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: Vidhur Last Name: Potluri UIN: 626007235

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. Dr. Tyagi	1. Course material	1.	1.
2.	2.	2.	2.	2.
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: 11/18/2019

Printed Name (in lieu of a signature): Vidhur Potluri

CSCE 221 505

Programming Assignment #4

Report

Vidhur Potluri

UIN: 626007235

Introduction

The aim of this assignment is to determine the run time complexity and efficiency of the different sorting algorithms. The assignment includes implementing and using the four different sorting functions – bubble sort, heap sort, merge sort, and quick sort – to sort arrays of different sizes and different types in an ascending order, and comparing the times it takes these functions to do so. Each algorithm is subject to three different types of input arrays: sorted, reverse sorted, and randomly sorted.

Theoretical analysis

An analysis of the complexity of each of the four different sorting algorithms is provided below.

a. Bubble sort: The bubble sort algorithm sorts through the input array and swaps two elements that are next to each other if the successor is smaller than the predecessor. In order for the sorting to end, the function needs to pass through the array without swapping any of the elements. A Boolean variable, therefore, would help the function determine whether the array is sorted. The first time the function passes through the array, it would make (n-1) comparisons. The second pass would have (n-2) comparisons and so forth. The total number of comparisons would be:

$$n - 1 + n - 2 + \cdots + 2 + 1 = \frac{n(n+1)}{2}$$

Therefore, the average and worst-case runtime complexity is

$$O(n^2)$$

An already sorted array would have a runtime complexity of O(n) since there is no swapping involved.

b. Heap sort: The heap sort algorithm sorts through the input array by creating a heap of the elements of the array using upheap. The elements are then removed and inserted back into the array using the removeMin() function of a heap, which uses downheap to look for the smallest number in the heap. We know that the height of a binary tree is log(n). The heapsort makes multiple log(n) comparisons and swaps both during the insertion and removal processes. Since the sorting process involves swapping the root element with the latest element being added to the heap, the insertion of n elements would lead to a worst, best and average runtime complexity of -

$$O(nlogn)$$

c. Merge sort: The merge sort algorithm sorts through the input array by recursively dividing the array into halves until the array consists of only 1 element and then merging the arrays back to form a sorted array. The calculation of the midpoint takes O(1) time. Since the division of the array represents a binary tree of height log(n), and therefore log(n) comparisons and the merging of n arrays of 1 element each takes O(n) runtime, the worst case and average runtime complexity of the merge sort algorithm is -

$$O(nlogn)$$

However, a sorted array as input would have a runtime of O(n).

d. Quick sort: The runtime for the quick sort algorithm to sort through an array of 1 element is -

$$T(1) = 1 \qquad (base\ case)$$

The runtime for an array of n elements is -

$$T(n) = T(n-1) + n$$

Therefore,

$$T(n) = T(n-k) + kn - \frac{(k-1)k}{2}$$

Since $n - k = 1$,

$$T(n) = \frac{n^2 + n}{2}$$

$$O(n) = n^2$$

The best case runtime complexity would be when the sort algorithm divides and conquers similar to the merge sort. Therefore, the worst-case and average case runtime complexities would be

$$O(nlogn)$$

Experimental setup

Machine Specifications:

Computer model: HP Spectre x360

OS: Windows 10 Home

Processor: Intel® Core™ i5-8250U CPU @ 1.60 GHz 1.80 GHz

Installed Memory: 8.00 GB (7.84 GB Usable)

Procedure and testing:

Arrays were used as arguments for all of the sorting functions. The sorted arrays for each sorting function were created using a for loop with an iterator that was used as both the index of the array and the number in that index (array = {0,1,2,3,4...n}). The reverse sorted arrays were created using a for loop as well. The iterator began at n and decremented until it reached a value of 0. The random array's elements were generated using the rand() function from the 'stdlib.h' library. The array sizes were different for the different sorting algorithms because of their runtime complexities and depending on the type of array used, some of the sorting algorithms took significantly longer than others. Each function was called twice to ensure the runtime calculations were accurate. The sizes and the arguments of the different functions are listed below:

a. Bubblesort(): The largest array sorted by the function has a size of 10000 since unsorted input arrays of larger size take significantly longer than sorted input arrays. The size of the array to be sorted is increased by 100 every time. A while loop is used to increase the size of the array each time.
b. heapsort(): The largest array sorted by the function has a size of 100000 since smaller sizes were sorted by the functions in milliseconds and their runtimes were too small to be compared with other functions' runtimes. The size of the array to be

sorted is increased by 1000 every time. A while loop is used to increase the size of the array each time.

c. mergesort(): The largest array sorted by the function has a size of 100000 since smaller sizes were sorted by the functions in milliseconds and their runtimes were too small to be compared with other functions' runtimes. The size of the array to be sorted is increased by 1000 every time. A while loop is used to increase the size of the array each time.

d. quicksort(): The largest array sorted by the function has a size of 10000 since sorted input arrays of larger size take significantly longer than random input arrays. The size of the array to be sorted is increased by 1000 every time. A while loop is used to increase the size of the array each time.

Experimental results

The runtimes could only be calculated to a 2 decimal place precision. This is why some of the graphs seem to be oscillating between two points. However, the general trend can still be observed for the runtime graphs. The following are the runtimes of the different sorting functions -

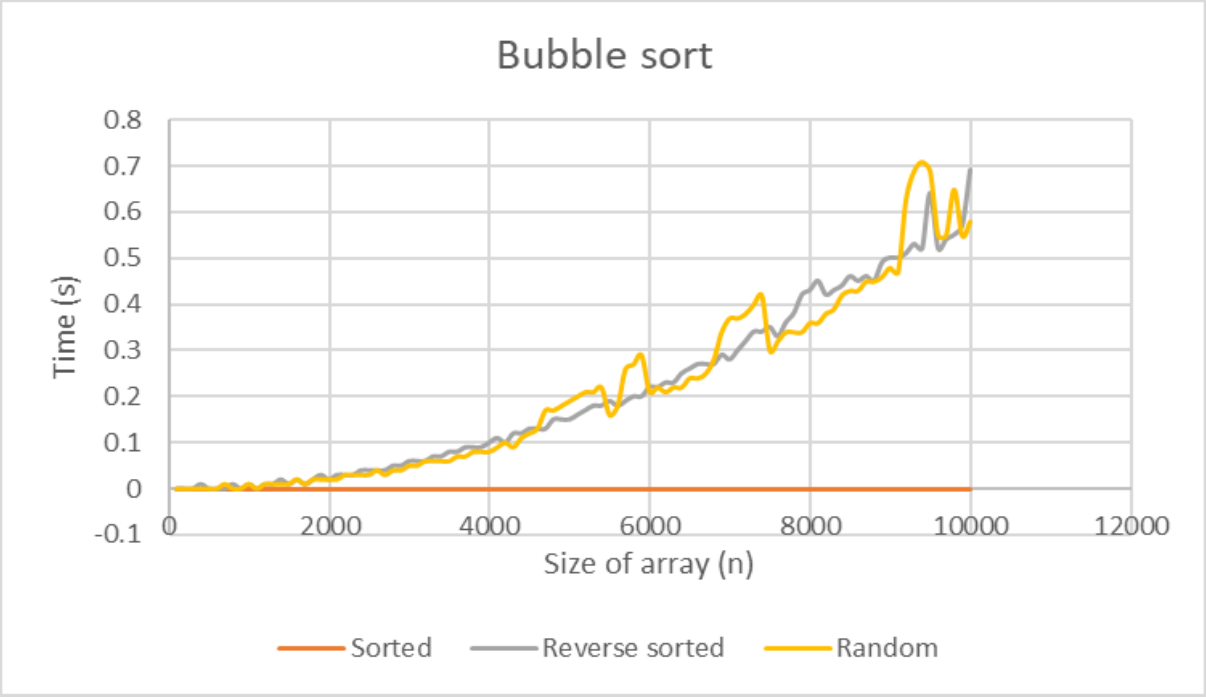

Fig 1 – Bubble sort

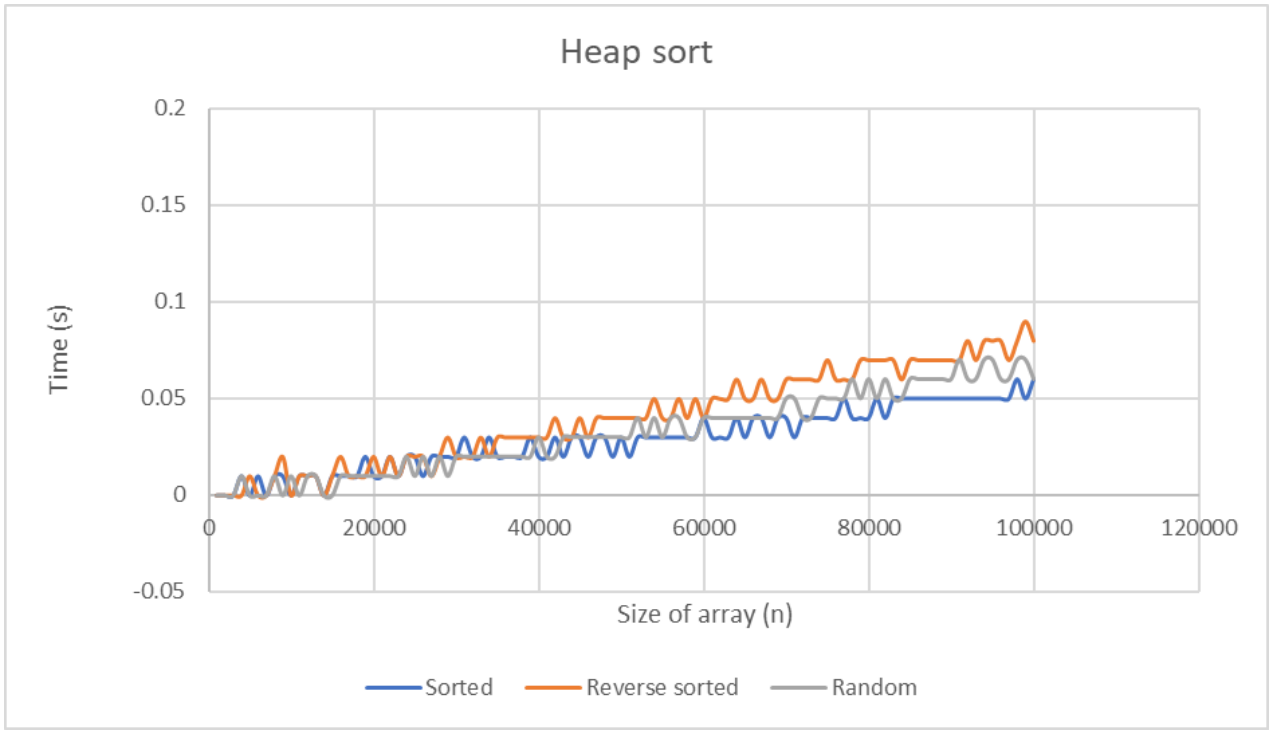

Fig 2 – Heap sort

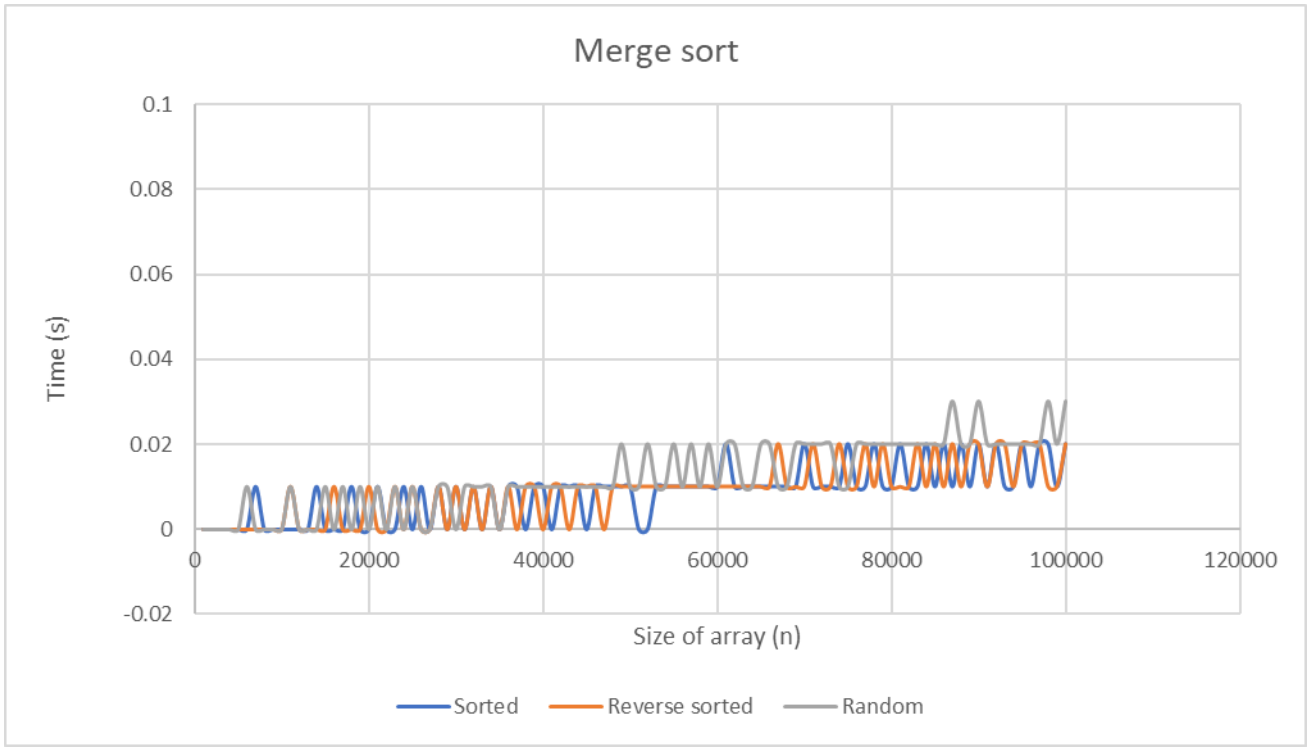

Fig 3 – Merge sort

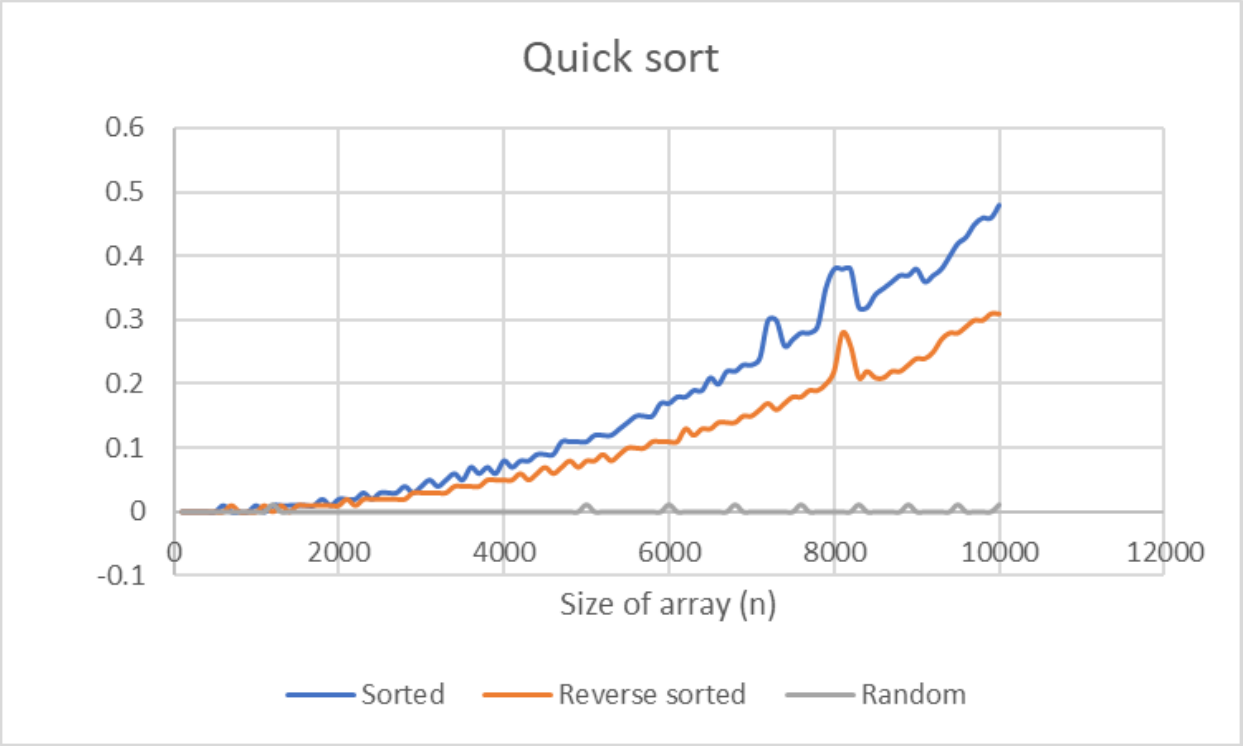

Fig 4 – Quick sort

Discussion:

a. Sorted array: The bubble sort, merge sort and heap sort are much quicker than the quicksort when a sorted array is used as an argument. The sorted array corresponds to the best-case scenario of the bubble sort, heap sort and merge sort algorithms. All three of these sorting algorithms run as predicted. The merge-sort function's best case run time complexity was calculated to be $O(n)$ which means it should theoretically be quicker than the heap sort which has a runtime complexity of $O(nlogn)$ for all scenarios. From the graphs, we can see that the experimental data is consistent with this prediction as the merge sort is slightly faster than the heap sort. The quicksort takes significantly longer than the other three sorting functions because of its worst-case runtime complexity of $O(n^2)$. (heap sort and merge sort used similar array sizes; bubble sort and quick sort used similar array sizes).
b. Reverse order sorted array: The reverse order array could be considered the average case scenario for all of the sorting algorithms. The bubble sort graph is a quadratic function and is consistent with our theoretical analysis. It is slower than the quick sort function whose graph is approximately a function whose runtime complexity is

$O(n\ log\ n)$. The merge sort and heap sort both have $O(n\ log\ n)$ runtime complexities and the data observed agrees with this. They have very similar runtimes.

c. Random input array: The quick sort, merge sort and heap sort perform better than the bubble sort. The bubble sort's graph corresponds to $O(n^2)$ as opposed to $O(nlogn)$ for the other three. All of the graphs are consistent with the runtime complexities calculated earlier in the theoretical analysis section.

The random input array could be considered the most common array that needs sorting. Therefore, its runtime analysis would give the user a better idea of which sorting mechanism they would like to use. The quicksort function is very quick with the random input array but performs very poorly when it is used to sort an already sorted array. The merge sort and heap sort have very similar run times. The merge sort has a slight edge because it has better runtime when it sorts an already sorted array. I would consider the merge sort to be the ideal because it has a best-case run time complexity of O(n) (same as the bubble sort but quicker than the heap sort and quick sort) and an average runtime complexity of $O(nlogn)$ (same as the heap sort and quick sort but more efficient than the bubble sort). The slight discrepancies that we have observed could be because of the low precision mechanism we have used for runtime calculation, and changing frequency of the processor of the computer.