REPORT: PROJECT 4

Muhammad Taha Haqqani
Section 505

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: **MUHAMMAD TAHA** Last Name: **HAQQANI** UIN: **426004967**

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.wikipedia	1.
2.	2.	2.	2.stackOverflow	2.
3.	3.	3.	3.youtube	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: 11/19/2019

Printed Name (in lieu of a signature): **Muhammad Taha Haqqani**

INTRODUCTION:

- Implement bubble sort, Heap sort, Merge sort, and Quick Sort algorithms and analyze their performances
- Measure the sorting time for each algorithm by testing with sorted, randomly sorted and reverse sorted inputs
- Discuss each algorithm in terms of time complexity, space complexity, stability and other factors that need to be taken into account when choosing a sorting algorithm
- Analyze the results using graphs and compare the time complexity for each algorithm

THEORETICAL ANALYSIS

<u>Bubble Sort:</u>

Bubble sort is a naïve and simple sorting algorithm. It is also called an internal exchange sort. It works by comparing two adjacent elements in a sequence and if they are not in order, these two elements are swapped. The process is repeated for each element in the sequence until all the elements are in order.

To sort every element in the array, the time complexity of bubble sort is O(n). This means that if only one element needs to be sorted, there will only be one swap required and this will only need one iteration. The worst-case time complexity for bubble sort is $O(n^2)$. This happens when every element of the array is unordered and needs to be swapped and the number of times it needs to iterate is also equal to the number of elements present. Therefore, for n number of elements, the sorting time complexity is equal to O(n*n) which is equal to $O(n^2)$. The case when this can happen is when the array is reversely sorted.

The best-case time complexity of bubble sort is when the array is already sorted. This means that there will not be any swaps needed and the time complexity will be O(1). For n number of elements, this will mean O(1*n) which is equal to O(n).

Advantages:

- Easy to understand and implement
- In-place sorting algorithm which means that memory usage is minimal and it does not require additional space
- Stable sorting algorithm so it will maintain the relative order of elements
- Suitable for sorting just one element or nearly sorted arrays

Disadvantages:

- Highly inefficient because of the poor running time complexity of $O(n^2)$
- Unoptimized bubble sort will loop through the entire array even when it is completely sorted

Type of Input and expected performance:

- Sorted: O(n) (optimized)
- Reversely-Sorted: $O(n^2)$
- Randomly-sorted: $O(n^2)$

<u>Heap Sort:</u>

A heap is a binary tree that stores elements in its internals nodes while complying to the heap order property in which the parent node is always smaller or equal to its children. The binary tree must also be complete which means that all levels must be filled and nodes should be added from left to right. The leaves do not have to be completely filled but every other level must be filled. A min heap stores the smallest element at its root whereas a max heap stores the largest element at its root. In this experiment, we will use Min Heap. Heap sort works by extracting the minimum value from the heap and moving it to the beginning of the array. This is done until all the elements are removed from the heap into the array resulting in a sorted array.

For one down-heap operation, as we observed in the previous assignment, the time required was O(log n) because there are log n levels considering that n elements were inserted into the heap. For every swap the time taken is just O(1), therefore, the total time complexity for each removal is just O(1*log n) which is O(log n). Hence, for n removals, the total time required will be O(n log n) and this will be the total sorting time for heap sort.

Advantages:

- Efficient and consistent logarithmic time complexity of O(n log n)
- Suitable for sorting large data sets
- Does not require recursion or creation of multiple arrays
- Optimal performance for every data set
- In-place sorting algorithm which means that memory usage is minimal and does not need additional space

Disadvantages:

- Unstable sorting algorithm so the relative order of elements in not maintained
- Slowest of the O(n log n) sorting algorithms

Type of Input and expected performance:

- Sorted: O(n log n)
- Reversely-Sorted: O(n log n)
- Randomly-sorted: O(n log n)

<u>Merge Sort:</u>

Merge Sort is a divide and conquer algorithm and works by recursively cutting an array into two halves (almost equal) until only one element is left, and then merging all those sub arrays together in order.

There are three parts of the merge sort that need to be analyzed. The first part is the divide part in which the array is split in two halves. This takes constant O(1) time because we are just finding the mid and populating the two halves which is done through direct array indexing. The second part is the conquer part in which we recursively sort the halves. In this step, we recursively break the array into halves until we reach a size of 1. This means that if we have n elements, we will have a total of log n levels. The last step is merge in which we merge all the n elements which will take O(n) time. Since we have to merge log n levels, the total time complexity of merge sort will be O(n log n).

Advantages:

- Efficient and consistent logarithmic time complexity of O(n log n), slightly faster than heap sort
- It is a stable sorting algorithm so relative order in maintained
- Suitable for storing large data sets

Disadvantages:

- Not in-place so it requires additional memory for each sub array
- Less efficient than quick sort
- Requires a lot of recursions

Type of Input and expected performance:

- Sorted: O(n log n)
- Reversely-Sorted: O(n log n)
- Randomly-sorted: O(n log n)

Quick Sort:

Quick sort is also a divide and conquer algorithm. It works by selecting one of the elements as a pivot and partitioning the left and right side of the array in a way that all the elements greater than the pivot are inside the right side and the elements smaller than the pivot are in the left side of the pivot.

The time complexity for quick sort depends on the selection of the pivot and the type of input. Since we are partitioning the array into subarrays, the best case is if we have two nearly equal subarrays. This is because it will only require $\log_2 n$ partitions and hence for n elements, the total time complexity will be O(n log n). The average case is also the same as the best-case because the number of partitions will always be log n for random data. The problem with quick sort arises when we encounter a scenario where the partition obtained is always equal to n-1. This occurs when the pivot is either the smallest or largest element in the list or all the elements are repeated. If this happens in every partition call, there will be n-1 calls on n number of elements and this will result in the worst-case time complexity of $O(n^2)$.

Advantages:

- In place sorting algorithm
- More efficient than merge sort for smaller data sets
- Preferred for arrays
- Exhibits good cache locality

Disadvantages:

- In worst-case scenario, the time complexity is $O(n^2)$
- May not work well with large data sets

- Unstable sorting algorithm
- Difficult to implement and improve

Type of Input and expected performance:

- Sorted: O(n^2) (deterministic approach: last element is pivot)
- Repeated elements: O(n^2)
- Reversely-Sorted: O(n log n)
- Randomly-sorted: O(n log n)

Summary:

Sorting Algorithm	Time Complexity	Space Complexity	Stable?	In-Place?
Bubble Sort	Best Case: O(n) Worst Case: O(n^2)	O(1)	Yes	Yes
Heap Sort	All Cases: O(n log n)	O(1)	No	Yes
Merge Sort	All Cases: O(n log n)	O(n)	Yes	No
Quick Sort	Best Case: O(n log n) Worst Case: O(n^2) Average Case: O(n log n)	O(1)	No	Yes

EXPERIMENTAL SETUP

Machine Specifications:

- Processor: Intel(R) Core (TM) i7-7500U-CPU @2.70 GHz 2.90 GHz
- RAM: 8.00 GB
- System Type: 64-bit Operating System

Implementation:

- Bubble Sort

Bubble sort is implemented by looping through the array and comparing each element with its next element. To make sure we are within the bounds of the array, we have to loop from start to end-1 so we can compare with the last element without running out of bounds. There is an if statement within this loop that executes swapping of the two consecutive elements if the element is greater than the next element. These swaps happen until the whole array is sorted. To optimize the performance, a Boolean variable is made which will stop the looping of the array once the whole array is sorted. This happens when there are no swaps in the array.

- Heap Sort

The heap sort algorithm uses the heap data structure which was implemented in the previous assignment. The function works by creating a heap using bottom up heap construction function. Once the heap is created, elements are removed from the heap and inserted back into the array that needs to be sorted. Since this is a minheap, the elements are inserted in ascending order and the loop is terminated till all the items are inserted.

- Merge Sort

In this divide and conquer algorithm, the basic idea is that the original array is broken down into subarrays until we are left with one element. Then the subproblems are merged together in sorted order. We first find the midpoint of the original array and create two subarrays, left and right, which will have equal or almost equal size depending on whether the size of the original array is even or odd. Then, the elements less than or equal to midpoint are copied into the left array and the elements greater than the midpoint are copied into the right array from the original array. Then, we recursively call merge sort on the left and right arrays until we have one element left. After the subarrays are built, the merge function is called. This function will have the original array, left array and right array, and their sizes as parameters. Elements will be inserted from the left or right array into the original array by comparing the elements of the both arrays and choosing the smallest elements first. Once the elements are inserted into the original array, the array is sorted and the algorithm will work on the other subarrays.

- Quick Sort

Quick sort also uses divide and conquer strategy, but it is an in-place algorithm and will not require additional space unlike merge sort. In this algorithm, we choose a pivot and partition the array based on the pivot. In this implementation, we have a deterministic approach and we always choose the last element as the pivot. After choosing the pivot, we place this pivot in its correct position by placing all the elements smaller than the pivot on the left, and all the elements greater than the pivot on the right. We do this by comparing all the elements with the pivot and the swapping the elements in their correct positions. We recursively call quicksort on the left and right half of the partition Index and the array gets sorted eventually.

<u>Input Generation and Testing:</u>

- For testing, an array was built on the heap and a pointer to that array was passed into each algorithm for efficiency since copying array from one function to another takes more space and time
- Testing was done by passing increments of 1000 elements into each sorting algorithm
- For bubble sort, increments were made till 100,000 while others were incremented till data sets were as large as 1,000,000
- Random array was created by generating random numbers which contained duplicates as well
- The algorithms were timed separately but with same inputs in three different categories: sorted, reversely sorted and randomly sorted inputs
- The environment was kept constant by running the same programs during testing, using the same timing strategy, and using the same compiler
- The experiment was repeated for each testing category at least 3 times and an average were taken to remove any discrepancies in data

EXPERIEMENTAL RESULTS

<u>Bubble Sort:</u>

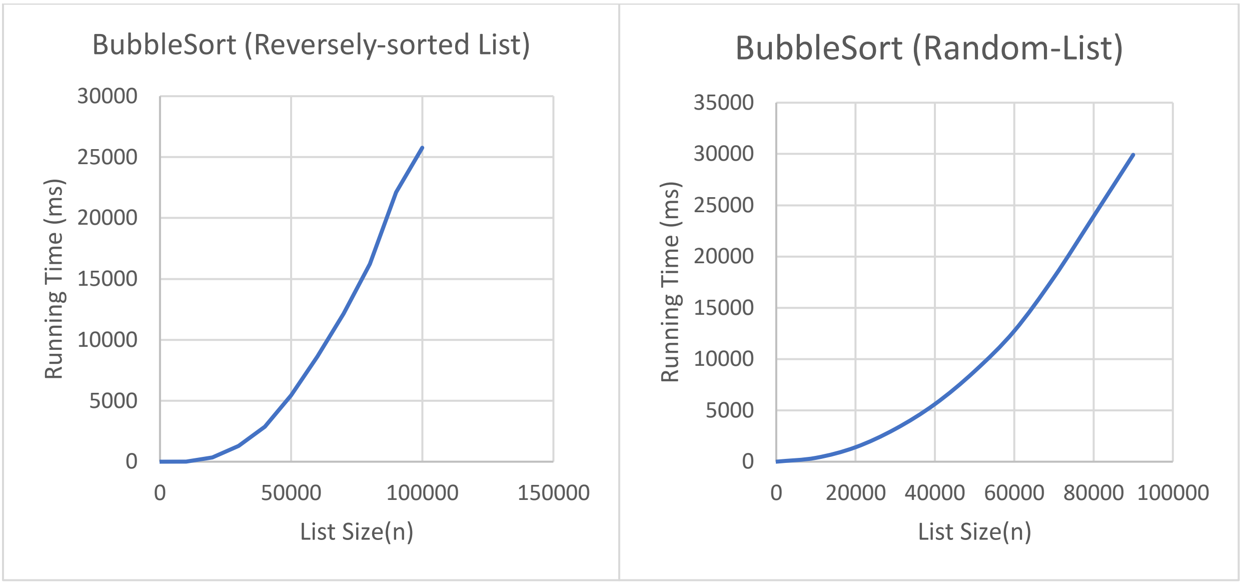


Fig: Worst Case Performance of Bubble Sort in Reversely Sorted and Random List

As expected, the worst-case time complexity of bubble sort was seen to be O(n^2) which took place in the case of reversely sorted and random list. Reversely sorted list took more time than the random list because the algorithm had to loop through the whole list in case of reversely sorted list, whereas in case of random list, the algorithm may not have looped through the entire list because the algorithm stops when the array is sorted.

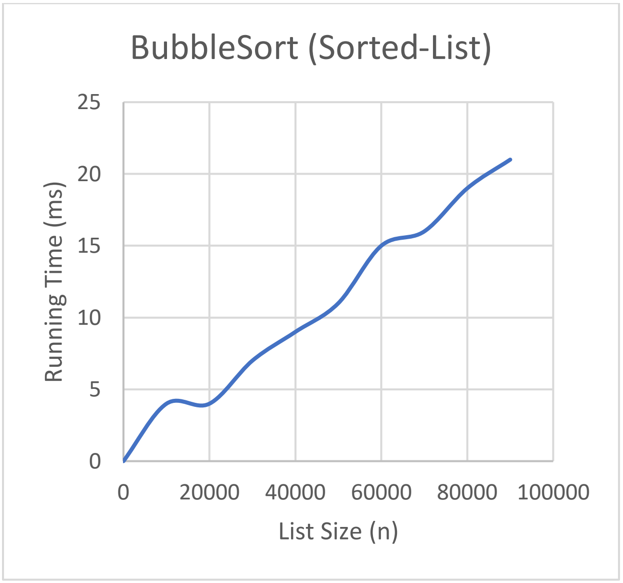
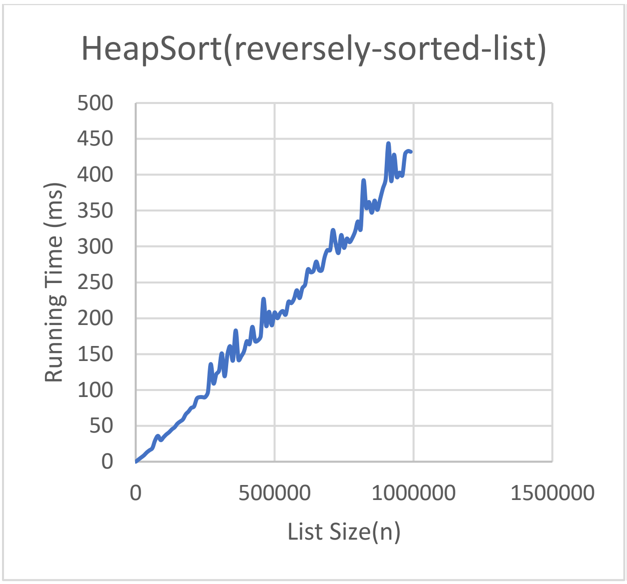
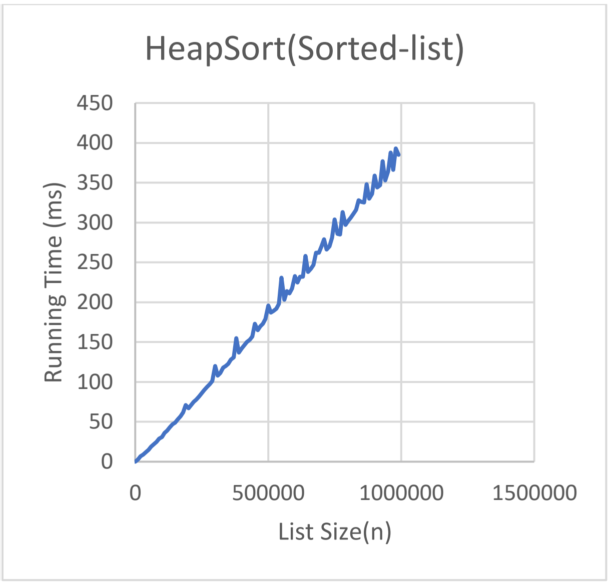


Fig: Best Case Performance of Bubble Sort in Sorted List

The best-case scenario was seen when bubble sort was used on a sorted list which was O(n). This is because the list was only looped once and since no swaps were made, the looping was stopped.

Heap Sort:





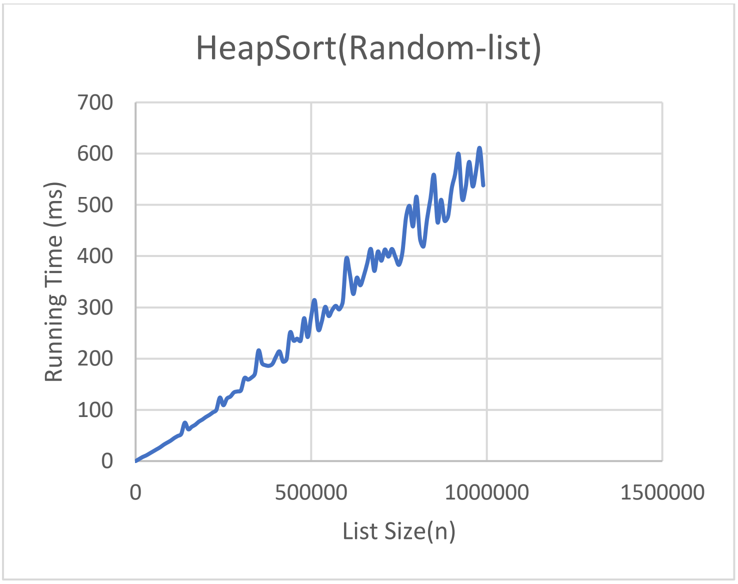
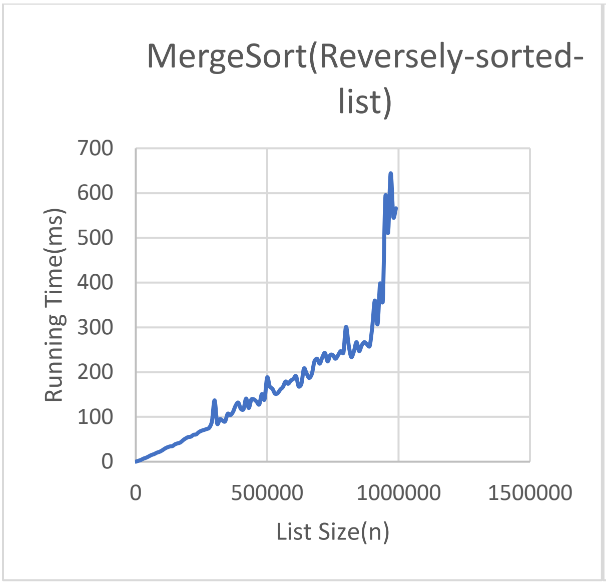
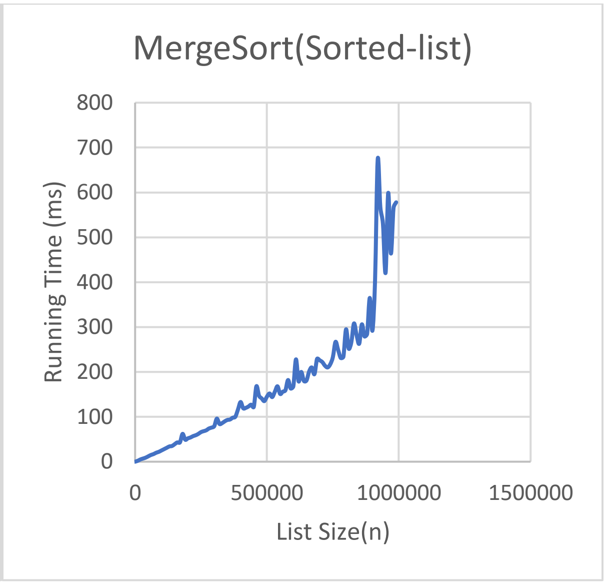


Fig: Best Case, average case and worst case time complexity for Heap Sort which is O(n log n)

For Heap sort, every data set was sorted in O(n log n) time complexity. This includes the bottom up heap construction and the removal process in which downheap operation takes place. For very large data, the heap sort starts getting slower as we see in the spikes at the end of the graph.

Merge Sort:





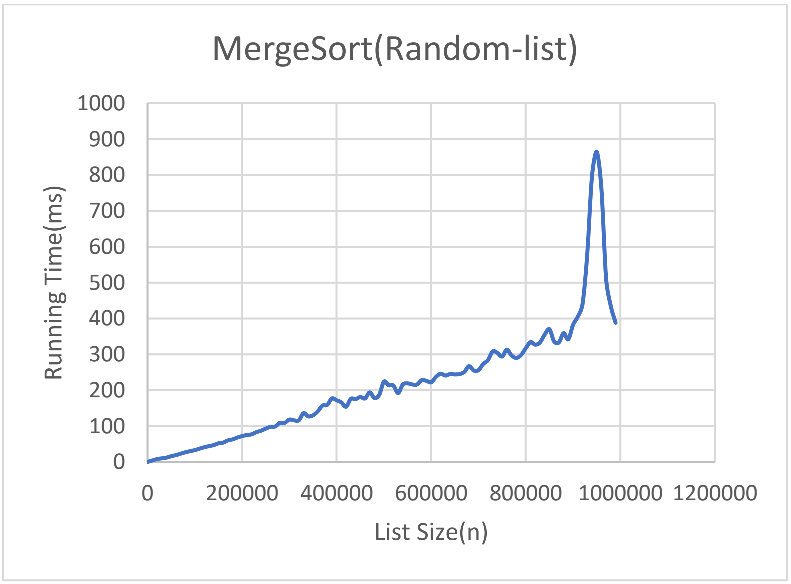
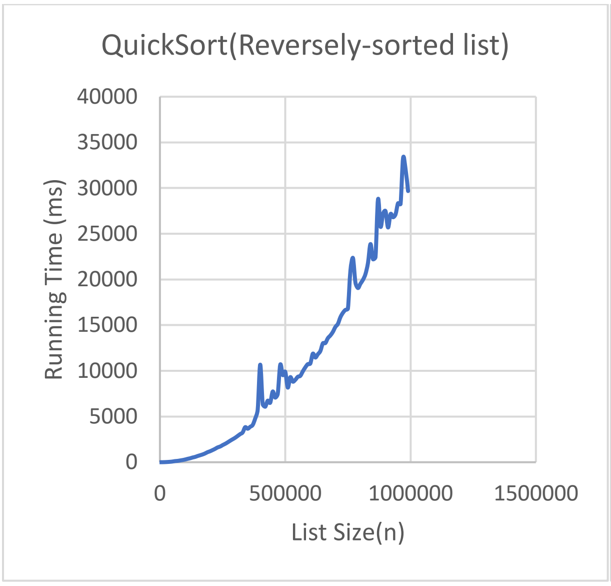


Fig: Best Case, average case and worst case time complexity for Merge Sort which is O(n log n)

Merge sort is has similar time complexity to heap sort which is O(n log n), however, we can observe that for very large input size, merge sort is slower than heap sort and for small to medium sized inputs, merge sort is faster.

Quick Sort:



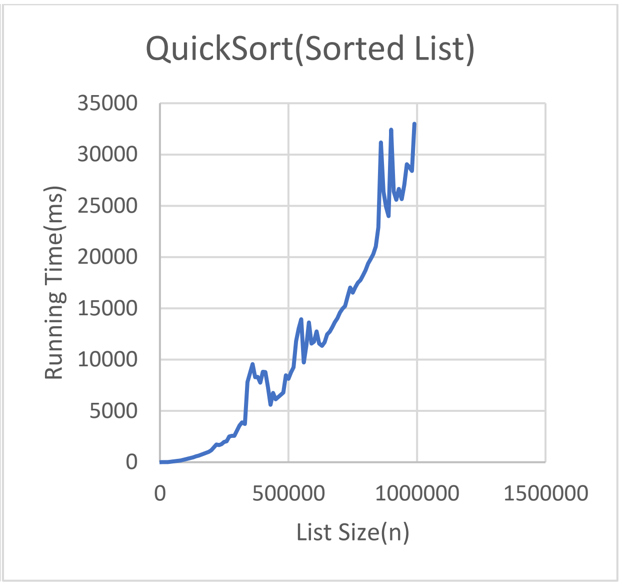


Fig: Worst Case Performance of QuickSort when the list is sorted and reversely sorted list

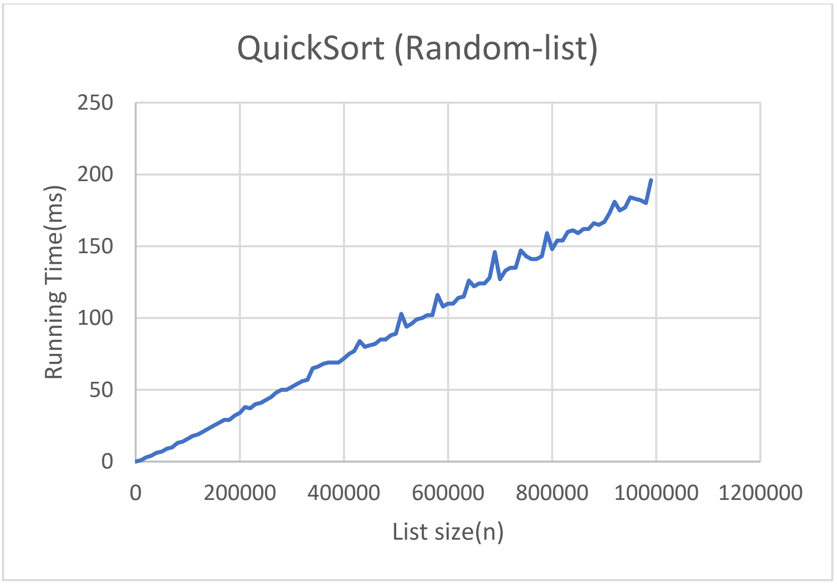


Fig: Best Case performance of Quick Sort for a random list

Quick sort had worst-case time complexity of O(n^2) when the list was sorted or reversely sorted. This is because the pivot was either the greatest or smallest element in the list and n-1 partition calls had to be made since all the elements were either smaller or greater than the pivot. However, the quick sort performed really well for the random list. With time complexity of O(n log n), it proved to be the fastest sorting algorithm. Quicksort was faster than merge sort and heap sort and had great time complexity even for large data sets. Unlike merge sort and heap sort, the graph did not show any spikes for very large data sets.

CONCLUSIONS

Based on our theoretical and experimental results, the following conclusions can be made:

- Bubble sort is not practical to use unless the data we have is small and nearly sorted in which case the performance can be O(n).
- Bubble sort is the least efficient sorting algorithm for random data sets since it has the time complexity of O(n^2).
- Merge sort is slightly faster than heap sort for all input types until the data sets are extremely large (almost 1 million), for which heap sort does fairly well.
- Quick sort has the worst time complexity for reversely sorted and sorted lists, almost as worse as bubble sort.
- Quick sort is the fastest sorting algorithm for random data sets and has the most consistent running time for large data sets.
- Experimental data satisfies our theoretical analysis for most cases.
- Type of input clearly affects the performance of quicksort and bubble sort but not for heap sort and merge sort
- Very large input size affects the performance of merge sort and heap sort but not for quicksort in terms of time complexity when data is random
- Discrepancies in some results such as merge sort vs heap sort for large data sets can happen because of space constraints since data sets were too large.

IMPROVEMENTS

Bubble Sort:

- Stop loop once array is sorted

This optimization will reduce the number of swaps needed and will improve the running time complexity. For sorted lists, we already saw how the running time complexity was improved to O(n) with this optimization.

Heap Sort:

- Two swap method

The two-swap method sorts two elements at a time for each heap construction and reduces the time complexity by 30-50% for construction of heap and in sorting of array elements.

Merge Sort:

- Use sentinel

  Having a sentinel at the end of the array will reduce the total number of comparisons while merging the arrays. This is because the sentinel can be used to check if we have reached the end of each of the two arrays we are merging. Without the sentinel, an additional check is required every time we are comparing.

- In-place merge-sort

  One of the arguments against merge sort is that inherently it is not an in-place sorting algorithm which means that it requires additional space. There are algorithms that can help achieve in-place merge sorting but it downgrades the performance to O(n^2) which will not be practical for large data sets. However, there is also an algorithm of merge sort called bottom-up merge sort which uses just one temporary array and achieves O(n log n) performance.

Quick Sort:

- Choose median of list as pivot

  The best case occurs when the pivot is the median of the list or close to the median so we don't run into having one large subarray. We can estimate the median of the list by calculating the median and then choosing the median as the pivot for each partition. This is useful when we know our dataset beforehand

- Choosing random pivot

  Random pivoting can improve the likelihood of getting the average case which is O(n log n) especially for sorted or nearly sorted lists and lists containing duplicate elements.

Faster sorting algorithms:

- Trim sort:

  Used as standard sorting algorithm for python, trim sort is a stable sorting algorithm which beats every sorting algorithm with worst case time complexity of O(n log n) and best case of O(n). Trim sort uses binary insertion sort and improved merge sort to sort data. Trim sort is highly effective and more learning needs to be done to fully explain how trim sort works.