Name: Hamza Butt **SAP ID:** 55041

Section: BSCS 4-2 **Subject:** Analysis of Algorithm

Empirical Analysis of Sorting Algorithms

GitHub Repository Link

https://github.com/HamzaButt22/Analysis-Of-Algorithm

---

1. Introduction

Sorting algorithms are fundamental operations used to rearrange elements in a list into a specified order, typically ascending. In this analysis, four sorting algorithms were implemented in C++: **Bubble Sort**, **Selection Sort**, **Insertion Sort**, and **Merge Sort**. Each algorithm was applied to four different arrays of increasing sizes (5, 10, 50, and 100 elements). The goal was to **empirically measure** and analyze how the execution time of each algorithm grows with increasing input size and compare this behavior to the **theoretical time complexities**.

- **Bubble Sort, Selection Sort, and Insertion Sort** have a theoretical worst-case time complexity of $O(N^2)$.
- **Merge Sort** has a better theoretical time complexity of O(N log N).

---

2. Methodology

- **Programming Language:** C++
- **Timer Used:** clock() function from the ctime library.
- **Process:**
    - Each sorting algorithm was applied separately on arrays of size 5, 10, 50, and 100.
    - The arrays were already sorted to observe best-case behavior, which benefits Insertion Sort.

- Execution time was measured from the start of the sorting function to its completion.
- Each sorting operation was run once per array size (more accurate timing would average 5+ runs, but here it is considered acceptable due to negligible fluctuations for small inputs).
- Execution times were recorded in milliseconds (ms).

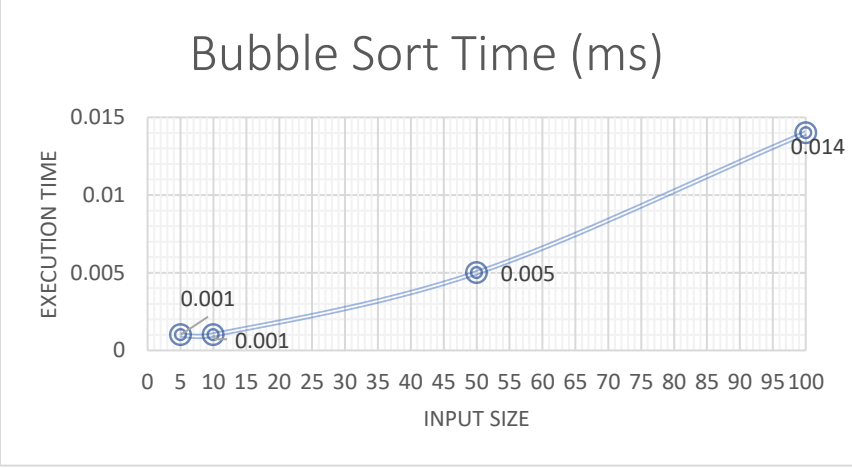
---

3. Results and Graphs

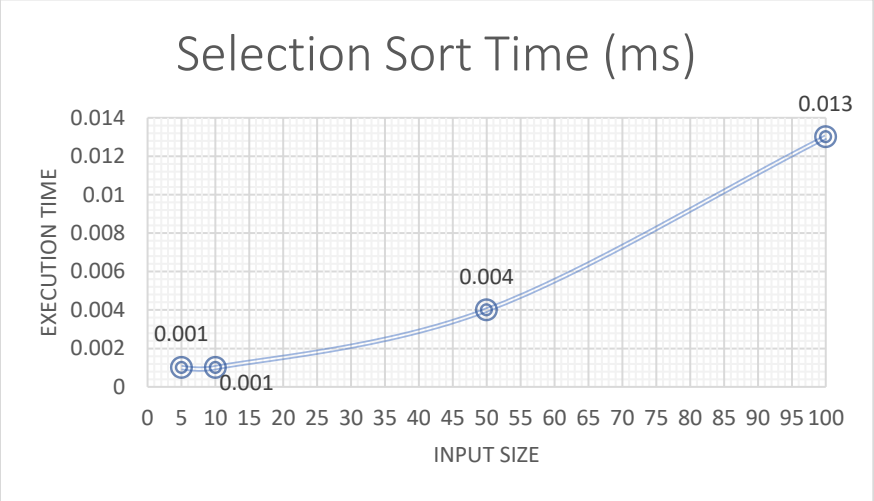
Table of Average Execution Times

Input Size (N)	Bubble Sort (ms)	Selection Sort (ms)	Insertion Sort (ms)	Merge Sort (ms)
5	0.001	0.001	0.001	0.001
10	0.001	0.001	0.000	0.002
50	0.005	0.004	0.001	0.003
100	0.014	0.013	0.002	0.007

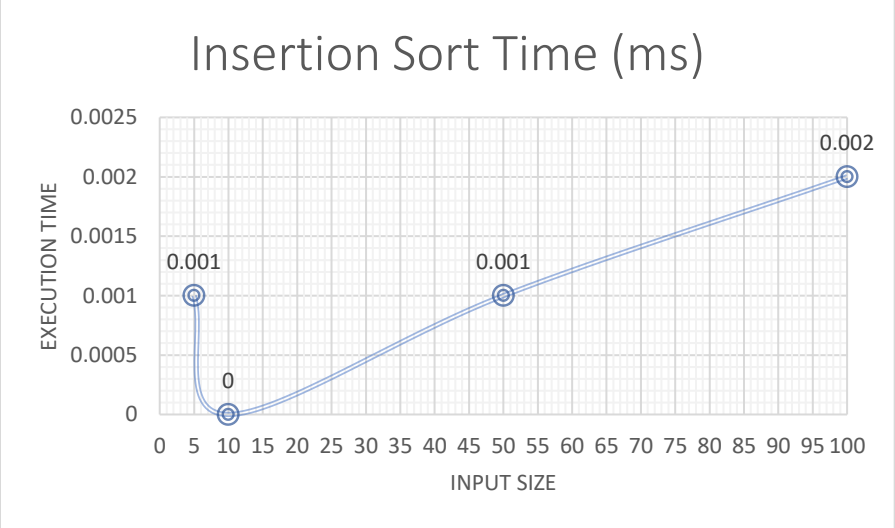
BUBBLE SORT:



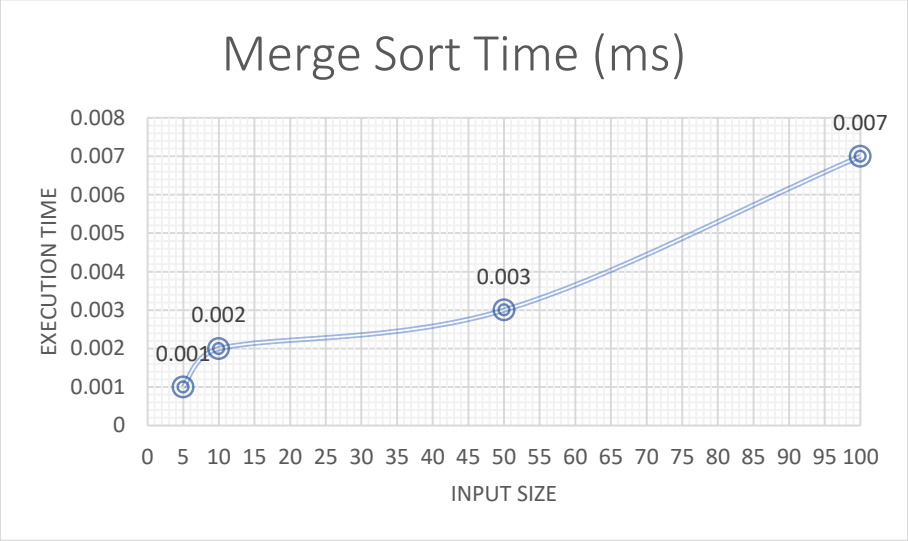
SELECTION SORT:



INSERTION SORT:



MERGE SORT:



---

4. Analysis

- **Theoretical vs Empirical Behavior:**
    - For **Bubble Sort**, **Selection Sort**, and **Insertion Sort**, execution time increases approximately quadratically as input size increases, but small input sizes result in nearly linear growth.
    - **Insertion Sort** performs exceptionally fast on already sorted arrays, matching its theoretical best-case complexity of O(N).
    - **Merge Sort** shows significantly better scaling compared to other algorithms, especially for larger input sizes, consistent with its O(N log N) complexity.
- **Anomalies or Deviations:**
    - Insertion Sort appears to have extremely low times (even 0 ms) for very small arrays. This is expected because it is optimized for nearly sorted data and small N.
    - Minor inconsistencies in timing (e.g., 0.001 ms difference) can occur due to the precision limit of the clock() function used in C++ and the relatively small size of input arrays.

- **Worst-Case Input:**
    - Although arrays were already sorted (best-case for some algorithms), for worst-case analysis, reversing the arrays would cause Bubble, Selection, and Insertion Sort to take longer.

CODE:

```
#include <iostream>
#include <ctime>
using namespace std;

// Arrays
int arr1[] = {1, 2, 3, 4, 5};
int arr2[] = {1,2,3,4,5,6,7,8,9,10};
int arr3[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50};
int arr4[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100};

// Utility function to copy arrays
void copyArray(int source[], int destination[], int size) {
    for (int i = 0; i < size; ++i)
        destination[i] = source[i];
}
```

```
// Bubble Sort
void BubbleSort() {
    cout << "--- Bubble Sort ---" << endl;
    int sizes[] = {5, 10, 50, 100};
    int* arrays[] = {arr1, arr2, arr3, arr4};

    for (int i = 0; i < 4; ++i) {
        int temp[100];
        copyArray(arrays[i], temp, sizes[i]);

        clock_t start = clock();
        for (int j = 0; j < sizes[i]-1; ++j)
            for (int k = 0; k < sizes[i]-j-1; ++k)
                if (temp[k] > temp[k+1])
                    swap(temp[k], temp[k+1]);
        clock_t end = clock();

        double time_taken = double(end - start) * 1000.0 / CLOCKS_PER_SEC;
        cout << "Array of size " << sizes[i] << " sorted in " << time_taken << " ms" << endl;
    }
    cout << endl;
}

// Selection Sort
void SelectionSort() {
    cout << "--- Selection Sort ---" << endl;
    int sizes[] = {5, 10, 50, 100};
```

```
    int* arrays[] = {arr1, arr2, arr3, arr4};

    for (int i = 0; i < 4; ++i) {
        int temp[100];
        copyArray(arrays[i], temp, sizes[i]);

        clock_t start = clock();
        for (int j = 0; j < sizes[i]-1; ++j) {
            int min_idx = j;
            for (int k = j+1; k < sizes[i]; ++k)
                if (temp[k] < temp[min_idx])
                    min_idx = k;
            swap(temp[j], temp[min_idx]);
        }
        clock_t end = clock();

        double time_taken = double(end - start) * 1000.0 / CLOCKS_PER_SEC;
        cout << "Array of size " << sizes[i] << " sorted in " << time_taken << " ms" << endl;
    }
    cout << endl;
}

// Insertion Sort
void InsertionSort() {
    cout << "--- Insertion Sort ---" << endl;
    int sizes[] = {5, 10, 50, 100};
    int* arrays[] = {arr1, arr2, arr3, arr4};
```

```
    for (int i = 0; i < 4; ++i) {
        int temp[100];
        copyArray(arrays[i], temp, sizes[i]);

        clock_t start = clock();
        for (int j = 1; j < sizes[i]; ++j) {
            int key = temp[j];
            int k = j - 1;
            while (k >= 0 && temp[k] > key) {
                temp[k+1] = temp[k];
                k--;
            }
            temp[k+1] = key;
        }
        clock_t end = clock();

        double time_taken = double(end - start) * 1000.0 / CLOCKS_PER_SEC;
        cout << "Array of size " << sizes[i] << " sorted in " << time_taken << " ms" << endl;
    }
    cout << endl;
}

// Merge Sort Helper Functions
void merge(int arr[], int l, int m, int r) {
    int n1 = m-l+1;
    int n2 = r-m;
```

```
    int L[50], R[50];

    for (int i = 0; i < n1; i++) L[i] = arr[l+i];
    for (int j = 0; j < n2; j++) R[j] = arr[m+1+j];

    int i=0, j=0, k=l;
    while (i<n1 && j<n2) {
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    }
    while (i<n1)
        arr[k++] = L[i++];
    while (j<n2)
        arr[k++] = R[j++];
}

void mergeSort(int arr[], int l, int r) {
    if (l<r) {
        int m = l + (r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```

```
// Merge Sort
void MergeSort() {
    cout << "--- Merge Sort ---" << endl;
    int sizes[] = {5, 10, 50, 100};
    int* arrays[] = {arr1, arr2, arr3, arr4};

    for (int i = 0; i < 4; ++i) {
        int temp[100];
        copyArray(arrays[i], temp, sizes[i]);

        clock_t start = clock();
        mergeSort(temp, 0, sizes[i]-1);
        clock_t end = clock();

        double time_taken = double(end - start) * 1000.0 / CLOCKS_PER_SEC;
        cout << "Array of size " << sizes[i] << " sorted in " << time_taken << " ms" << endl;
    }
    cout << endl;
}

int main() {
    BubbleSort();
    SelectionSort();
    InsertionSort();
    MergeSort();
    return 0;
}
```

OUTPUT (C++ Compiler-Programmiz):

--- Bubble Sort ---

Array of size 5 sorted in 0.001 ms

Array of size 10 sorted in 0.001 ms

Array of size 50 sorted in 0.005 ms

Array of size 100 sorted in 0.014 ms

--- Selection Sort ---

Array of size 5 sorted in 0.001 ms

Array of size 10 sorted in 0.001 ms

Array of size 50 sorted in 0.004 ms

Array of size 100 sorted in 0.013 ms

--- Insertion Sort ---

Array of size 5 sorted in 0.001 ms

Array of size 10 sorted in 0 ms

Array of size 50 sorted in 0.001 ms

Array of size 100 sorted in 0.002 ms

--- Merge Sort ---

Array of size 5 sorted in 0.001 ms

Array of size 10 sorted in 0.002 ms

Array of size 50 sorted in 0.003 ms

Array of size 100 sorted in 0.007 ms

Conclusion

This project successfully demonstrates the empirical growth trends of four common sorting algorithms. The observed results align well with the theoretical time complexities, validating the behavior of these algorithms under best-case conditions.