KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Report

Department of Computer Science and Engineering

Course Title: Data Structures and Algorithms Laboratory

Course No: CSE 2106

Topic: Sorting Algorithms Implementation and Time Complexity Comparison

Name	:	Kazi Rifat Al Muin
Roll	:	2107042
Group	:	A2
Year	:	2nd
Semester	:	1st
Date of Submission	:	20/04/2024

Objectives

- To know about different 8 different sorting algorithms: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort, and Counting Sort.
- To Learn how to implement these 8 sorting algorithms and how they work.
- To compare their theoretical time complexities and empirical time complexities obtained from the runtime measurements.
- To analyze if any unexpected result is found and discuss its reason
- To know the best use case of every algorithm and efficiently choose the algorithm.

Introduction

Sorting algorithms are fundamental in computer science, providing methods to arrange elements in a specific order efficiently. Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order until the entire list is sorted. Insertion Sort builds the final sorted array one element at a time by inserting each new element into its correct position among the already sorted elements. Selection Sort divides the input list into two parts: the subarray of sorted elements and the subarray of unsorted elements, selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. Quick Sort employs a divide-and-conquer strategy, selecting a 'pivot' element and partitioning the array into two sub-arrays around the pivot, recursively sorting the sub-arrays. Merge Sort divides the array into two halves, sorts each half independently, and then merges the sorted halves to produce a single sorted array. Heap Sort uses a binary heap data structure to represent the input array as a complete binary tree, repeatedly removing the root element (the maximum in a max-heap or the minimum in a min-heap) and maintaining the heap property. Radix Sort sorts the elements by first grouping the individual digits representing the same significant position and then sorting the elements according to their significant positions. Count Sort counts the frequency of each distinct element in the input array and uses this information to place the elements in sorted order without comparison.

Implementation

Attached is a C++ file containing implementations of eight sorting algorithms: Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Merge Sort, Heap Sort, Radix Sort, and Count Sort. Each algorithm is thoroughly commented to explain its operation. The code utilizes a random number generation function to create arrays of varying sizes, and the C++ chrono library measures the execution time of each sorting algorithm for each input size. This approach allows for a standardized evaluation of the efficiency and effectiveness of these sorting techniques.

Time Complexity Analysis

Theoretically

Here are the theoretical best-case, average-case, and worst-case time complexities for the implemented sorting algorithms :

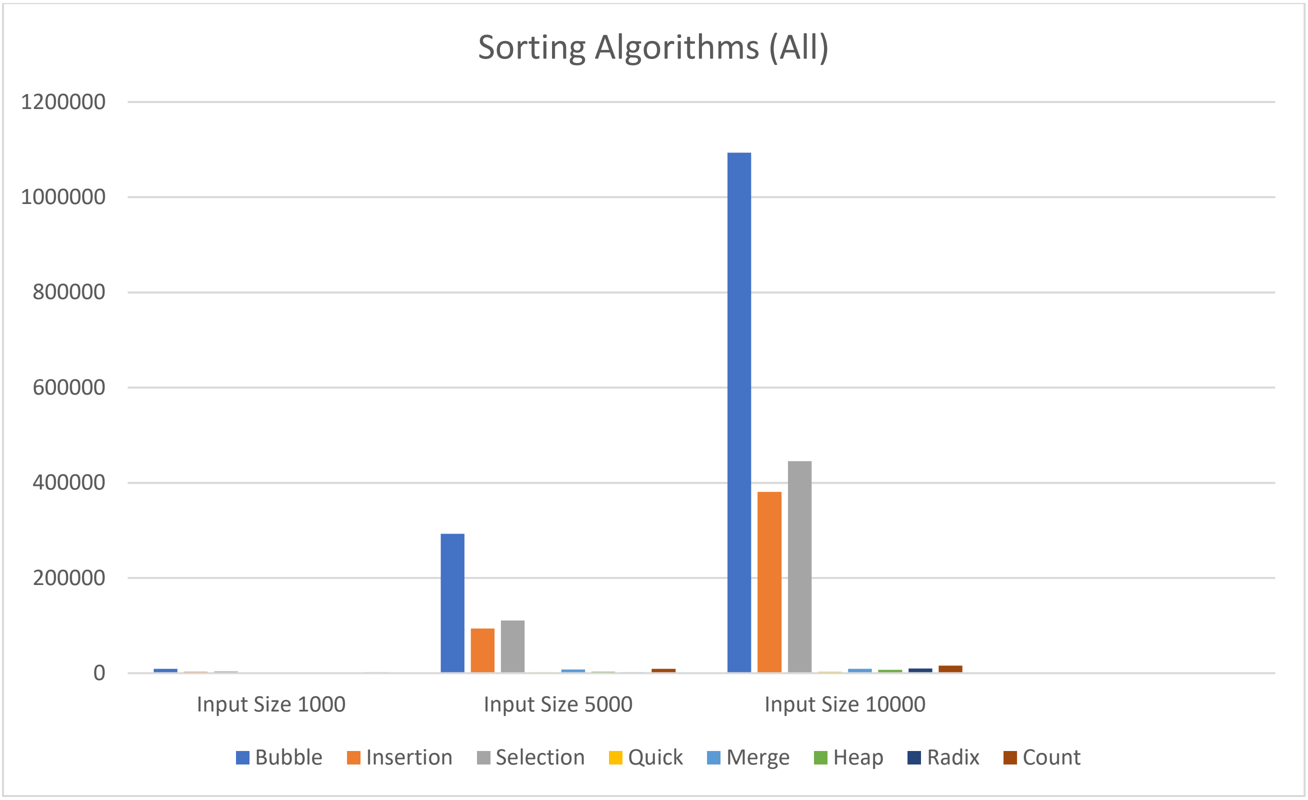
Algorithm	Worst Case	Average Case	Best Case
Bubble Sort	O(n^2)	O(n^2)	O(n)
Insertion Sort	O(n^2)	O(n^2)	O(n)
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Quick Sort	O(n^2)	O(nlog(n))	O(nlog(n))
Merge Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))
Heap Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))
Radix Sort	O(nd)	O(nd)	O(nd)
Count Sort	O(n+k)	O(n+k)	O(n+k)

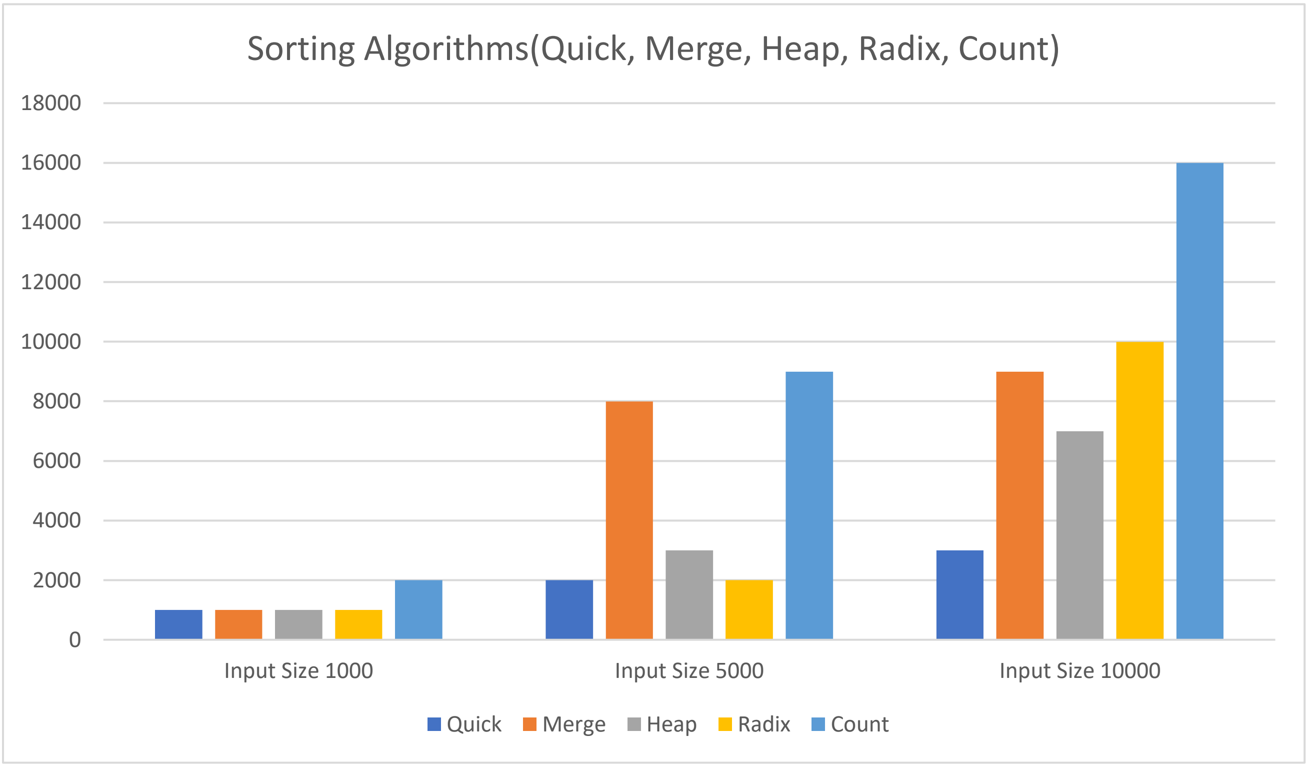
Empirical Analysis :

Here is the dataset from the implemented program for different input array sizes:

Input Size & Elements	Algorithm	Execution Time (µS)
Input size: 1000 Elements up to 100000	Bubble Sort	8994
	Insertion Sort	2997
	Selection Sort	3996
	Quick Sort	998
	Merge Sort	1000
	Heap Sort	998
	Radix Sort	999
	Count Sort	1996
Input size: 5000 Elements up to 500000	Bubble Sort	292817
	Insertion Sort	93940
	Selection Sort	110930
	Quick Sort	1997
	Merge Sort	7994
	Heap Sort	2997
	Radix Sort	1999
	Count Sort	8994
Input size: 10000 Elements up to 1000000	Bubble Sort	1093320
	Insertion Sort	380764
	Selection Sort	445723
	Quick Sort	2998
	Merge Sort	8994
	Heap Sort	6996
	Radix Sort	9994
	Count Sort	16000

Graphs


Sorting Algorithms (All)
1200000
1000000
800000
600000
400000
200000
0
Input Size 1000
Input Size 5000
Input Size 10000
Bubble
Insertion
Selection
Quick
Merge
Heap
Radix
Count


Sorting Algorithms(Quick, Merge, Heap, Radix, Count)
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Input Size 1000
Input Size 5000
Input Size 10000
Quick
Merge
Heap
Radix
Count

Discussion and Analysis

From the analysis of actual execution time, we can see that the bubble sort gives the worst performance. Insertion sort gives better performance than the other two sorting algorithms with O(n^2) time complexity.

The remaining 5 algorithms give far better performance. From all, the quick sort amazingly gives excellent performance. Despite having the worst-case complexity of O(n^2), most of the time quick sort gives the surprisingly best result.

Merge sort gives a consistent performance in large input sizes. Heap sort gives a slightly better result than merge sort. Radix sort gives very good performance in smaller input sizes and increases in larger input sizes. The count sort gives worse performance than these other 4 and takes more time with a bigger input size.

The unexpected result was from quick sort, which worked so much faster than expected.

Conclusion

According to input size and data size, the performance of every algorithm differs. All algorithms are important in different cases. We need to understand when and why the algorithms perform better or worse. According to that, We can pull off the best result from the sorting algorithms.