EMPIRICAL ANALYSIS OF SORTING ALGORITHMS

Name: Aya Samara

ID: 202011867

A. Implementation:

This report presents an empirical analysis of three sorting algorithms: Bubble Sort, Merge Sort, and Quick Sort. The goal is to compare their practical performance in terms of execution time using C++.

I used C++ programming language applied on C-free 5.0 version

My device: windows 11 pro, core i7, 10^{th} gen, 8GB RAM

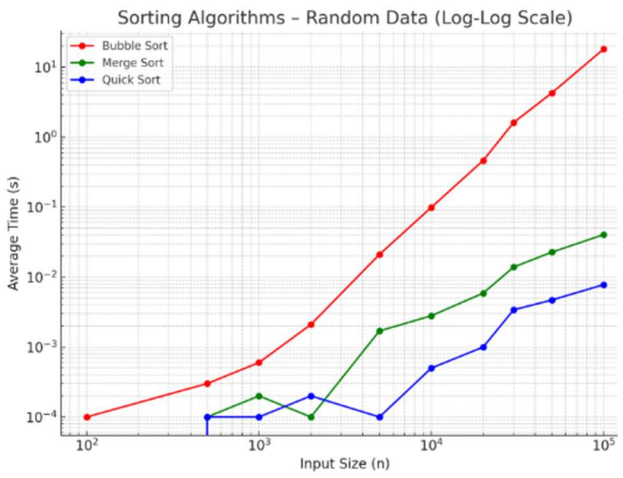
Data sizes: {100, 500, 1000, 2000, 5000, 10000, 20000, 30000, 50000, 100000} Each algorithm was executed 10 times, and the average was taken.

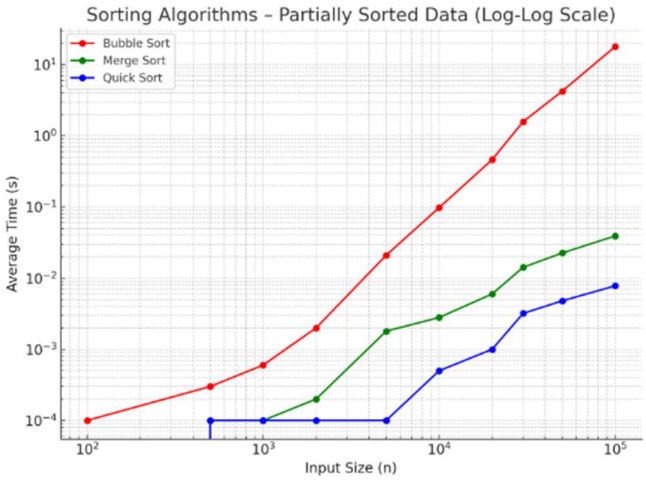
Data types: random, partially sorted, reverse sorted
*the code is at the end of the file in GitHub link

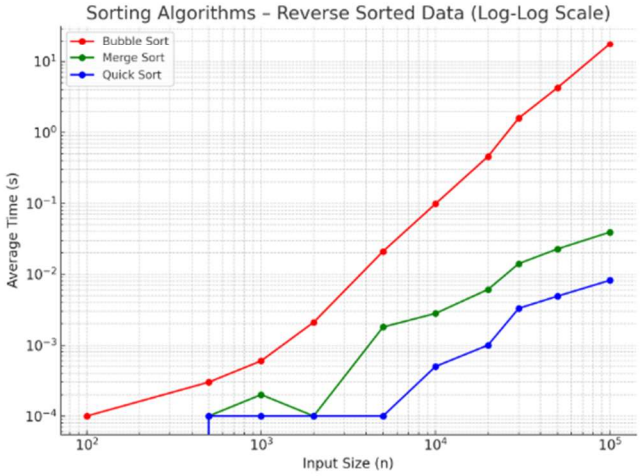
B. Experimental work:

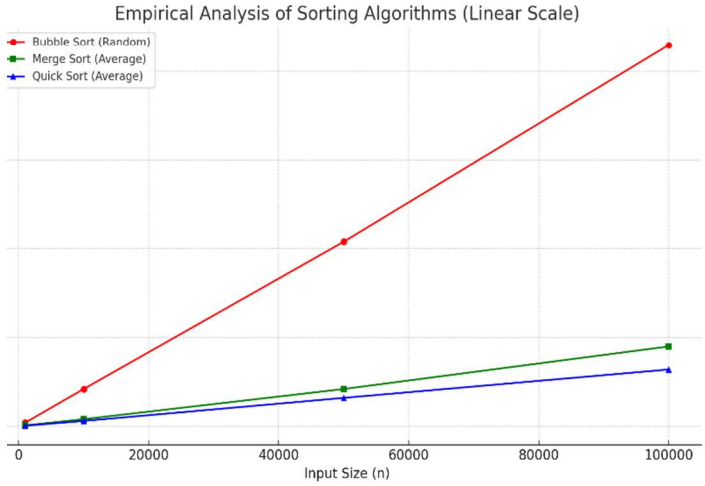
Results will be compared based on: Efficiency, Time Complexity, and Stability

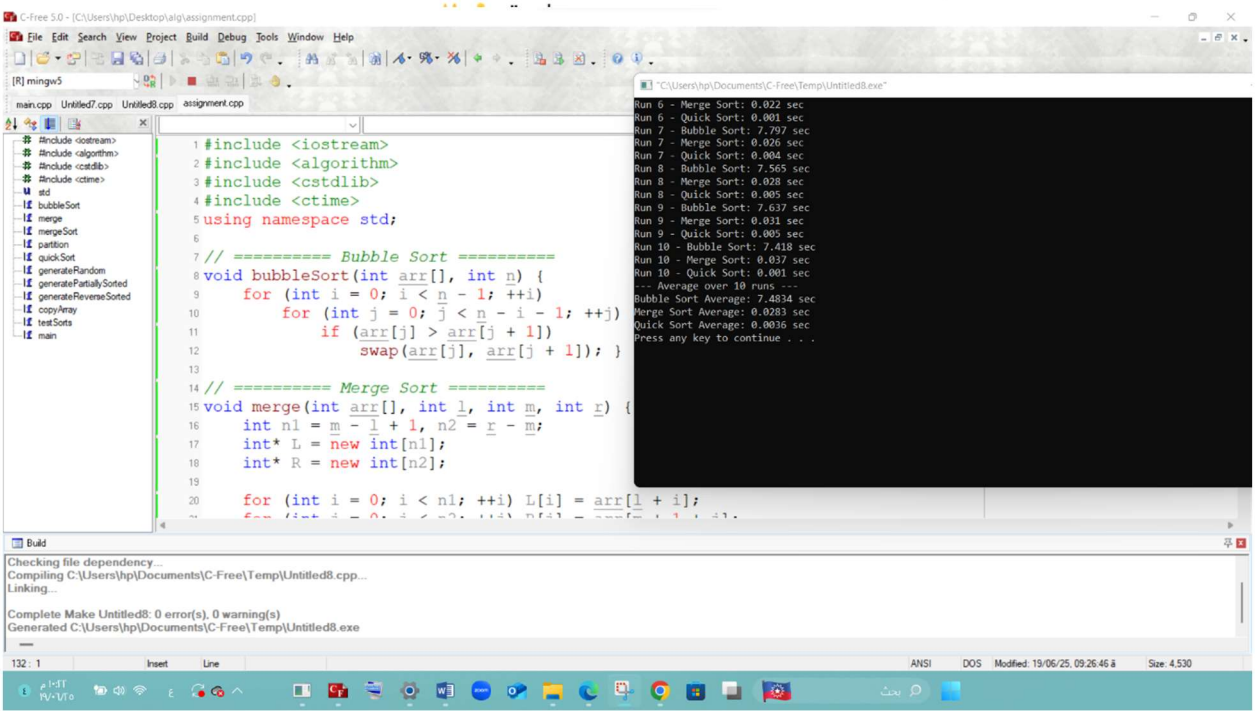
Size	Data Type	Bubble Sort	Merge Sort	Quick Sort
1000	Random	0.0007	0.0004	0.0001
1000	Sorted	0.0005	0.0003	0.0000
1000	Reverse Sorted	0.0006	0.0001	0.0000
10000	Random	0.0992	0.0026	0.0003
10000	Sorted	0.0640	0.0025	0.0006
10000	Reverse Sorted	0.0570	0.0022	0.0005
50000	Random	3.9726	0.0181	0.0035
50000	Sorted	2.6558	0.0166	0.0028
50000	Reverse Sorted	1.9404	0.0188	0.0026
100000	Random	17.7640	0.0578	0.0098
100000	Sorted	11.5838	0.0318	0.0042
100000	Reverse Sorted	7.4834	0.0283	0.0036


Sorting Algorithms – Random Data (Log-Log Scale)
Bubble Sort
Merge Sort
Quick Sort
Average Time (s)
Input Size (n)


Sorting Algorithms – Partially Sorted Data (Log-Log Scale)
Bubble Sort
Merge Sort
Quick Sort
Average Time (s)
Input Size (n)


Sorting Algorithms – Reverse Sorted Data (Log-Log Scale)
Bubble Sort
Merge Sort
Quick Sort
Average Time (s)
Input Size (n)


Empirical Analysis of Sorting Algorithms (Linear Scale)
Bubble Sort (Random)
Merge Sort (Average)
Quick Sort (Average)
Input Size (n)


C-Free 5.0 - [C:\Users\hp\Desktop\alg\assignment.cpp]
File Edit Search View Project Build Debug Tools Window Help
main.cpp Untitled7.cpp Untitled8.cpp assignment.cpp
#include <iostream>
#include <algorithm>
#include <cstdlib>
#include <ctime>
using namespace std;
// ========== Bubble Sort ==========
void bubbleSort(int arr[], int n) {
for (int i = 0; i < n - 1; ++i)
for (int j = 0; j < n - i - 1; ++j)
if (arr[j] > arr[j + 1])
swap(arr[j], arr[j + 1]); }
// ========== Merge Sort ==========
void merge(int arr[], int l, int m, int r) {
int n1 = m - l + 1, n2 = r - m;
int* L = new int[n1];
int* R = new int[n2];
for (int i = 0; i < n1; ++i) L[i] = arr[l + i];
"C:\Users\hp\Documents\C-Free\Temp\Untitled8.exe"
Run 6 - Merge Sort: 0.022 sec
Run 6 - Quick Sort: 0.001 sec
Run 7 - Bubble Sort: 7.797 sec
Run 7 - Merge Sort: 0.026 sec
Run 7 - Quick Sort: 0.004 sec
Run 8 - Bubble Sort: 7.565 sec
Run 8 - Merge Sort: 0.028 sec
Run 8 - Quick Sort: 0.005 sec
Run 9 - Bubble Sort: 7.637 sec
Run 9 - Merge Sort: 0.031 sec
Run 9 - Quick Sort: 0.005 sec
Run 10 - Bubble Sort: 7.418 sec
Run 10 - Merge Sort: 0.037 sec
Run 10 - Quick Sort: 0.001 sec
--- Average over 10 runs ---
Bubble Sort Average: 7.4834 sec
Merge Sort Average: 0.0283 sec
Quick Sort Average: 0.0036 sec
Press any key to continue . . .
Build
Checking file dependency...
Compiling C:\Users\hp\Documents\C-Free\Temp\Untitled8.cpp...
Linking...
Complete Make Untitled8: 0 error(s), 0 warning(s)
Generated C:\Users\hp\Documents\C-Free\Temp\Untitled8.exe

Bubble Sort's poor performance with large inputs aligns with its $O(n^2)$ time complexity.

Merge Sort and Quick Sort both demonstrate log-linear growth, consistent with O(n log n) time.

Quick Sort's worst-case time complexity of $O(n^2)$ didn't appear in our tests, likely because the pivot selection strategy avoided worst-case scenarios.

*Comparison:

Criterion	Bubble Sort	Merge Sort	Quick Sort
Theoretical Best Case	O(n) (already sorted)	O(n log n)	O(n log n)
Theoretical Average Case	$O(n^2)$	O(n log n)	O(n log n)
Theoretical Worst Case	$O(n^2)$	O(n log n)	$O(n^2)$ (bad pivot)
Practical Observation	Very slow as n increases	Fast and consistent across inputs	Fastest overall; slightly varies with input

*Observations:

The practical results **match** the theoretical complexity **quite closely.**

Merge Sort use more memory due to recursion and array copying, but this did not significantly affect speed.

*Efficiency:

Bubble Sort is the slowest across all scenarios. Especially for large inputs (least efficient and unsuitable for large datasets)

Merge Sort consistently performed well with stable and relatively fast execution times, regardless of input ordering.

Quick Sort outperforms others in most average/random cases, particularly on random and sorted inputs. It. its performance can be slightly affected by input order and pivot choice, but this was not a major issue in our tests. (The most efficient practically)

*Stability:

Merge and Bubble are **stable**, which is important when sorting records with duplicate keys (Bubble Sort only swaps adjacent elements when necessary, preserving the relative order of equal elements. Merge Sort maintains the order of equal elements during the merging process)

Quick Sort is **not stable (by default)** because it may change the relative order of equal elements during the partitioning phase.

*Conclusion:

Quick Sort is generally the fastest but not stable. Recommended for speed and performance

Merge Sort is highly consistent and stable. Recommended for stability and predictability

Bubble Sort is significantly slower and inefficient on large datasets. It is best used for educational purposes only.

*Github link:
https://github.com/Aya-s12/Aya-Samara-assignment