CMPE 223/242
Programming Homework #2
“Sorting Algorithms”

Author:
Rahymberdi Annyyev

Experimental Setup/Procedure

In our experiment, we will guess 5 sorting algorithms, which methods implementations are hidden in the .jar file provided by the university. This is a classic example of API (Application Programming Interface). We will test each sorting algorithm by increasing the array size we will sort by:

$$2*2*2*\ldots*2^k$$

We will test the sorting methods by 3 different arrays of the same size: ascending, descending, and random array. Each sorting will be completed in each k iteration of the main for-loop that increases the sample size of the array. The reader can look at the code that will test these sorting methods in the project folder, `SortingAlgorithmsTester.java`

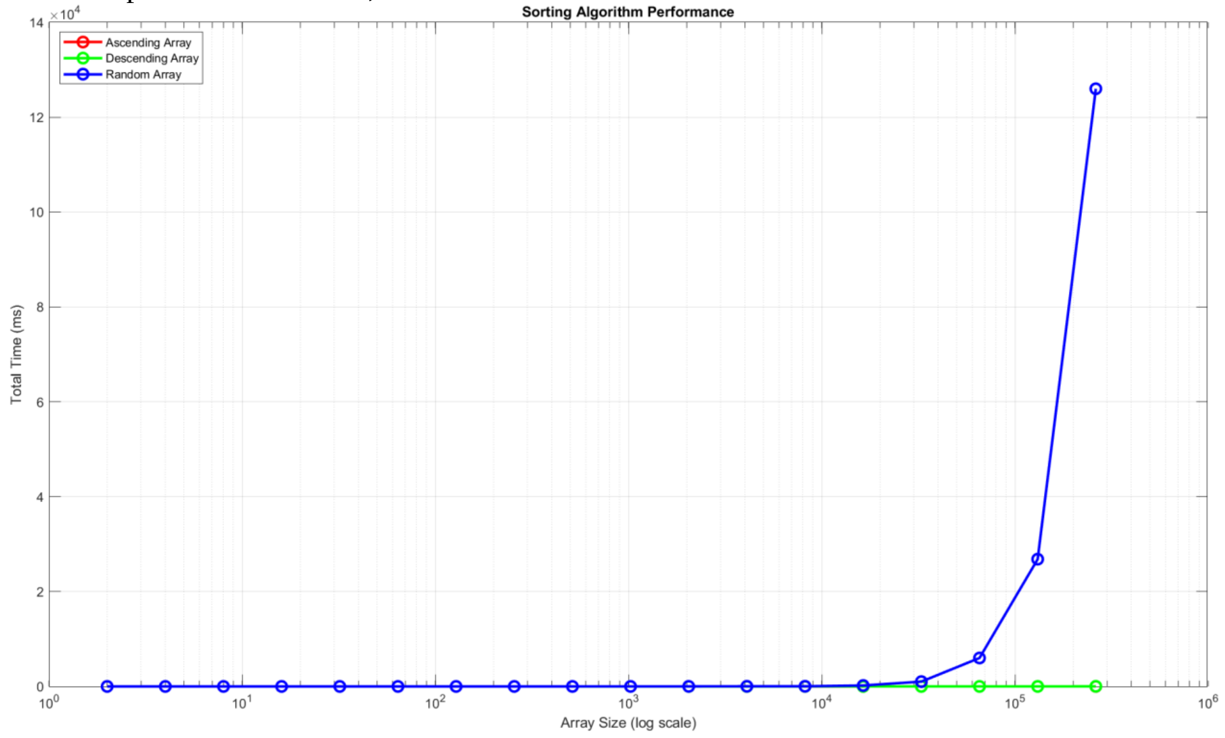
These methods were executed on Windows 11, with usable RAM of 16 GB. Tables were made in **Excel**®. Graphs were made in **MATLAB**®.

Experimental Results

1. For our first sorting method, `sort1()`, we have these results:

Array Size	Ascending Time (ms)	Descending Time (ms)	Random Time (ms)
2	1	0	0
4	0	0	0
8	0	0	0
16	0	1	0
32	0	0	0
64	0	0	0
128	0	0	0
256	0	0	0
512	0	0	1
1024	0	0	1
2048	0	1	13
4096	1	1	31
8192	0	1	30
16384	1	1	200
32768	1	1	1014
65536	1	1	5997
131072	2	2	26819
262144	4	18	125984

And if we plot these numbers, we will have this:



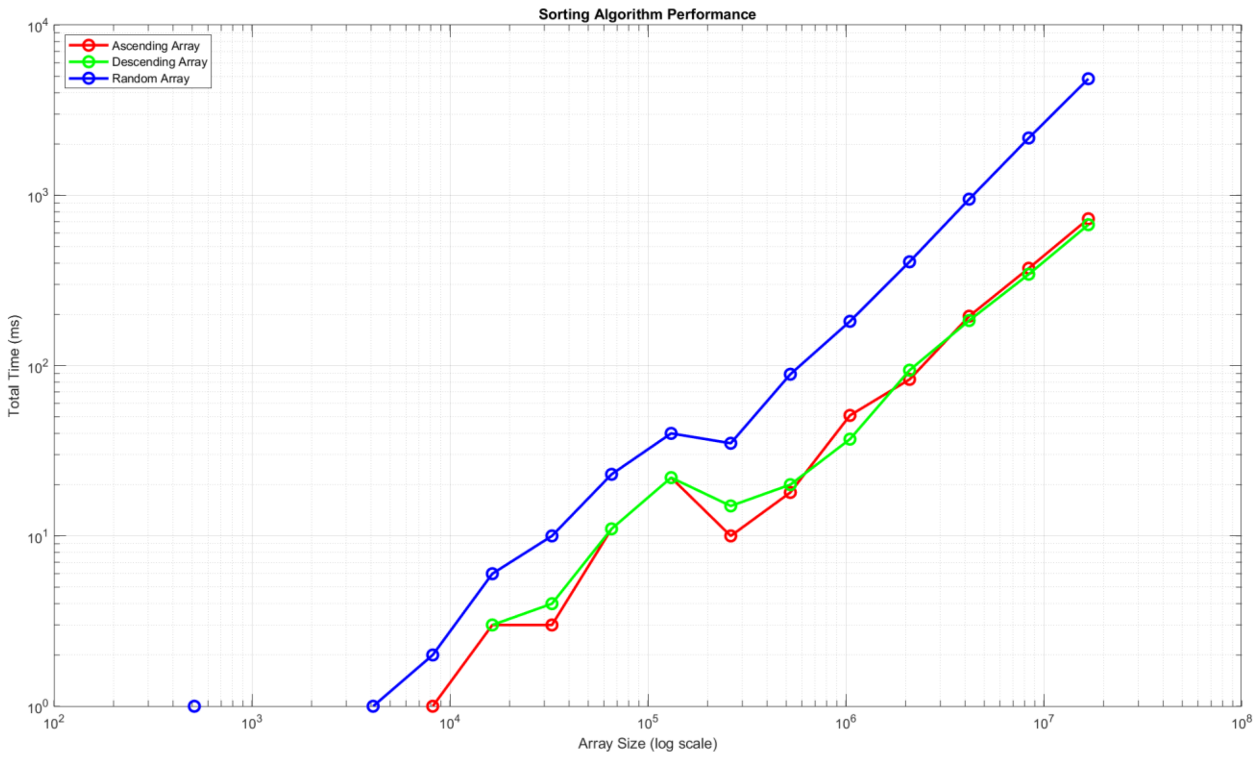
Analysis: If we analyze the plot and its progression, we can see that for ordered (descending and ascending) arrays, it executes the sorting in relatively O(1) time, regardless of the size of the array. But for randomized arrays, we can see the spike of time execution when the array becomes big enough, and progression seems to be of O(n^2). So, if we would take a guess, it is obvious the sorting algorithm takes a "pair" approach to sorting, that is, it continuously takes passes through the array to find the minimum/maximum. That is why for this sorting algorithm, it is faster to go through/sort the ordered (either descending or ascending) arrays but struggles with randomized inputs. So, the worst-case is O(n^2), which coincides with the Insertion sort worst-case scenario [2] and the average/best-case is O(1). I would suggest that this is either Insertion or Bubble sort because both have similar time complexities. But I would choose Insertion sort because of the mechanism of how it swaps the data in the array, which makes the randomized arrays time-consuming to implement.

Answer: Insertion Sort.

2. For our first sorting method, `sort2()`, we have these results:

Array Size	Ascending Time	Descending Time	Random Time
2	0	0	0
4	0	0	0
8	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0
128	0	0	0
256	0	0	0
512	0	0	1
1024	0	0	0
2048	0	0	0
4096	1	0	1
8192	3	3	2
16384	3	4	6
32768	11	11	10
65536	22	22	23
131072	10	15	40
262144	18	20	35
524288	51	37	89
1048576	83	94	182
2097152	195	184	407
4194304	373	344	949
8388608	728	672	2170

And if we plot these numbers, we will have this:



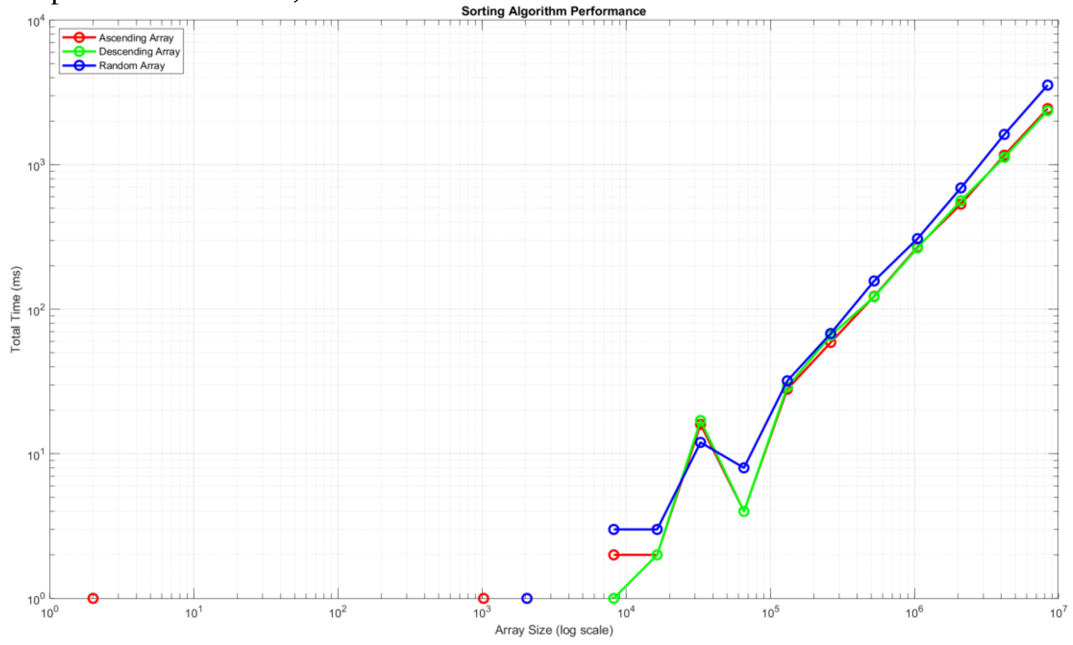
Analysis: If we analyze the plot and its progression, we can observe that ordered (descending and ascending) arrays, execute relatively at the same time, while randomized arrays take + k for some constant k in comparison to ordered ones, with similar slopes. For smaller arrays, it doesn't take a lot of time to sort, but for bigger ones, it takes time that is very close to linear progression. So, I would suggest that for best/average scenarios, it has O(n*log(n)) time complexity. But if we consider the gap between randomized and ordered sorting execution time, I will say in the worst-case scenario, it becomes very close to O(n^2), which coincides with Quick sort worst-case time complexity [1]. So, we can choose between Quick sort and Merge sort, but because of the last fact, it seems that it is Quick sort because it relies heavily on the positions of pivots and the balance of partitions. If we had generally inefficient partitions due to the positions of pivots, it would slow down the Quick sort significantly. So, I would say this sorting method implements Quick Sort.

Answer: Quick Sort.

3. For our first sorting method, `sort3()`, we have these results:

Array Sizes	Ascending Times (ms)	Descending Times (ms)	Random Times (ms)
2	1	0	0
4	0	0	0
8	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0
128	0	0	0
256	0	0	0
512	0	0	0
1024	1	0	0
2048	0	1	1
4096	0	2	3
8192	2	17	3
16384	2	4	12
32768	16	29	8
65536	4	66	32
131072	28	122	68
262144	59	266	157
524288	123	564	308
1048576	270	1125	691
2097152	534	2368	1621
4194304	1164	3556	3556
8388608	2447	2368	3556

And if we plot these numbers, we will have this:



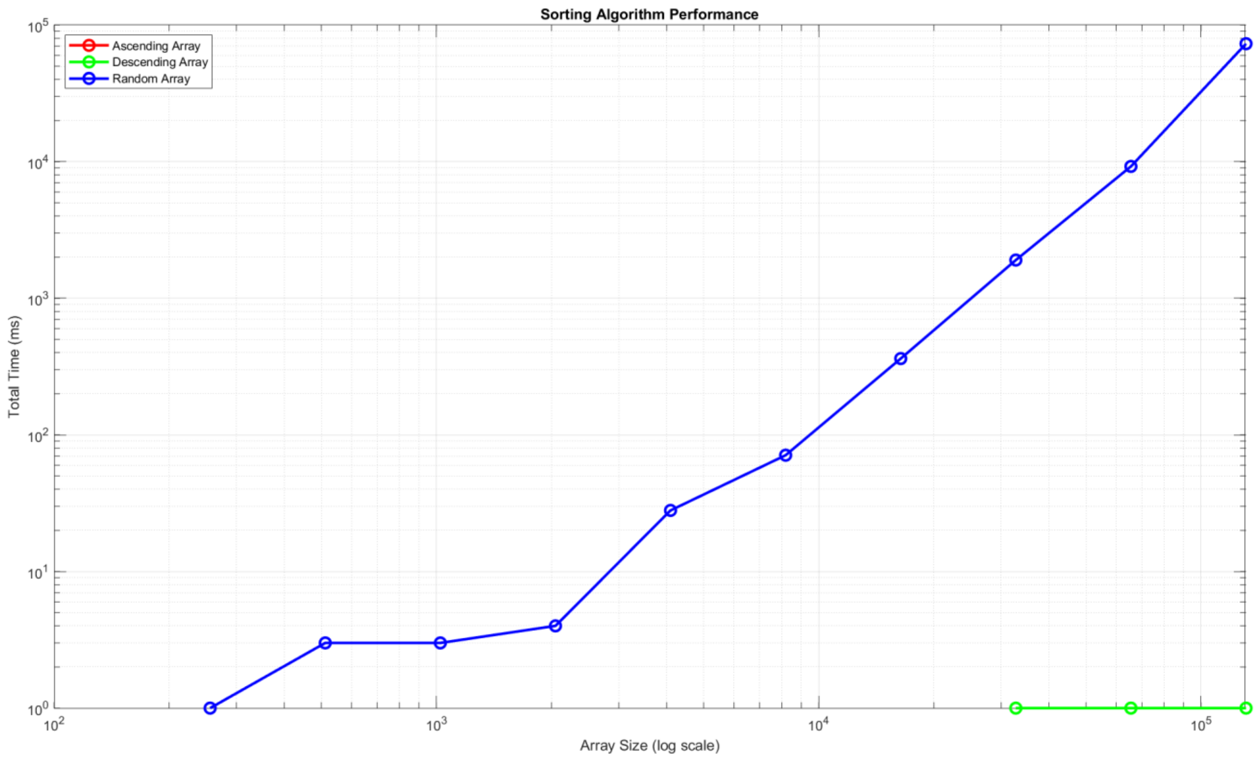
Analysis: If we analyze the plot and its progression, we can observe that ordered (descending and ascending) and randomized arrays execute relatively at the same time. For smaller arrays, it takes minimal time to sort, and for bigger ones, it takes time that is very close to linear progression too, as previous `sort2()`. So, I would suggest that for best/average and worst-case scenarios, it has O(n*log(n)) time complexity. So, because we already choose between Quick sort and Merge sort, this sorting method implements Merge Sort, which also agrees with its cases of time complexities [3].

Answer: Merge Sort.

4. For our first sorting method, `sort4()`, we have these results:

Array Sizes	Ascending Times (ms)	Descending Times (ms)	Random Times (ms)
2	0	0	0
4	0	0	0
8	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0
128	0	0	0
256	0	0	1
512	0	0	3
1024	0	0	3
2048	0	1	4
4096	0	0	28
8192	0	0	71
16384	0	0	362
32768	0	1	1904
65536	1	1	9231
131072	1	1	73022

And if we plot these numbers, we will have this:



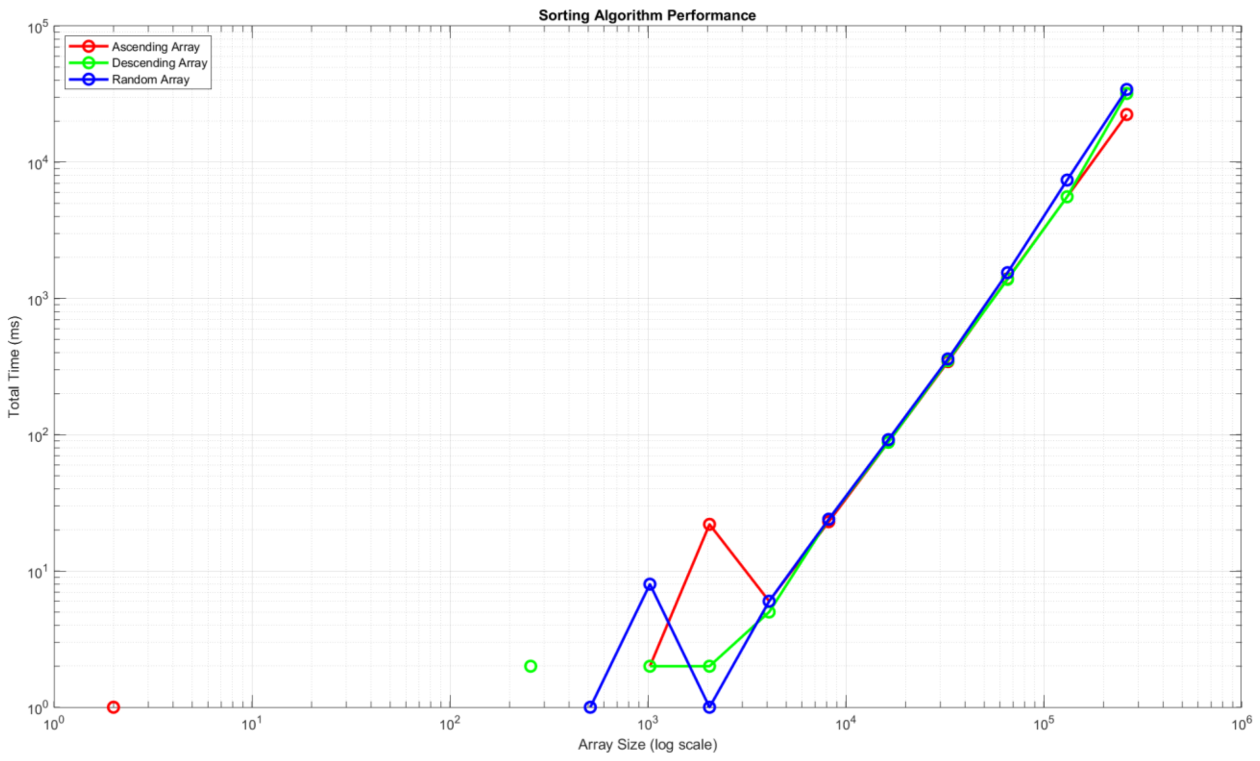
Analysis: If we analyze the plot and its progression, we can observe that ordered (descending and ascending) takes minimal time for execution, which is close to O(1) time complexity. For randomized arrays, it executes relatively at O(n^2) time complexity progression. This data coincides with the results we had with `sort1()`, which suggests that it is either Insertion sort or Bubble sort. Because we already chose Insertion sort, I would suggest that this sorting algorithm implements Bubble sort, which agrees with its time complexities [4].

Answer: Bubble Sort.

4. For our first sorting method, `sort5()`, we have these results:

Array Size	Ascending Time	Descending Time	Random Time
2	1	0	0
4	0	0	0
8	0	0	0
16	0	0	0
32	0	0	0
64	0	0	0
128	0	0	0
256	0	2	0
512	0	0	1
1024	2	2	8
2048	22	2	1
4096	6	5	6
8192	23	24	24
16384	88	88	92
32768	344	350	360
65536	1395	1381	1547
131072	5583	5581	7402
262144	22401	31966	34219

And if we plot these numbers, we will have this:



Analysis: Out of all the sorting methods we covered, this one is the most inefficient, if we observe its slope. The last sorting method that wasn't assigned is Selection sort, so we could say straight away that this is Selection sort. But we can still analyze this practice method. If we analyze the plot and its progression, we can observe that in ordered (descending and ascending) and randomized arrays, the sorting method executes relatively at O(n^2) time complexity progression regardless. So, it is indeed a Selection sort, which agrees with its time complexities scenarios [5].

Answer: Selection Sort.

Conclusion

Sort 1: Insertion Sort

Sort 2: Quick Sort

Sort 3: Merge Sort

Sort 4: Bubble sort

Sort 5: Selection Sort

Credits

1) SoftwareTestingHelp. (2024, March 7). QuickSort In Java – Algorithm, Illustration & Implementation. Retrieved from [QuickSort In Java - Algorithm, Example & Implementation (softwaretestinghelp.com)].
2) Alake, R. (2021, December 9). Insertion sort explained: A data scientist's algorithm guide. Retrieved from [Insertion Sort Explained–A Data Scientists Algorithm Guide | NVIDIA Technical Blog].
3) GeeksforGeeks. (2024, April 8). Merge Sort - Data Structure and Algorithms Tutorials. Retrieved from here.
4) StudySmarter. (2024, April 8). Bubble Sort - Algorithms in Computer Science. Retrieved from here.
5) Simplilearn. (n.d.). Selection Sort Algorithm. Simplilearn. https://www.simplilearn.com/tutorials/data-structure-tutorial/selection-sort-algorithm