

CSCE 221 Cover Page  
Programming Assignment #4

**Submission Deadlines:** Fri Nov 15, 11:59pm (5 extra credits), Mon Nov 18, 11:59pm (submit without penalty)

First Name: Khoa

Last Name: Diep

UIN: 926005094

**Any assignment turned in without a fully completed coverpage will receive ZERO POINTS.**  
Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

| CSCE 221 Students | Other People | Printed Material | Web Material (URL)   | Other        |
|-------------------|--------------|------------------|----------------------|--------------|
| 1.                | 1.           | 1.               | 1. cplusplus.com     | 1. 221 notes |
| 2.                | 2.           | 2.               | 2. stackoverflow.com | 2. 121 notes |
| 3.                | 3.           | 3.               | 3.                   | 3.           |
| 4.                | 4.           | 4.               | 4.                   | 4.           |
| 5.                | 5.           | 5.               | 5.                   | 5.           |

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today's Date: 11/18/19

Printed Name (in lieu of a signature): Khoa Diep

## **Introduction:**

The objective of this project is to implement various sorting algorithms. To be more specific, there will be four sorting algorithms to be implemented, bubblesort, heapsort, mergesort, and quicksort. For this project, we will sort the elements using arrays, with the smaller element be towards the left of the array and the larger element be towards the right of the array. Heapsort will be sorted with a heap, which was implemented and explained in the previous project. Heapsort will have to insert an array into a heap and then removed and placed into another array. Bubblesort will be an in-place sorting algorithm that parses through the array repeatedly and compares an element with the next element and swaps if the elements are in the wrong order. Mergesort will be a divide and conquer sorting algorithm that divide the array and repeatedly merge the array to produce a ordered subarrays until it merges back into a sorted array. Quicksort will be a divide and conquer where will pick an element (could be random, front, back, etc.) and partition the array. Partitioning the array means that all the elements less than the pivot will be before the pivot and all the elements greater than the pivot will be after the pivot.

## **Theoretical Analysis:**

Let n represent the size of the array needed to be sorted.

For bubblesort we will use a nested for loop to compare the first element to the second, swapping to the correct order, and doing the same for the second element and the third element. We will do this process until we reach the second to last element. After one loop, the largest element should at the end of the array. We will do n loops and each loop will be ending with the last loops second to last element. There is a flag called swap to check if a swap ever occurred. If a swap does not occur during one of the inner loops, then we will break out of the outer for loop. The best-case scenario is while parsing through the inner loop and there are no swaps happening. This means that the input array is already sorted and the inner loop only happened once. This means the best-case time complexity will be  $O(n)$ . However, the worst-case scenario will be if there are not any breaks for the outer loop. This means that the inner loop we happen n times. Since the inner loop has a time complexity of  $O(n)$ , n inner loops we have a time complexity of  $n O(n) = O(n^2)$ . This worst-case scenario will happen if we sort a reverse-ordered array. The average case will be when will insert random items will also be  $O(n^2)$ . The worst-case space complexity will be  $O(1)$ , since it is in-place.

In conclusion -

The time complexity of sorting n ordered items using bubblesort is  $O(n)$ .

The time complexity of sorting n reversed-ordered items using bubblesort is  $O(n^2)$ .

The time complexity of sorting n random items using bubblesort is  $O(n^2)$ .

The space complexity of sorting n ordered items using bubblesort is  $O(1)$ .

For heapsort we will need to first insert n elements from the array into the heap, and then remove n elements from the same heap into an array, which will be the sorted array. Since heaps were

covered in the last project, we can conclude that the time complexity of heapsort of  $n$  elements will be the same as inserting and then removing  $n$  items into the heap which is  $O(n \log n)$ . This will be consistent whether or not a sorted array, a reverse sorted array, and a random array is sorted using heapsort, since the best-case scenario, worst-case scenario, and average-case scenario will have the same time complexity. The worst-case space complexity will be  $O(1)$ , since it is in-place.

In conclusion -

The time complexity of sorting  $n$  ordered items using heapsort is  $O(n \log n)$ .

The time complexity of sorting  $n$  reversed-ordered items using heapsort is  $O(n \log n)$ .

The time complexity of sorting  $n$  random items using heapsort is  $O(n \log n)$ .

The space complexity of sorting  $n$  ordered items using heapsort is  $O(1)$ .

For mergesort, we created a merge function that merges two subsets of the array putting them in order. We divide the unsorted array into  $n$  subarrays (part of the original array) and repeatedly merge the subarrays recursively using said merge function. For mergesort the best-case scenario, worst-case scenario, and average-case scenario will be  $O(n \log n)$ . This is because the entire input much be halved  $\log n$  times. Then merging them  $n$  element together again will take  $O(n)$  time,  $O(n) * O(\log n) = O(n \log n)$ . This means regardless of the input array needed to be sorted the time complexity of every mergesort will be  $O(n \log n)$ . The mergesort is not in place, since it needs to copy elements and sort them then add them back in so the space complexity is  $O(n)$ .

In conclusion -

The time complexity of sorting  $n$  ordered items using mergesort is  $O(n \log n)$ .

The time complexity of sorting  $n$  reversed-ordered items using mergesort is  $O(n \log n)$ .

The time complexity of sorting  $n$  random items using mergesort is  $O(n \log n)$ .

The space complexity of sorting  $n$  ordered items using mergesort is  $O(n)$ .

For quicksort, we created a partition function, where the pivot will be the last element, which at the end of the partition function all elements less than the pivot will occur before the pivot and all the element more than the pivot will occur after pivot. Then we will recursively apply the previous steps for elements less than the pivot and for the elements greater than the pivot. The worst-case scenario will happen if the there are no elements greater/lesser than the pivot for every call of the function, meaning quicksort function will just be a glorified bubble sort, having a time complexity of  $O(n^2)$ . However in the average case and the best case, the time complexity will be  $O(n \log n)$ , where the partition will separate the array into half meaning this will take  $O(\log n)$  and the partition step will have a time complexity of  $O(n)$ . For the implementations used in the project, it is in place, meaning it has a space complexity of  $O(1)$ . Since the pivot is at the end of the array, for the sorted list it will all the elements will be less than the pivot for each function call, and similarly for the reverse ordered input, all elements will be greater than the pivot, which is the worst-case scenario. For random ordered, it will be the average-case, which  $O(n \log n)$ .

In conclusion -

The time complexity of sorting  $n$  ordered items using quicksort is  $O(n^2)$ .

The time complexity of sorting  $n$  reversed-ordered items using quicksort is  $O(n^2)$ .

The time complexity of sorting  $n$  random items using quicksort is  $O(n \log n)$ .

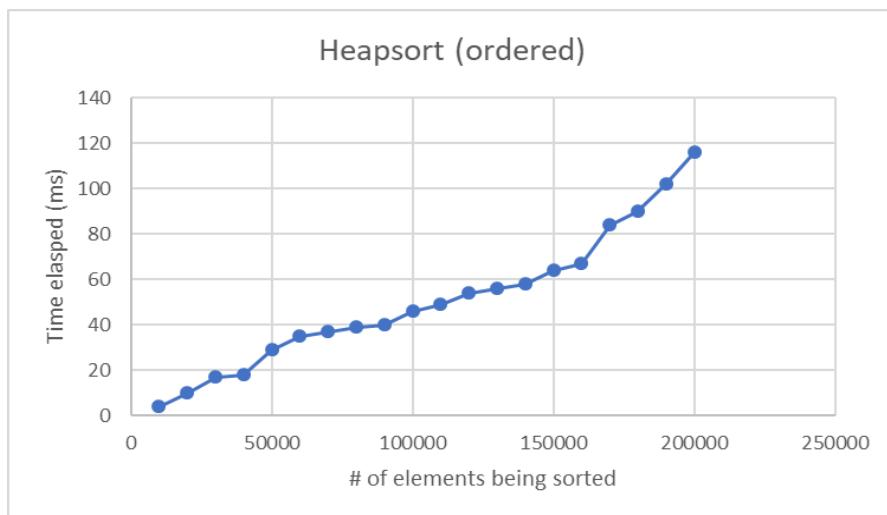
The space complexity of sorting  $n$  ordered items using quicksort is  $O(1)$ .

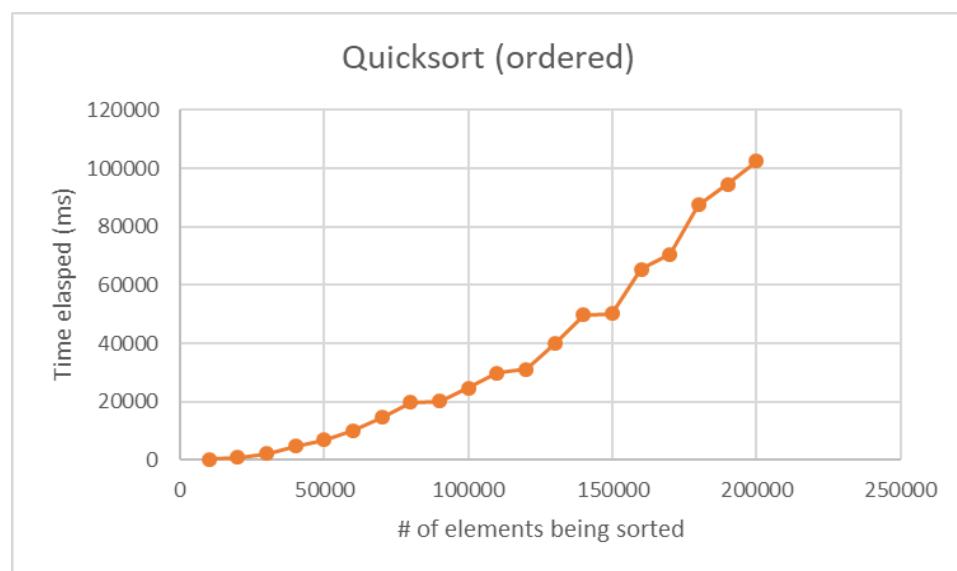
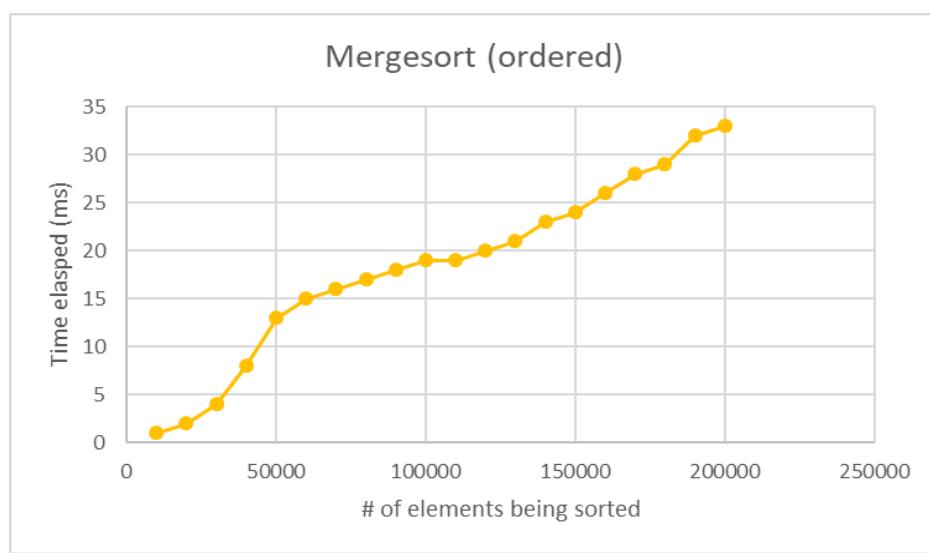
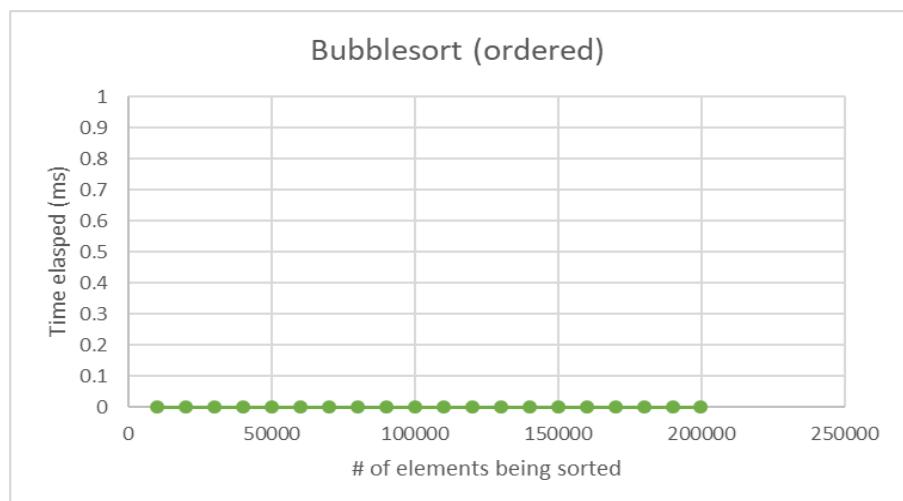
## **Experimental Setup:**

I tested on my laptop, which has an Intel Core i7 processor, 16 GB of RAM, and operating on Windows 10. An IDE called Eclipse is used to write, compile and run the project. In order to test the functions, I created arrays using a for loop, testing with all three types of inputs (sorted, reverse-sorted, and random). I then input these arrays into each implanted algorithm. Using a print function, I then print the values into the console and inspected them for accuracy. I used arrays allocated on the heap (dynamic memory) for both inputs and outputs of the sorting algorithms. I used dynamic memory because I will not have to worry about overflowing the stack (although unlikely/almost impossible for my tests but more generic for other users to use the sorting algorithms). The input sizes used are 10,000 to 200,000 to be sorted, being incremented by 10,000. These were chosen to show the differences of each sorting algorithm since at 10,000 there should be a significant difference in the time to compare each sorting algorithm and by 200,000, we will see the difference of the growth behavior of each algorithm. I repeated this experiment three times for each sorting algorithm.

## **Experimental Results:**

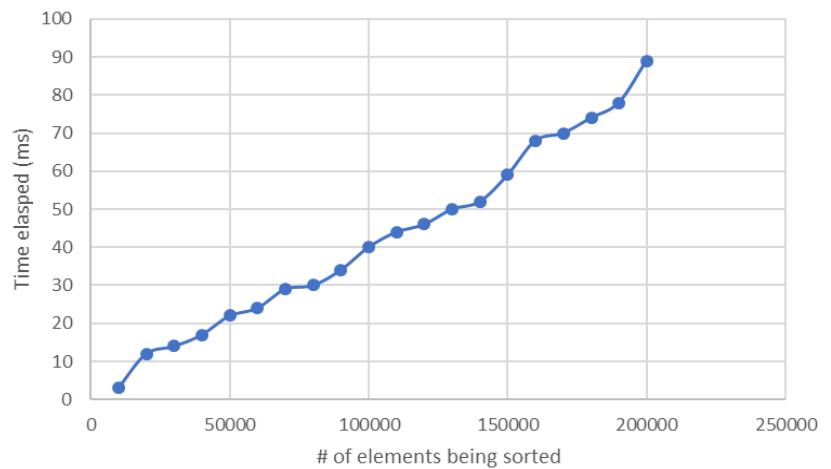
Ordered items



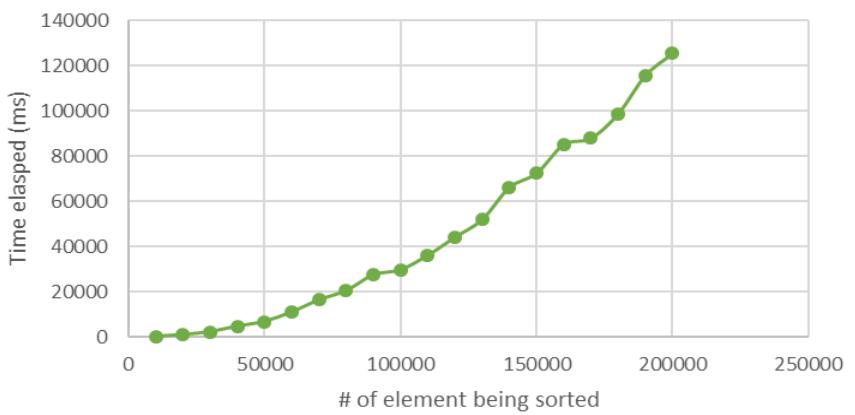


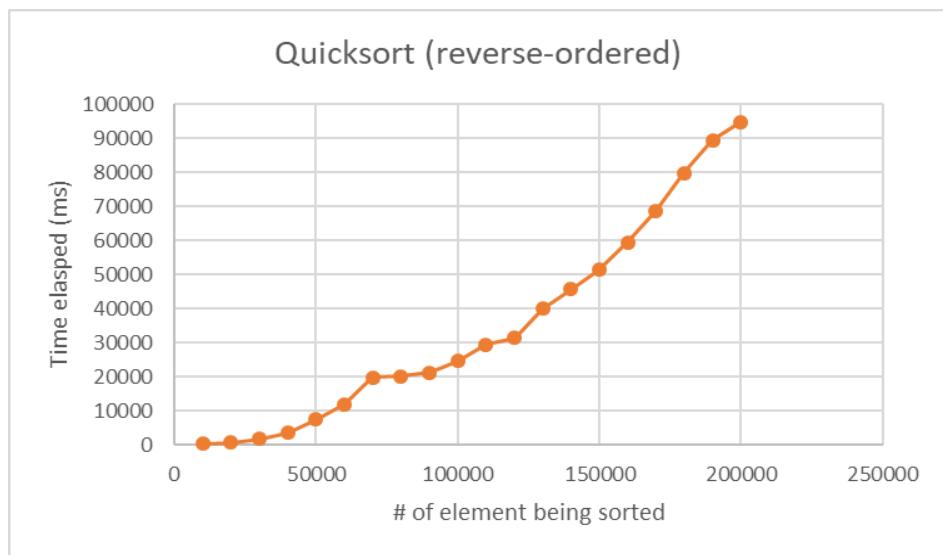
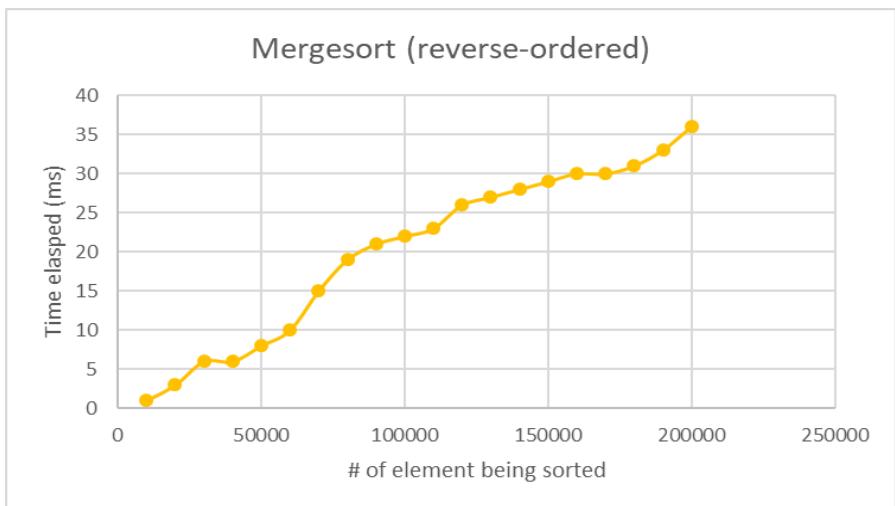
### Reverse ordered

Heapsort (reverse-ordered)

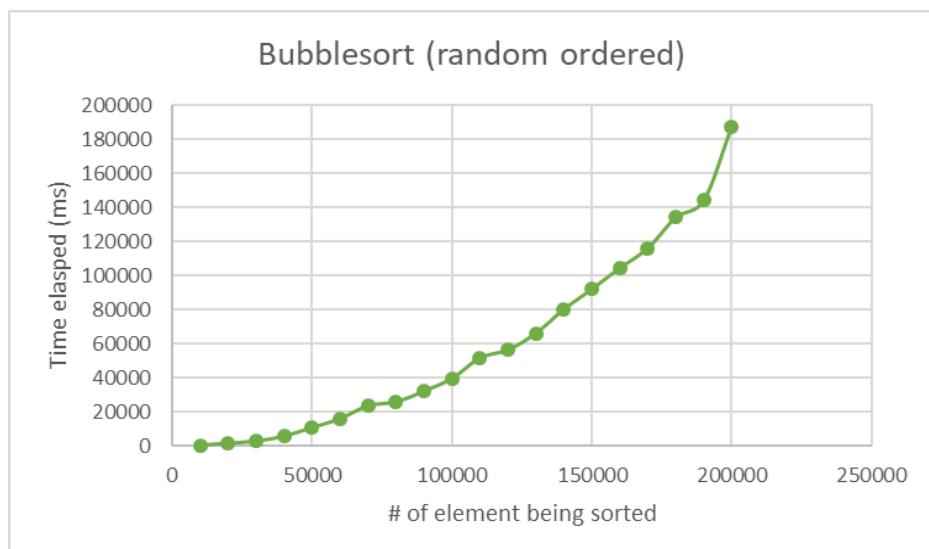
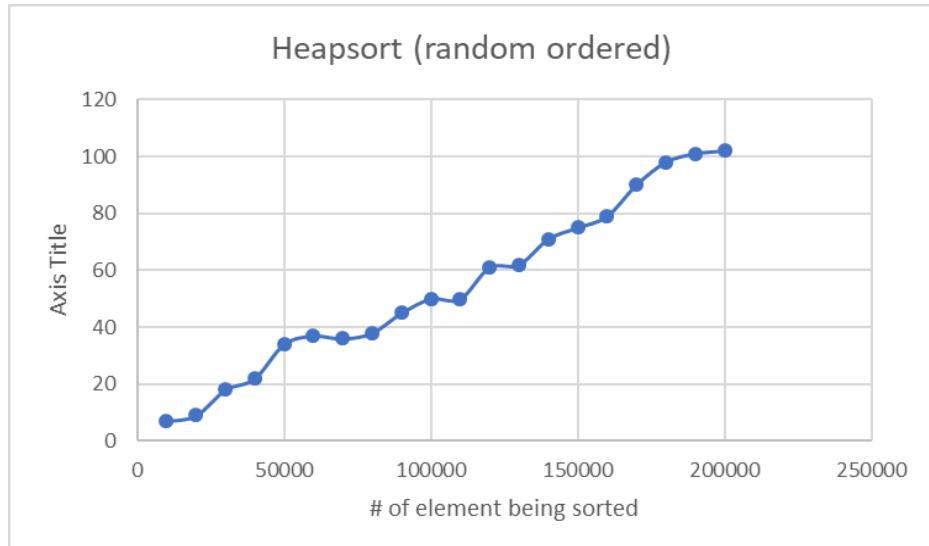


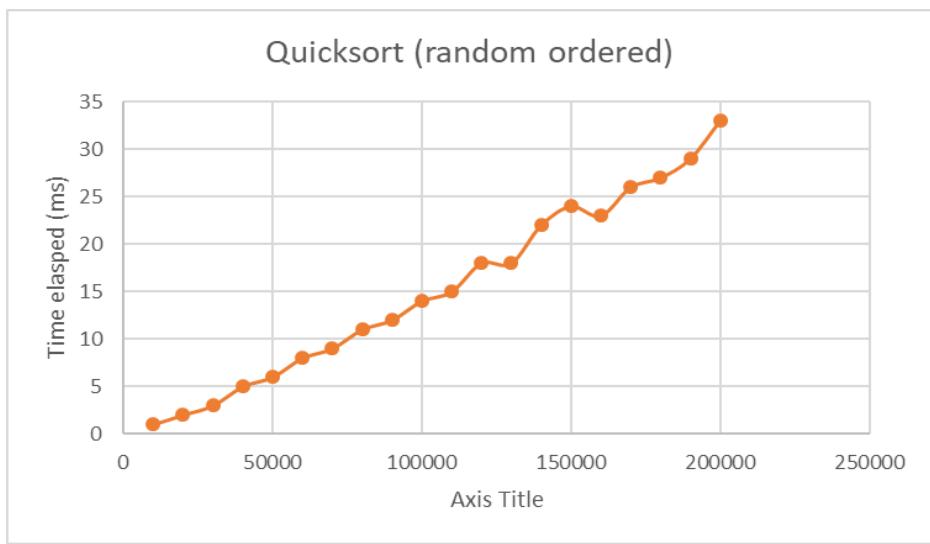
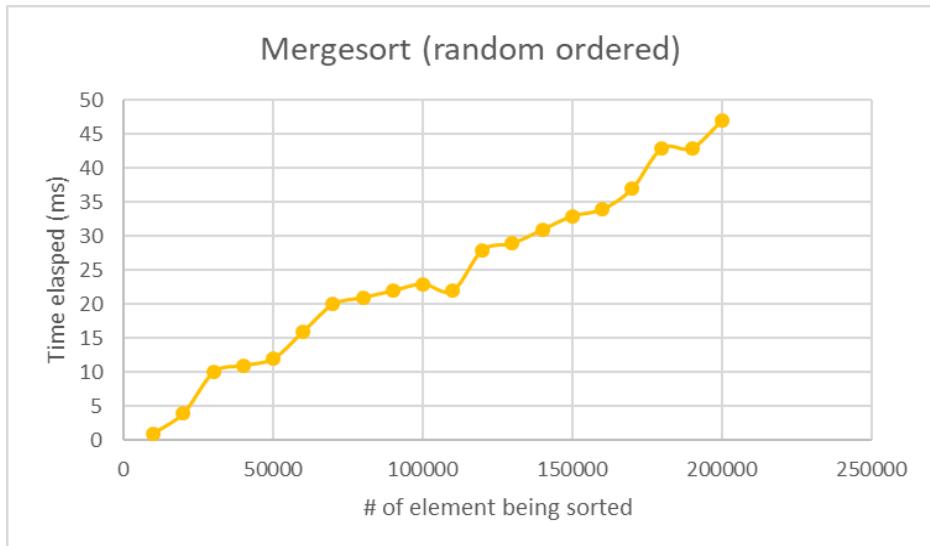
Bubblesort (reverse-order)





Random ordered





### Discussion:

In general, the mergesort performed the best out of all the sorting algorithms. The worse case scenario for both bubblesort and mergesort happens when the input array it either ordered or reverse-ordered. Mergesort performs better than heapsort marginally, because heapsort has to insert the items into a heap and remove them, where as mergesort does not need to insert any items and can start working on the sorting part right after the function is called. In general the graph prove my theoretical analysis. However, I did not expect the quicksort to perform that poorly. This is because the pivot was at the end of the array. If we change the pivot to be at the middle of the array, rather than the end of the array, the worse-case scenario would not happen for the ordered and the reverse ordered inputs and perhaps the performance could be, on average, better than mergesort and far better than bubblesort, rather than being on par with bubblesort in my implementation of mergesort.