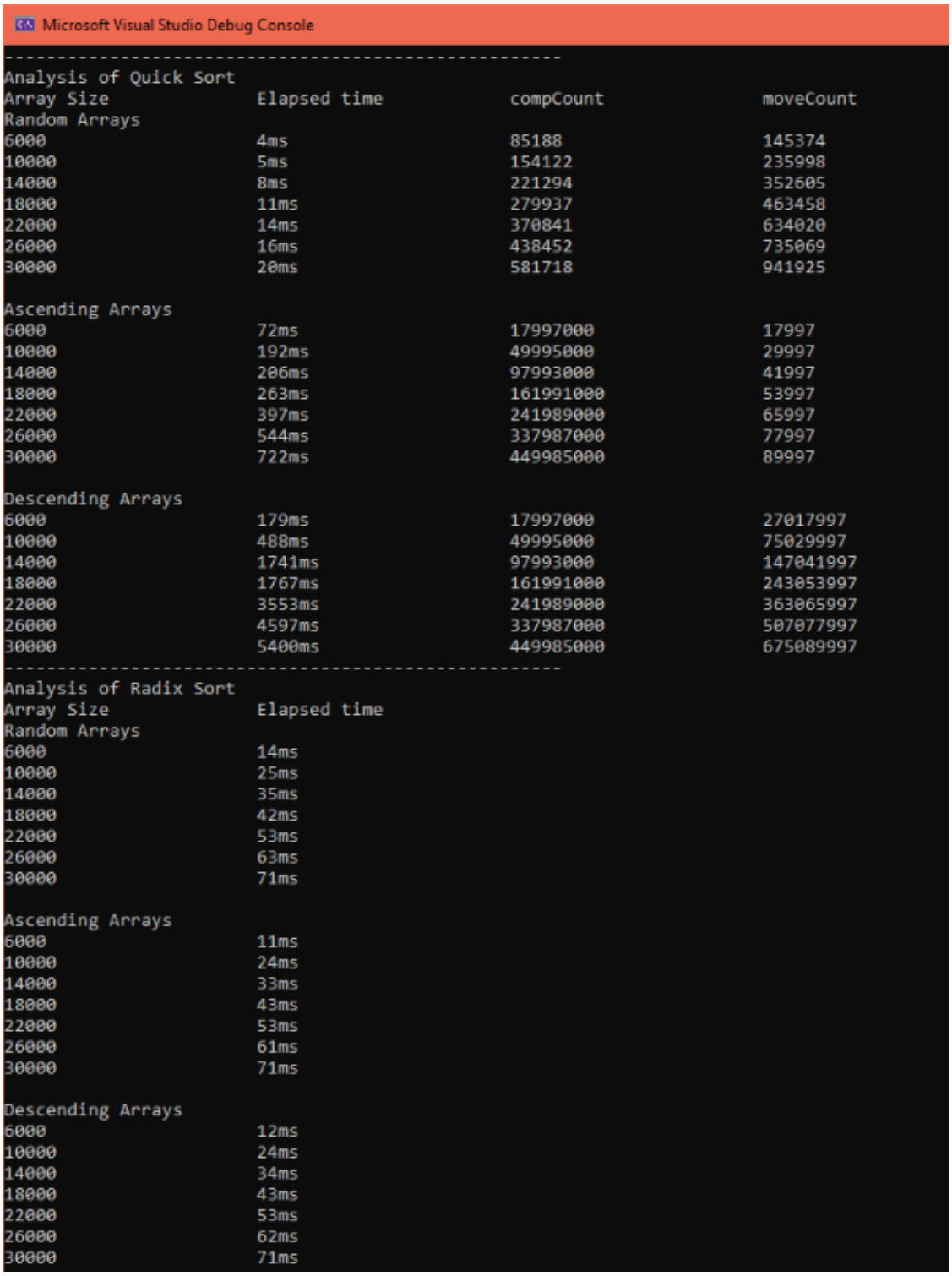
d)

For the d section, the output of the performanceAnalysis() function can be seen in the next two pages.


Microsoft Visual Studio Debug Console

Analysis of Selection Sort
Array Size Elapsed time compCount moveCount
Random Arrays
6000 32ms 17997000 17997
10000 83ms 49995000 29997
14000 178ms 97993000 41997
18000 285ms 161991000 53997
22000 463ms 241989000 65997
26000 549ms 337987000 77997
30000 716ms 449985000 89997
Ascending Arrays
6000 29ms 17997000 17997
10000 82ms 49995000 29997
14000 151ms 97993000 41997
18000 256ms 161991000 53997
22000 376ms 241989000 65997
26000 527ms 337987000 77997
30000 694ms 449985000 89997
Dexcending Arrays
6000 32ms 17997000 17997
10000 88ms 49995000 29997
14000 176ms 97993000 41997
18000 280ms 161991000 53997
22000 424ms 241989000 65997
26000 596ms 337987000 77997
30000 1154ms 449985000 89997

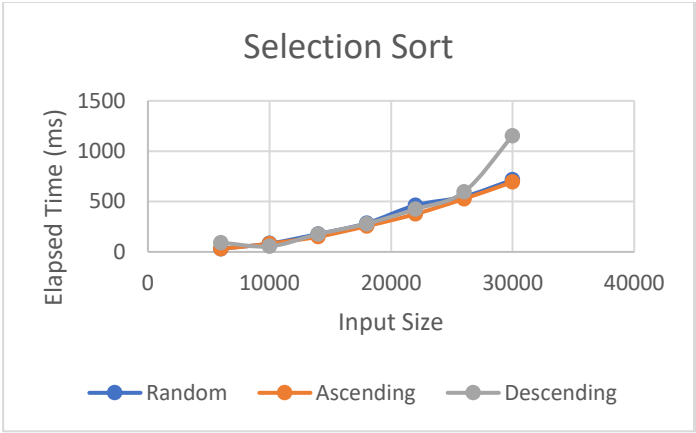
Analysis of Merge Sort
Array Size Elapsed time compCount moveCount
Random Arrays
6000 5ms 67959 151616
10000 9ms 120526 267232
14000 13ms 175466 387232
18000 15ms 232012 510464
22000 19ms 290060 638464
26000 24ms 348974 766464
30000 28ms 408620 894464
Ascending Arrays
6000 4ms 36656 151616
10000 8ms 64608 267232
14000 12ms 94256 387232
18000 13ms 124640 510464
22000 15ms 154208 638464
26000 19ms 186160 766464
30000 21ms 219504 894464
Descending Arrays
6000 4ms 39152 151616
10000 7ms 69008 267232
14000 10ms 99360 387232
18000 12ms 130592 510464
22000 17ms 165024 638464
26000 21ms 197072 766464
30000 22ms 227728 894464

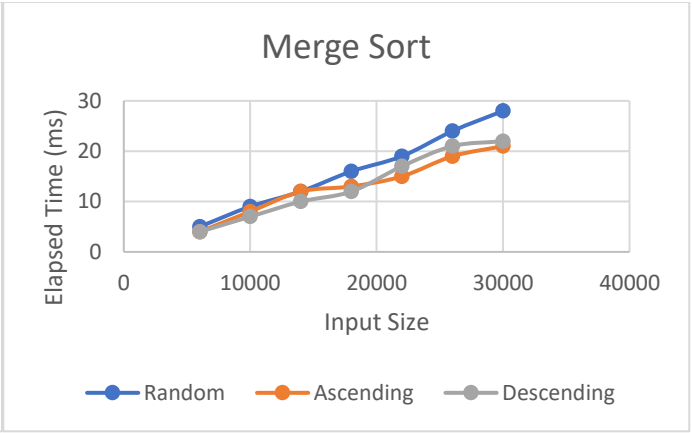

Microsoft Visual Studio Debug Console

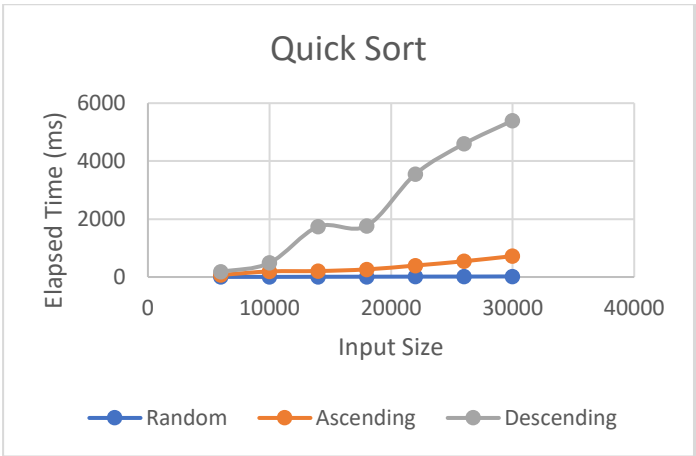
Analysis of Quick Sort
Array Size Elapsed time compCount moveCount
Random Arrays
6000 4ms 85188 145374
10000 5ms 154122 235998
14000 8ms 221294 352605
18000 11ms 279937 463458
22000 14ms 370841 634020
26000 16ms 438452 735069
30000 20ms 581718 941925
Ascending Arrays
6000 72ms 17997000 17997
10000 192ms 49995000 29997
14000 206ms 97993000 41997
18000 263ms 161991000 53997
22000 397ms 241989000 65997
26000 544ms 337987000 77997
30000 722ms 449985000 89997
Descending Arrays
6000 179ms 17997000 27017997
10000 488ms 49995000 75029997
14000 1741ms 97993000 147041997
18000 1767ms 161991000 243053997
22000 3553ms 241989000 363065997
26000 4597ms 337987000 507077997
30000 5400ms 449985000 675089997

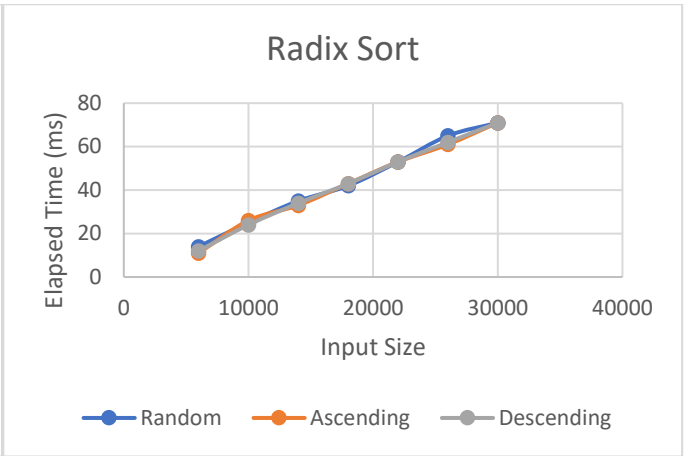
Analysis of Radix Sort
Array Size Elapsed time
Random Arrays
6000 14ms
10000 25ms
14000 35ms
18000 42ms
22000 53ms
26000 63ms
30000 71ms
Ascending Arrays
6000 11ms
10000 24ms
14000 33ms
18000 43ms
22000 53ms
26000 61ms
30000 71ms
Descending Arrays
6000 12ms
10000 24ms
14000 34ms
18000 43ms
22000 53ms
26000 62ms
30000 71ms

Question 3:









From the experimental results, I observed that for selection sort, array type (random, ascending, or descending) does not affect the program's performance as expected. As it can be seen, for all array types, it has $O(n^2)$ complexity. For the merge sort, already sorted arrays (ascending or descending) have slightly better performance due to fewer swap operations. This performance was also expected because in the merge part if all the elements of one array are bigger(or smaller) than the other array's elements, we just do half of the swap operations. The other elements are copied directly to the final array without any comparisons. Since already sorted arrays provide this situation, merge sort for those has slightly better performance and already sorted arrays create the worst case for merge sort. In the worst case, it has the complexity of O(nlogn). For the quick sort, already sorted arrays (ascending or descending) have worse performance than a randomly created array. This was also expected because, in the partition part, the first element is selected as the pivot. For already sorted arrays, selecting the first element as a pivot is bad because it does not divide the array into two almost equal parts. Instead, it divides the array into two parts of size 0 and (arraySize – 1). For descending arrays, the situation is worst than ascending arrays because, at the end of the partition section, we need to move the pivot to its correct position (between newly created two arrays). However, for ascending arrays, since the first element is the smallest, we don't have to move it. Thus, already sorted arrays(especially descending arrays) are the worst case for quick sort, and they have the complexity $O(n^2)$, while randomly created arrays have the complexity of O(nlogn). For the radix sort, array type also does not affect the performance since we don't make any comparisons. Instead, we group them in by their digits, and because of this, it has O(n) complexity.