Mehmet Onur Uysal

22002609

CS202-Sec1

Homework 1

Question 1

a) Show that $f(n) = 8n^4 + 5n^3 + 7$ is O(n^5) by specifying appropriate c and n0 values in Big-O definition

We need to find two positive constants: **c** and **n_0** such that:

$0 \leq f(n) = 8n^4 + 5n^3 + 7 \leq cn^5$ for all $n \geq n_0$

If we take c = 21 equations become equal for n0 = 1 and for the values of n0 that are greater than 1, $21n^5$ get larger much faster than $8n^4 + 5n^3 + 7$.

So, for c = 25 and n0 = 1
$0 \leq f(n) = 8n^4 + 5n^3 + 7 \leq cn^5$ for all $n \geq n_0$

Selection Sort

At the beginning all of the array is unsorted. So, the unsorted part of the array starts form the 0th index. Starting from index zero all the array is iterated to find the minimum value.

unsortedIndex = 0;

22	8	49	25	18	30	20	15	35	27

Minimum value of the array is 8 which is in the 1st index.

22	8	49	25	18	30	20	15	35	27

Minimum of the unsorted part (8) is replaced with the first index of the unsorted part.

unsortedIndex = 1;

8	22	49	25	18	30	20	15	35	27

Now, the minimum of the unsorted section is 15 so, it is swapped with 22 which is in the first index of the unsorted section of the array.

8	22	49	25	18	30	20	15	35	27

Now the unsorted section starts from the second index of the array. New minimum is 18 so it is swapped with 49 which is in the left most index of the unsorted section

unsortedIndex = 2;

8	15	49	25	18	30	20	22	35	27

New minimum of the unsorted section is 20. It is swapped with 25 which is in the unsorted index.

unsortedIndex = 3;

8	15	18	25	49	30	20	22	35	27

New minimum is 22 which is in index 7. So, it is swapped with unsorted index in order to get it to it's place.

unsortedIndex = 4;

8	15	18	20	49	30	25	22	35	27

Now, the minimum of the unsorted section is found 25 which is in 6th index. It is swapped with the item in the unsorted index.

unsortedIndex = 5

8	15	18	20	22	30	25	49	35	27

New minimum of the unsorted section is 27. So, it is swapped with the item in the unsorted index.

unsortedIndex = 6;

8	15	18	20	22	25	30	49	35	27

New minimum of the unsorted section is 30. It is swapped with the item in the unsorted index.

unsortedIndex = 7;

8	15	18	20	22	25	27	49	35	30

New minimum of the unsorted section is 35 which is already at the right place. The algorithm on the slides does not check for this situation so, in practice it is swapped with itself and the unsorted index is incremented.

unsortedIndex = 8;

8	15	18	20	22	25	27	30	35	49

If the algorithm has worked correctly, last number must be on the right place since it is the maximum value of the array. So, algorithm is done and the array is sorted.

unsortedIndex = 9;

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

Bubble Sort

Array is again divided into sorted and unsorted sections. However, bubble sort creates a sorted section of two elements at each step and carries the max element to the last.

Also, a boolean variable “sorted” is used for determining if the array is already sorted and avoiding unnecessary steps.

Maximum n passes are made over the array to make it sorted at worst case.

Sorted = true; // Array is considered sorted until an unsorted section is found.

Pass = 1;

Index = 0;

22 > 8. The numbers are swapped. (sorted = false)

22	8	49	25	18	30	20	15	35	27

8	22	49	25	18	30	20	15	35	27

Index = 1;

22 < 49. Nothing happens, section is already sorted.

8	22	49	25	18	30	20	15	35	27

8	22	49	25	18	30	20	15	35	27

Index = 2;

49 > 25. They are swapped.

8	22	49	25	18	30	20	15	35	27

8	22	25	49	18	30	20	15	35	27

Index = 3;

49 > 18. They are swapped.

8	22	25	49	18	30	20	15	35	27

8	22	25	18	49	30	20	15	35	27

Index = 4;

49 > 30. They are swapped.

8	22	25	18	49	30	20	15	35	27

8	22	25	18	30	49	20	15	35	27

Index = 5;

49 > 20. They are swapped.

8	22	25	18	30	49	20	15	35	27

8	22	25	18	30	20	49	15	35	27

Index = 6;

15 < 30. They are swapped.

8	22	25	18	30	20	49	15	35	27

8	22	25	18	30	20	15	49	35	27

Index = 7;

35 < 49. They are swapped.

8	22	25	18	30	20	15	49	35	27

8	22	25	18	30	20	15	35	49	27

Index = 8;

27 < 49. They are swapped.

8	22	25	18	30	20	15	35	49	27

8	22	25	18	30	20	15	35	27	49

sorted = true; // Array is considered sorted until an unsorted section is found.

pass = 2;

index = 0;

8 < 22. Nothing happens, section is already sorted.

8	22	25	18	30	20	15	35	27	49

8	22	25	18	30	20	15	35	27	49

index = 1;

22 < 25. Nothing happens, section is already sorted.

8	22	25	18	30	20	15	35	27	49

8	22	25	18	30	20	15	35	27	49

index = 2;

25 > 18. They are swapped. (sorted = false;)

8	22	25	18	30	20	15	35	27	49

8	22	18	25	30	20	15	35	27	49

index = 3;

25 < 30. Nothing happens, section is already sorted.

8	22	18	25	30	20	15	35	27	49

8	22	18	25	30	20	15	35	27	49

index = 4;

30 > 20. They are swapped.

8	22	18	25	30	20	15	35	27	49

8	22	18	25	20	30	15	35	27	49

index = 5;

30 > 15. They are swapped.

8	22	18	25	20	30	15	35	27	49

8	22	18	25	20	15	30	35	27	49

index = 6;

30 < 35. Nothing happens, section is already sorted.

8	22	18	25	20	15	30	35	27	49

8	22	18	25	20	15	30	35	27	49

index = 7;

35 > 27. They are swapped.

8	22	18	25	20	15	30	35	27	49

8	22	18	25	20	15	30	27	35	49

sorted = true; // Array is considered sorted until an unsorted section is found.

pass = 3;

index = 0;

8 < 22. Nothing happens, section is already sorted.

8	22	18	25	20	15	30	27	35	49

8	22	18	25	20	15	30	27	35	49

index = 1;

22 > 18. They are swapped.

8	22	18	25	20	15	30	27	35	49

8	18	22	25	20	15	30	27	35	49

index = 2;

22 < 25. Nothing happens, section is already sorted.

8	18	22	25	20	15	30	27	35	49

8	18	22	25	20	15	30	27	35	49

index = 3;

25 > 20. They are swapped.

8	18	22	25	20	15	30	27	35	49

8	18	22	20	25	15	30	27	35	49

index = 4;

25 > 15. They are swapped.

8	18	22	20	25	15	30	27	35	49

8	18	22	20	15	25	30	27	35	49

index = 5;

22 < 25. Nothing happens, section is already sorted.

8	18	22	20	15	25	30	27	35	49

8	18	22	20	15	25	30	27	35	49

index = 6;

30 > 27. They are swapped.

8	18	22	20	15	25	30	27	35	49

8	18	22	20	15	25	27	30	35	49

sorted = true; // Array is considered sorted until an unsorted section is found.

pass = 4;

index = 0;

8 < 18. Nothing happens, section is already sorted.

8	18	22	20	15	25	27	30	35	49

8	18	22	20	15	25	27	30	35	49

index = 1;

18 < 22. Nothing happens, section is already sorted.

8	18	22	20	15	25	27	30	35	49

8	18	22	20	15	25	27	30	35	49

index = 2;

22 > 20. They are swapped. (sorted = false;)

8	18	22	20	15	25	27	30	35	49

8	18	20	22	15	25	27	30	35	49

index = 3;

22 > 15. They are swapped.

8	18	20	22	15	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

index = 4;

18 < 22. Nothing happens, section is already sorted.

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

index = 5;

25 < 27. Nothing happens, section is already sorted.

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

sorted = true; // Array is considered sorted until an unsorted section is found.

pass = 5;

index = 0;

8 < 18. Nothing happens, section is already sorted.

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

index = 1;

18 < 20. Nothing happens, section is already sorted.

8	18	20	15	22	25	27	30	35	49

8	18	20	15	22	25	27	30	35	49

index = 2;

22 > 15. They are swapped. (sorted = false;)

8	18	20	15	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

index = 3;

20 < 22. Nothing happens, section is already sorted.

8	18	15	20	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

index = 4;

22 < 25. Nothing happens, section is already sorted.

8	18	15	20	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

sorted = true; // Array is considered sorted until an unsorted section is found.

pass = 6

index = 0;

8 < 18. Nothing happens, section is already sorted.

8	18	15	20	22	25	27	30	35	49

8	18	15	20	22	25	27	30	35	49

index = 2;

22 > 15. They are swapped. (sorted = false;)

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

At this point the array is sorted. However, algorithm has to finish this pass and make another pass to check and confirm that the array is actually sorted.

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

sorted = true; // Array is considered sorted until an unsorted section is found.

pass = 7

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

8	15	18	20	22	25	27	30	35	49

Sorted = true at the and of the pass so, the array is sorted and the algorithm stops without making the last pass.

Question 2

c) Sorting the array {9, 6, 7, 16, 18, 5, 2, 12, 20, 1, 16, 17, 4, 11, 13, 8}

```
Initial Array
{ 9, 6, 7, 16, 18, 5, 2, 12, 20, 1, 16, 17, 4, 11, 13, 8 }

-------- Insertion Sort -------
{ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 }
Number of key comparisons : 69
Number of data moves : 88
---------------------------

-------- Bubble Sort -------
{ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 }
Number of key comparisons : 110
Number of data moves : 174
---------------------------

-------- Merge Sort -------
{ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 }
Number of key comparisons : 47
Number of data moves : 128
---------------------------

-------- Quick Sort -------
{ 1, 2, 4, 5, 6, 7, 8, 9, 11, 12, 13, 16, 16, 17, 18, 20 }
Number of key comparisons : 50
Number of data moves : 125
---------------------------
```

d) Performance Analysis

Using random arrays

Analysis of Insertion Sort

Array size	Elapsed Time	compCount	moveCount
5000	24 ms	6180381	6185383
10000	93 ms	25003650	25013660
15000	186 ms	56545220	56560230
20000	318 ms	100107502	100127508
25000	498 ms	156927036	156952043
30000	717 ms	224312219	224342224
35000	971 ms	304313988	304348997
40000	1261 ms	398672705	398712714

Analysis of Bubble Sort

Array size	Elapsed Time	compCount	moveCount
5000	105 ms	12494725	18526155
10000	443 ms	49978710	74980986
15000	1013 ms	112487550	169590696
20000	1804 ms	199970694	300262530
25000	2866 ms	312473805	470706135
30000	4125 ms	449969775	672846678
35000	5673 ms	612480847	912836997
40000	7380 ms	799950597	1195898148

Analysis of Merge Sort

Array size	Elapsed Time	compCount	moveCount
5000	2 ms	55250	123616
10000	4 ms	120451	267232
15000	5 ms	189396	417232
20000	7 ms	260890	574464
25000	9 ms	334037	734464
30000	12 ms	408493	894464
35000	14 ms	484377	1058928
40000	15 ms	561800	1228928

Analysis of Quick Sort

Array size	Elapsed Time	compCount	moveCount
5000	1 ms	84796	130738
10000	2 ms	150404	253443
15000	3 ms	247245	447996
20000	3 ms	314140	495275
25000	4 ms	437987	644850
30000	5 ms	517122	897860
35000	7 ms	630932	968971
40000	6 ms	718896	1064699

Using almost sorted arrays

Analysis of Insertion Sort

Array size	Elapsed Time	compCount	moveCount
5000	3 ms	800579	805578
10000	11 ms	3157707	3167706
15000	25 ms	7402175	7417174
20000	35 ms	10895355	10915354
25000	63 ms	19439075	19464074
30000	93 ms	29095955	29125954
35000	113 ms	35309151	35344150
40000	125 ms	39878593	39918592

Analysis of Bubble Sort

Array size	Elapsed Time	compCount	moveCount
5000	42 ms	12431070	2386740
10000	165 ms	49803110	9443124
15000	382 ms	112258914	22161528
20000	645 ms	198913222	32626068
25000	1051 ms	312322475	58242228
30000	1535 ms	449751414	87197868
35000	2020 ms	608563900	105822456
40000	2469 ms	770061020	119515782

Analysis of Merge Sort

Array size	Elapsed Time	compCount	moveCount
5000	2 ms	50245	123616
10000	3 ms	111050	267232
15000	4 ms	174556	417232
20000	7 ms	236196	574464
25000	9 ms	307701	734464
30000	10 ms	378403	894464
35000	12 ms	436723	1058928
40000	13 ms	483082	1228928

Analysis of Quick Sort

Array size	Elapsed Time	compCount	moveCount
5000	1 ms	223362	188497
10000	2 ms	415697	464486
15000	4 ms	829042	803740
20000	5 ms	1215805	1087764
25000	9 ms	1849672	1834669
30000	9 ms	2060903	1603045
35000	16 ms	4745806	1774841
40000	65 ms	28659982	1759455

Using almost unsorted arrays

Analysis of Insertion Sort

Array size	Elapsed Time	compCount	moveCount
5000	37 ms	11782891	11787910
10000	153 ms	46893601	46903636
15000	334 ms	105339721	105354732
20000	589 ms	188620459	188640476
25000	950 ms	293371704	293396720
30000	1349 ms	421632475	421662498
35000	1812 ms	576536272	576573514
40000	2415 ms	759436406	759483648

Analysis of Bubble Sort

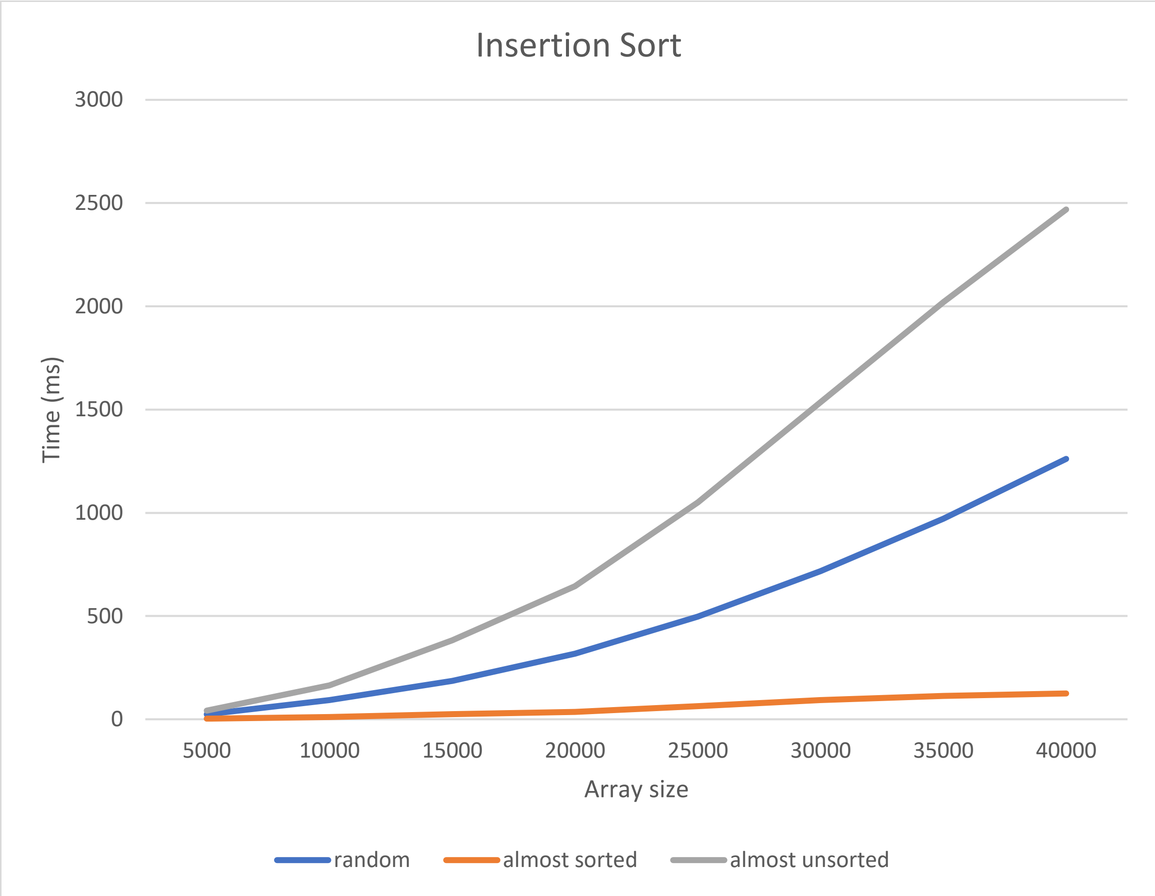
Array size	Elapsed Time	compCount	moveCount
5000	114 ms	12497500	35333733
10000	463 ms	49995000	140650911
15000	1041 ms	112492500	315974199
20000	1859 ms	199990000	565801431
25000	2895 ms	312487500	880040163
30000	4178 ms	449985000	1264807497
35000	5713 ms	612482500	1729510545
40000	7381 ms	799980000	-2016756349

Analysis of Merge Sort

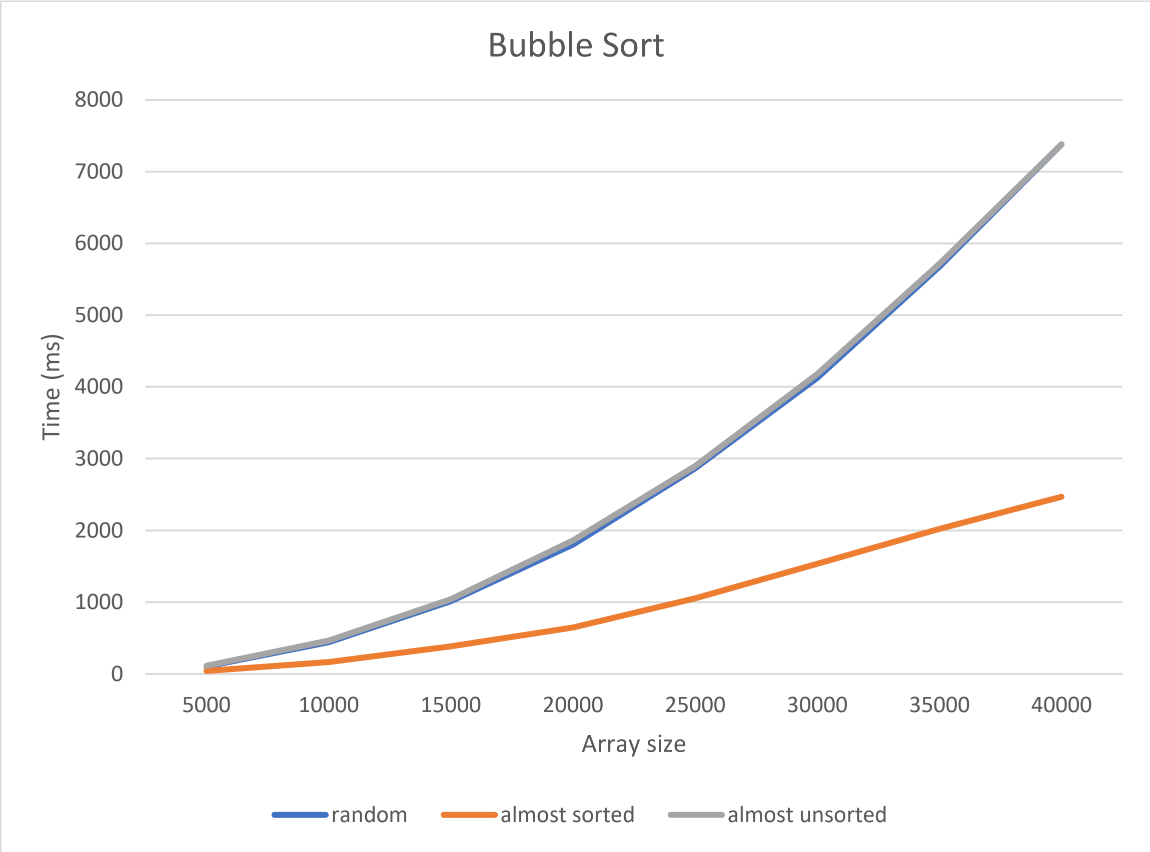
Array size	Elapsed Time	compCount	moveCount
5000	2 ms	49086	123616
10000	4 ms	108409	267232
15000	5 ms	172293	417232
20000	7 ms	236996	574464
25000	9 ms	306866	734464
30000	10 ms	377286	894464
35000	11 ms	436065	1058928
40000	13 ms	484921	1228928

Analysis of Quick Sort

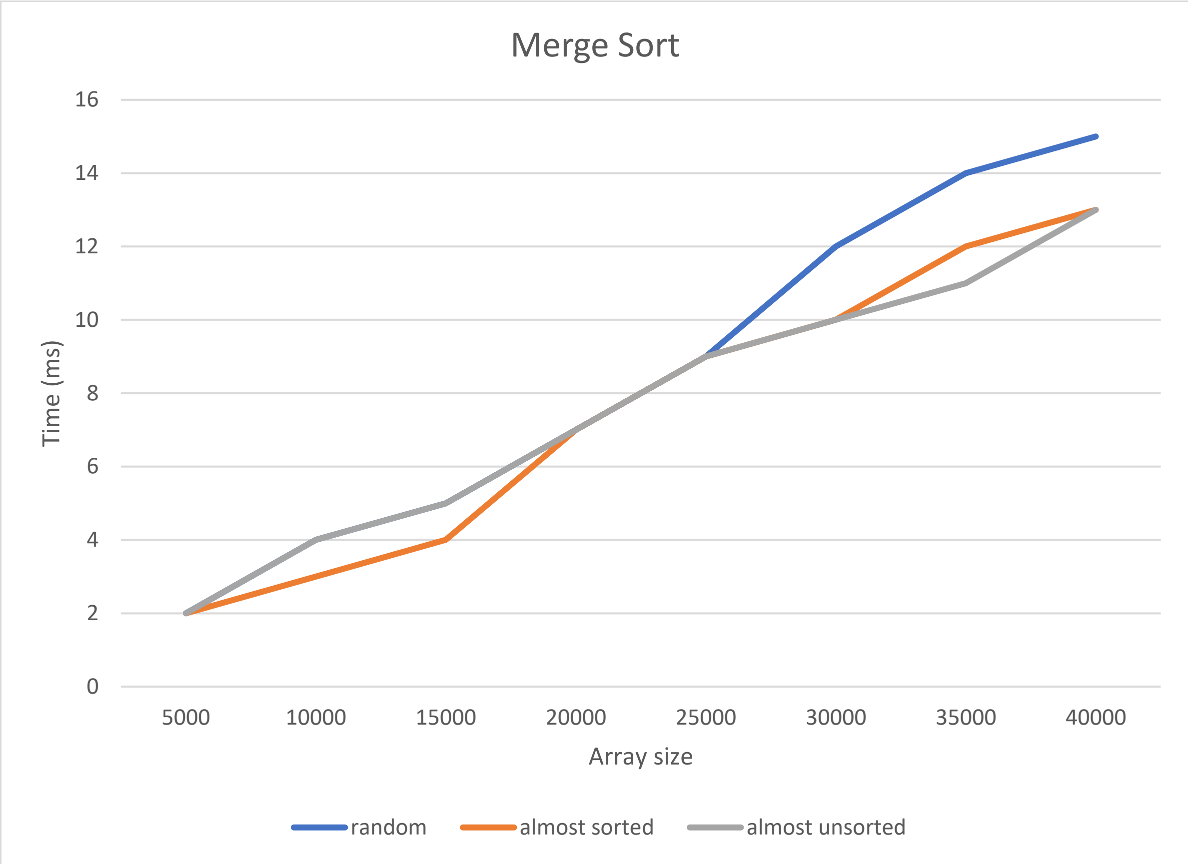
Array size	Elapsed Time	compCount	moveCount
5000	1 ms	146770	233727
10000	3 ms	369029	587200
15000	3 ms	404936	667194
20000	3 ms	563431	895096
25000	6 ms	827977	1344050
30000	7 ms	949980	1553177
35000	22 ms	3554923	5458597
40000	150 ms	27273627	41061418



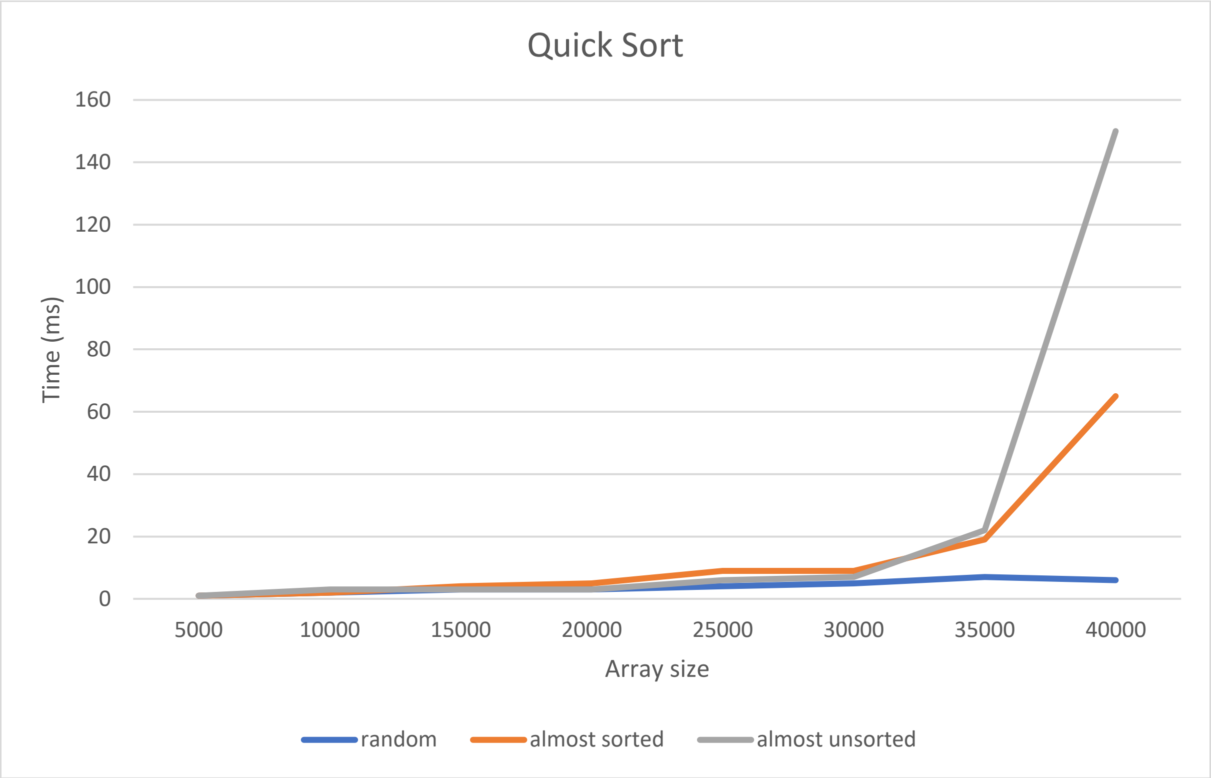
Insertion sort iterates over the array to find the max or min value and puts it into its place. It is a relatively simple sorting algorithm which has theoretical speed O(n^2). Results show that insertion sort is not so fast. It is clear from the graph that results of the random array and the almost unsorted array has quadratic curve which is in parallel to theoretical assumptions. Worst case of the algorithm is when the array is in reverse order (O(n^2)) and the best case is when the array is already sorted (O(n)). This behavior is seen in the graph. Process becomes faster and more linear when the array is more similar to sorted version.



Bubble sort algorithm iterates over the array and swaps each consecutive number if necessary, until the array is sorted. Graph of the results shows that the time it takes the algorithm to sort the almost sorted algorithm is significantly lower than the other two cases. This is the result of the fact that bubble sort's best case is when the array is already in the ascending order. When the array is already sorted or almost sorted, algorithm makes only a few iterations to sort the array. In this case the complexity of the algorithm is O(n). However, when the array is unsorted or in the reverse order, the complexity of the algorithm is O(n^2). This is the reason that graphs of random array sorting and almost unsorted array sorting results produce quadratic curves.



Merge sort was significantly faster than bubble sort and insertion sort at each case. Because merge sort has similar results for best, worst and average cases, results were similar for random, almost sorted and almost unsorted arrays. Time complexity of the algorithm is O(logn*n) for average and worst case. This makes the algorithm very fast and efficient. However, merge sort requires an extra array whose size is equal to the size of the original array. So, merge sort may not be the best choice if the memory is limited despite its efficiency.



Quick sort was again very fast like merge sort. The algorithm has theorical speed O(logn*n) for the best case and the average case and the wort case is O(n^2). Choosing the pivot determines the efficiency of the algorithm. Since we modified the algorithm to choose the first item in the array to be the pivot, worst case happens when the array is sorted or in the reverse order. This can be seen in the graph. Results of the random array are lower and more consistent than the almost sorted and almost unsorted array results. Although the worst case is not so efficient, average case is much more efficient than the worst case which makes quick sort one of the best sorting algorithms. Also, quick sort does not require any extra arrays like merge sort, so it is also better for memory use.