SORTING ALGORITHMS: IMPLEMENTATION AND ANALYSIS

TASMAIYA TAMBOLI MICHAEL OLUWOLE FATEMA KHATUN

tasmaiyatamboli@my.unt.edu michaeloluwole@my.unt.edu fatemakhatun@my.unt.edu

S M MAZHARUL HOQUE CHOWDHURY GOPINATH REDDY GANGIREDDYGARI

smmazharulhoquechowdhury@my.unt.edu gopinathreddygangireddygari@my.unt.edu

1 INTRODUCTION

The objective of this project is to implement and analyze four sorting algorithms: Insertion Sort, Selection Sort, Heapsort, and Merge sort. The analysis will cover a range of input sizes, from 10 to 1,000,000, to measure the runtime performance of each algorithm. This will help in understanding the time complexity of the algorithms and how their performance scales with varying input sizes.

2 SORTING ALGORITHMS

2.1 INSERTION SORT

Gradually creating a sorted section of the array, the Insertion Sort algorithm operates. Each element is inserted into the array in its proper location in relation to the previously sorted portion as it iterates through it.

1. Treat the first element as a sorted chunk and begin with the second element (index 1)
2. Compare the elements in the sorted section (left side) with the current element (referred to as the key)
3. Elements larger than the key should be moved to the right.
4. Put the key in the proper location. For every element in the array, repeat.
5. Repeat for every element in the array.

2.1.1 IMPLEMENTATION

1. i = 1 to n-1: Beginning with the second entry, we loop through the array.
2. The current element to be added to the sorted section is key = A[i].
3. The last element in the sorted segment is indicated by the value j = i - 1.
4. The key is compared with each element in the sorted section while loop.
5. A [j + 1] = A [j]: To create room for the key, move larger items to the right.
6. A [j + 1] = key; Position the key appropriately.

2.2 SELECTION SORT

Selection Sort is a straightforward comparison-based sorting algorithm that alternates the first element of the array's unsorted section with the smallest element periodically chosen from the array's unsorted segment.

2.2.1 ALGORITHM'S FUNCTION

1. Separate the array into two sections: the initially empty sorted section. The part that isn't sorted (originally the whole array).
2. Determine the lowest element in the section that isn't sorted.
3. Replace the minimum element with the unsorted portion's initial element.
4. Reduce the amount of unsorted material and increase the amount of sorted material.
5. Continue doing this until the array is fully sorted.

2.2.2 IMPLEMENTATION

1. Even if the array is already sorted, Selection Sort always executes in $O(n^2)$ time.
2. It performs comparisons between (n-1) + (n-2) +... + 1 = $O(n^2)$.
3. However, it performs only O(n) swaps, which makes it superior to Bubble Sort in terms of swaps.
4. (for (int i = 0; i < n - 1; i++)) Outer Loop. Every element saves the final one is iterated over. assumes that the lowest element in the unsorted section is A[i].
5. (for (int j = i + 1; j < n; j++)) Inner Loop, looks for the smallest element in the unsorted section.
6. Update min Index if a smaller element is discovered.
7. Replace it with A[i] once the minimum element in the remaining section has been determined.

2.3 MERGESORT

Recursively dividing the array into smaller subarrays, sorting them, and then recombining them in a sorted order is the process of sorting, a divide-and-conquer algorithm.

2.3.1 ALGORITHM'S FUNCTION

1. Divide the array in half so that there is only one element in each subarray.
2. Conquer: Sort each half recursively.
3. Merge: Create a single sorted array by combining the two sorted parts.

2.3.2 IMPLEMENTATION

1. Files : mergesort.cpp : This file contains the main function ,input file reading code and mergesort.hpp : This header file contains merge sort function.
2. mergesort.cpp reads a list of integers from a input file, sorts them using the , it displays the sorted output.
3. Otherwise, it simply verifies if sorting was do/output file handling and iostream for operations. A vector A is declared to station used to calculate size of vector.
4. Call merge sort function Merge Sort(A,0,n-1). which defined in mergesort.hpp based on Divide and Conquer algorithum. The program reads inputs from input file which categorized based on number of digits.
5. Few Constraints we have to consider before this The file should contains only integer and only one file name as command-line arg also program uses vector so for larger file it will consume more memory.
6. Merge Sort uses O(nlogn) also here in program it saves time for large inputs by size.

2.4 HEAP SORT

Insertion, deletion, and heap sorting are among the heap operations supported by the MinHeap class defined by this code. The parent node of the MinHeap is always smaller than its child nodes, in accordance with the binary heap property.

2.4.1 ALGORITHM'S FUNCTION

1. Swap (int *x, int *y) is a Heap Utility Function.
2. It uses pointers to swap two integers. Initialization and Heap Structure. MinHeap (int size).
3. Constructor: Sets up a blank heap of a specified size. MinHeap()
4. The dynamically allocated heap array is deleted by the destructor.
5. Heap Structure Helper Functions. The index of the left child is returned by int left (int parent). The index of the correct kid is returned by int right (int parent). The parent node's index is returned by int parent (int child).

2.4.2 IMPLEMENTATION

1. Add an element by using the syntax insert(int element) adds a new element to the heap's end. maintains the min-heap attribute by moving it upward.
2. Use extractMin() to extract the minimum, returns the root, the smallest element, after removing it uses Min-Heapify() to recover the heap property and replaces it with the final piece.
3. Heap Property Maintenance (MinHeapify(int i)) compares a node with its offspring to guarantee the min-heap property. The smallest value is moved up the tree recursively.
4. All elements are added to the heap using the heap sorting algorithm (heapsort()).
5. One by one, the minimum elements are extracted and stored in a sorted sequence.

3 GRAPHICAL REPRESENTATION AND ANALYSIS

Based on below mentioned experimental result if plot graph Execution Time(s) VS. Input size , we will get following analysis of time complexity for each algorithms. The given tables is presenting the execution time based on different input size (10, 100, 1000, 10000, 100000 and 1000000) for Insertion sort, Selection sort, Merge sort and Heapsort. This given dataset can be divided into three sets; for n = 10, 100 and 1000 can be considered as small input size, for n = 10000 and 100000 can be considered as medium input size and for n = 1000000 can be considered as large input size.

For the small input size all the algorithms were executed very fast and the execution time ranges between 0.002s to 0.006s. As the input size was considerably small, therefore the algorithms were able to execute them with no stress. As a result, the execution time difference are not significantly noticeable for almost all of them. However, Heapsort performed slightly better than other algorithms and produced either equal or faster execution or run time. Merge sort's performance was the second best among them and Insertion sort and Selection sort performed almost the same based on the execution time.

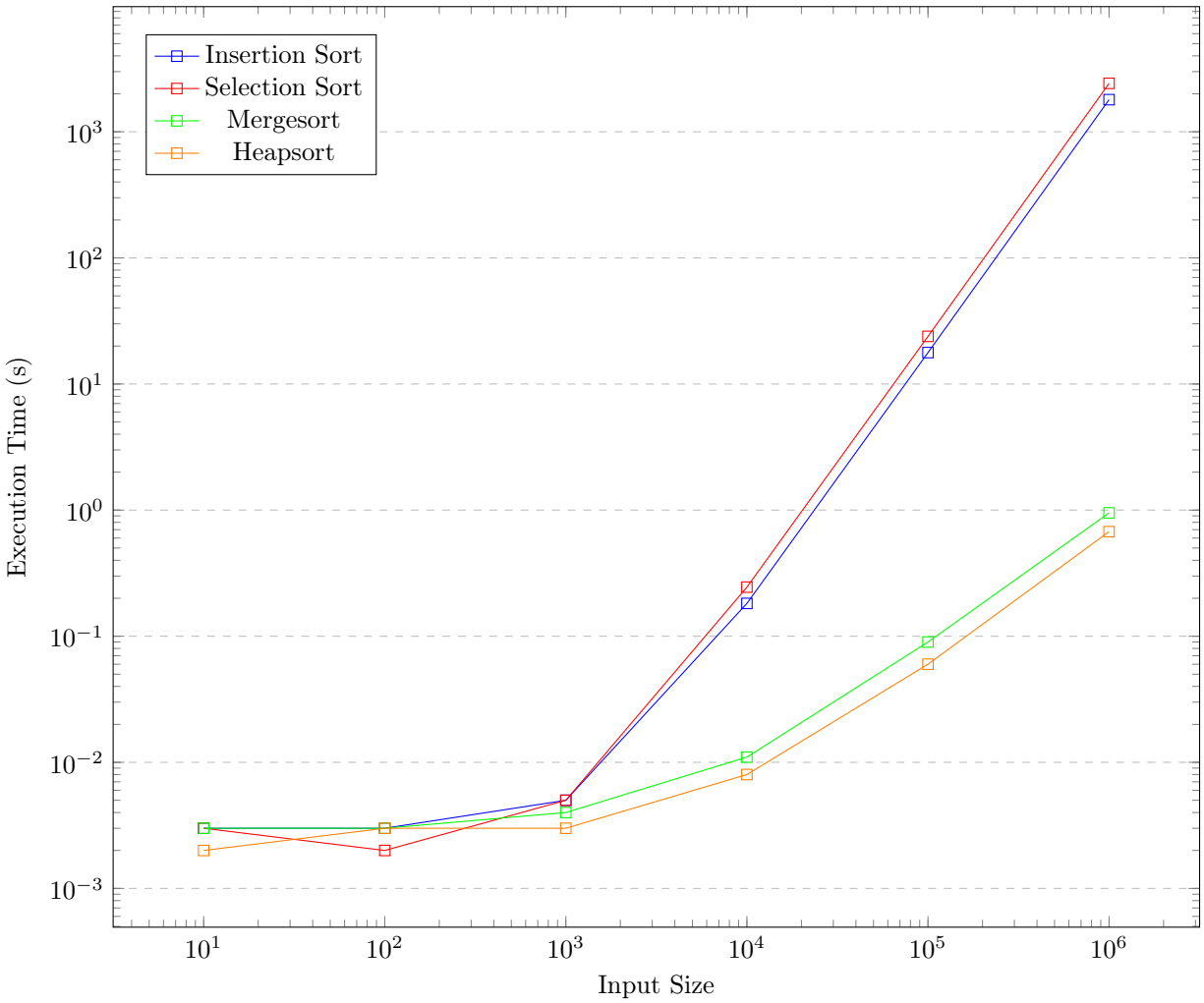
For the medium input size run time difference started to become significant, as for n = 10000 the Insertion sort took o.182s, selection sort took 0.245s, merge sort took 0.011s and heapsort took 0.008s on average. Therefore, it is clearly visible that merge sort and heapsort performed significantly better than the insertion sort and the selection sort and their run time difference was not significant but visible. It presents the advantage of O(nlogn) over O(n2) based on algorithms worst case analysis.

On the other hand, for input size n = 100000, the performance difference is even more significant. It took 17.748s for the insertion sort, 23.858s for the selection sort, 0.090s for the merge sort and 0.060s for the heap sort. This data indicates the inefficiency of the quadratic sorting algorithms for very large input size. Based on their performance we can rank them as heapsort, merge sort, insertion sort and selection sort from best to worst algorithm for medium input size.

Lastly, for input size n = 1000000, time difference of insertion sort and selection sort is futile against merge sort and heapsort. Insertion sort took almost 30 minutes on average and selection sort took around 40 minutes on average to execute the sorting. Meanwhile, merge sort took 0.951s and heapsort took 0.675s to execute the sorting. As a result it is clear that, merge sort and heapsort is superior to perform sorting on large input size.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion Sort	$O(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Selection Sort	$O(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Mergesort	$O(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heapsort	$O(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

Table 1: Time Complexity of Sorting Algorithms



4 EXPERIMENTAL RESULTS

The following tables document the execution time of the sorting algorithms for different input sizes. Each row represents the average execution time over multiple runs.

Table 2: Running time of different sorting algorithms with input size = 10

Sorting Algorithm	10.1	10.2	10.3	10.4	10.5	Average
Insertion Sort	0.003s	0.003s	0.003s	0.003s	0.003s	0.003s
Selection Sort	0.003s	0.003s	0.002s	0.003s	0.002s	0.003s
Mergesort	0.003s	0.002s	0.002s	0.003s	0.003s	0.003s
Heapsort	0.002s	0.002s	0.002s	0.002s	0.003s	0.002s

Table 3: Running time of different sorting algorithms with input size = 100

Sorting Algorithm	100.1	100.2	100.3	100.4	100.5	Average
Insertion Sort	0.003s	0.003s	0.003s	0.003s	0.003s	0.003s
Selection Sort	0.003s	0.002s	0.002s	0.003s	0.002s	0.002s
Mergesort	0.002s	0.003s	0.003s	0.003s	0.003s	0.003s
Heapsort	0.003s	0.002s	0.002s	0.003s	0.003s	0.003s

Table 4: Running time of different sorting algorithms with input size = 1000

Sorting Algorithm	1000.1	1000.2	1000.3	1000.4	1000.5	Average
Insertion Sort	0.005s	0.004s	0.005s	0.005s	0.004s	0.005s
Selection Sort	0.005s	0.005s	0.005s	0.006s	0.005s	0.005s
Mergesort	0.004s	0.004s	0.004s	0.004s	0.004s	0.004s
Heapsort	0.003s	0.003s	0.003s	0.003s	0.003s	0.003s

Table 5: Running time of different sorting algorithms with input size = 10000

Sorting Algorithm	10000.1	10000.2	10000.3	10000.4	10000.5	Average
Insertion Sort	0.179s	0.182s	0.188s	0.180s	0.179s	0.182s
Selection Sort	0.255s	0.247s	0.241s	0.239s	0.241s	0.245s
Mergesort	0.011s	0.011s	0.011s	0.011s	0.011s	0.011s
Heapsort	0.008s	0.008s	0.008s	0.008s	0.008s	0.008s

Table 6: Running time of different sorting algorithms with input size = 100000

Sorting Algorithm	100000.1	100000.2	100000.3	100000.4	100000.5	Average
Insertion Sort	17.680s	17.730s	17.673s	17.758s	17.897s	17.748s
Selection Sort	23.936s	23.904s	23.908s	23.840s	23.702s	23.858s
Mergesort	0.091s	0.089s	0.091s	0.089s	0.088s	0.090s
Heapsort	0.059s	0.061s	0.059s	0.061s	0.060s	0.060s

Table 7: Running time of different sorting algorithms with input size = 1000000

Sorting Algorithm	1000000.1	1000000.2	1000000.3	1000000.4	1000000.5	Average
Insertion Sort	30m11.762s	29m47.691s	29m43.851s	30m15.622s	29m51.773s	29m58.139s
Selection Sort	40m5.956s	40m6.316s	40m9.399s	40m53.501s	40m29.355s	40m20.905s
Mergesort	0.951s	0.966s	0.938s	0.941s	0.961s	0.951s
Heapsort	0.674s	0.668s	0.688s	0.672s	0.673s	0.675s

5 CONCLUSION

Finally, based on the overall performance it can be said that those algorithms can be ranked as heapsort, merge sort, insertion sort and selection sort from best to worst algorithm for overall sorting based on their run time. Sometimes it also depends on the bussiness requiremnet which algorithum to choose based on best fit to case study, memory constraints, the size and structure of the data, stability requirements and the type of data being sorted .