Task (1)

Task (1): Sorting Comparison

Description: implementing a different sorting algorithm (Bubble Sort, Selection Sort, and Insertion Sort). The purpose of these functions that sort an array while counting how many comparisons are made during sorting.

Bubble Sort: Repeatedly compares adjacent elements, swaps if out of order.

Bubble sort

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

int bubble_sort_count(int arr[], int size) {
    int count = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            count++;
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
    return count;
}

int main() {
    int arr[] = {5, 3, 8, 4, 2};
    int n = 5;
    cout << "Number of comparisons: "
         << bubble_sort_count(arr, n) << endl;
}
```

Explosions: in this code our program that using Bubble Sort algorithm and counts the number of comparisons made while sorting a list of integers and it's help to know performance of Bubble Sort by measuring how many times elements are compared during the sorting process.

Selection Sort: Selects the smallest element and places it at the start.

Selection sort

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

int selection_sort(int arr[], int size) {
    int count = 0, min_index;
    for (int i = 0; i < size - 1; i++) {
        min_index = i;
        for (int j = i + 1; j < size; j++) {
            count++;
            if (arr[j] < arr[min_index])
                min_index = j;
        }
        swap(arr[i], arr[min_index]);
    }
    return count;
}

int main() {
    int arr[] = {1, 2, 0};
    cout << "Comparisons: "
         << selection_sort(arr, 3) << endl;
}
```

Explosions: in this code our program that using Selection Sort algorithm and counts the number of comparisons made while sorting a list of integers and it's help to know performance of Selection Sort by measuring how many times elements are compared during the sorting process.

Insertion Sort: Inserts each element in the correct position among sorted ones.

Insertion sort

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

int insertion_sort(int arr[], int size) {
    int count = 0, j, key;
    for (int i = 1; i < size; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            count++;
            arr[j + 1] = arr[j];
            j--;
        }
        if (j >= 0) count++;
        arr[j + 1] = key;
    }
    return count;
}

int main() {
    int arr[] = {5, 3, 4};
    cout << "Comparisons: "
         << insertion_sort(arr, 3) << endl;
}
```

Explosions: in this code our program that using insertion Sort algorithm and counts the number of comparisons made while sorting a list of integers and it's help to know performance of insertion Sort by measuring how many times elements are compared during the sorting process.

Complexity Analysis

what is Complexity analysis:

Complexity analysis is a way of measuring how efficient an algorithm is by determining how much time it takes to run and how much memory it uses as the input size increases.

Comparative table:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity	Type
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Iterative
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	Iterative
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)	Iterative

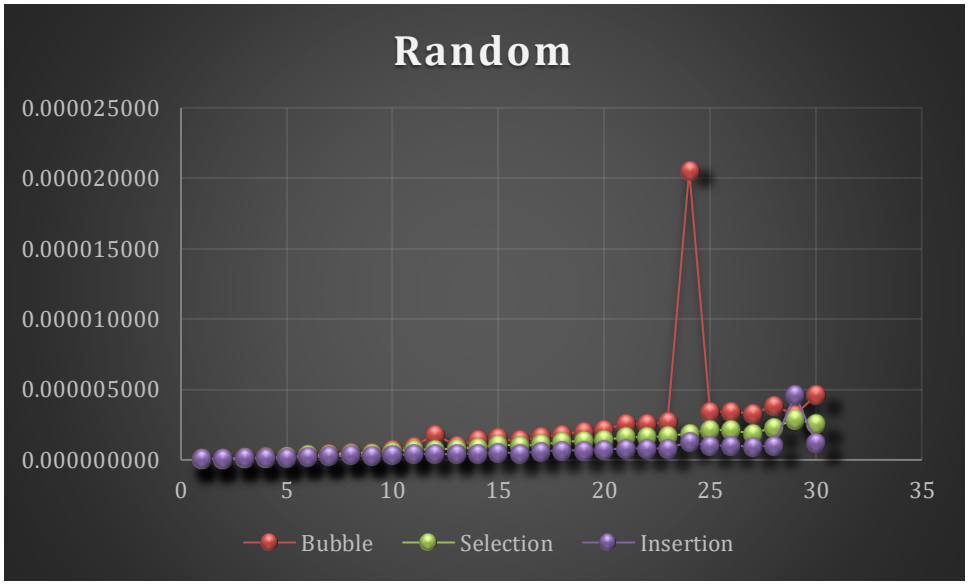
Explanation:

- **Bubble Sort**: Efficient only when the list is nearly in the best case. In the worst case, it requires many comparisons and swaps.
- **Selection Sort**: Always performs the same number of comparisons ($O(n^2)$), regardless of input order, but performs the fewest swaps.
- **Insertion Sort**: Performs very well on small or nearly sorted datasets, with O(n) best case.

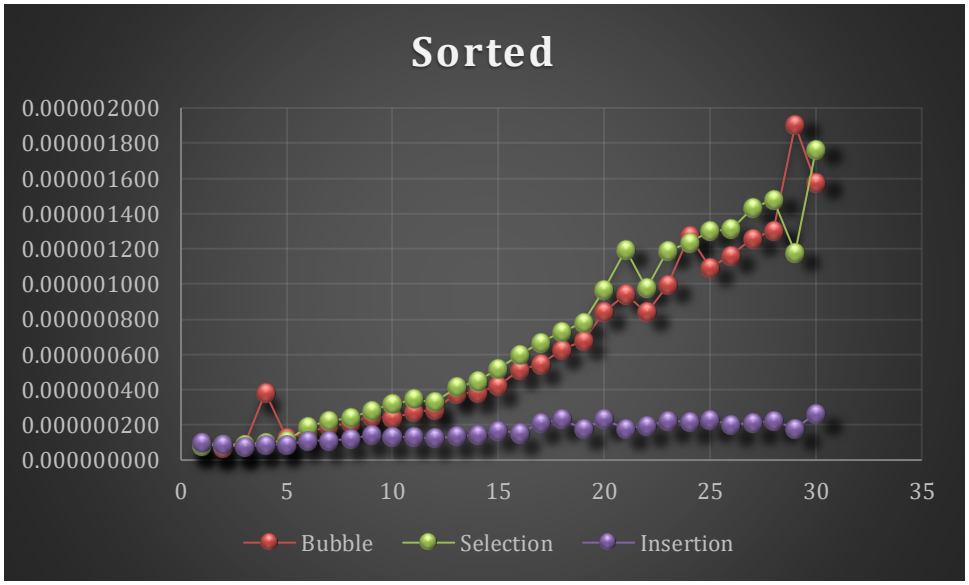
Comparison Between Iterative and Recursive Sorting Algorithms:

Feature	Iterative Sorting	Recursive Sorting
Sorting	Bubble Sort, Insertion Sort, Selection Sort	Merge Sort, Quick Sort
Approach	Uses loops to repeatedly compare and move elements	Divides the array into subproblems and solves them recursively
Speed (Average Case)	Usually $O(n^2)$ for basic sorts	Usually O(n log n) for divide-and-conquer sorts
Memory Usage	O(1) (uses constant extra memory)	O(log n) to O(n) (due to recursive call stack)
Implementation Difficulty	Simple and straightforward	More complex (requires understanding recursion and base cases)

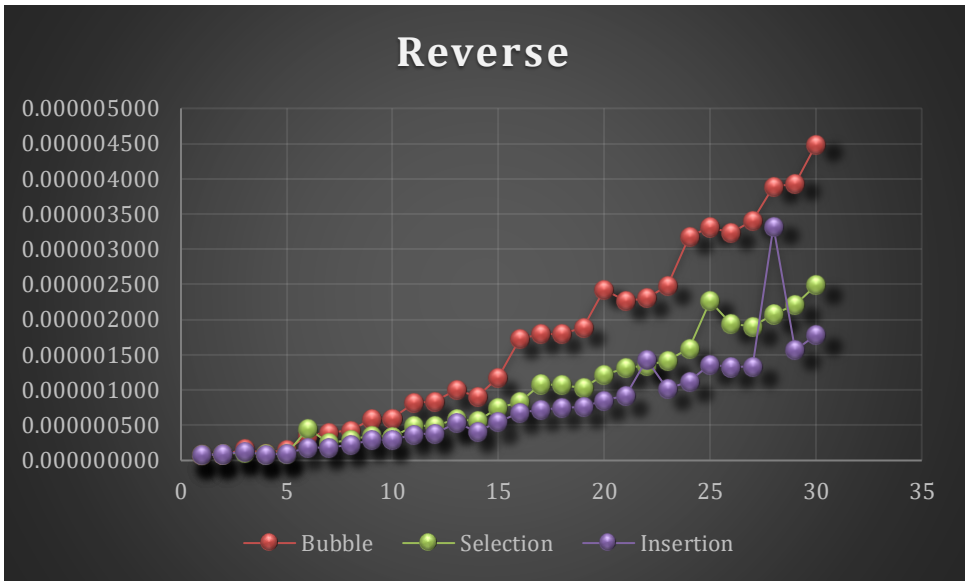
Results and Graphical Analysis



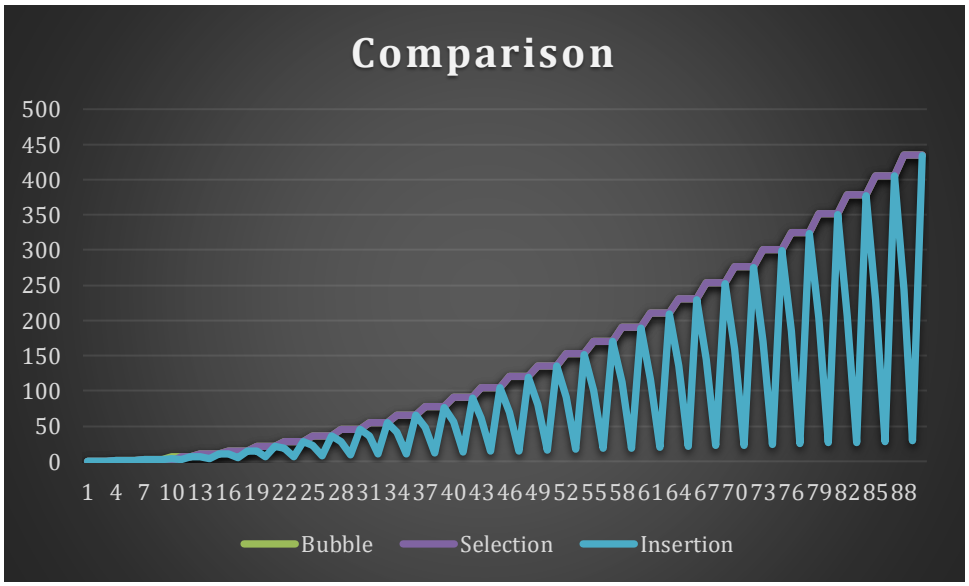
For random input data, all algorithms exhibit similar average runtimes with minor fluctuations, insertion and Selection Sort are slightly faster than Bubble Sort, the runtime growth confirms the expected $O(n^2)$ behavior.



When the input is already sorted, Insertion Sort performs best with nearly constant execution time O(n) best case, Bubble and Selection Sort still traverse most of the array, leading to higher runtimes.



In the reverse (worst-case) scenario, all algorithms show quadratic growth, bubble Sort takes the most time due to repeated swaps, while Insertion and Selection Sort are moderately faster.



The number of comparisons grows quadratically with array size for all algorithms, selection Sort and Bubble Sort have nearly identical comparison counts, while Insertion Sort performs fewer comparisons on nearly sorted data.

Description: Create a function called test_comparisons () that tests the three sorting functions (Bubble, Selection, and Insertion Sort) on arrays of different types and sizes. The function should generate 30 random arrays, 30 already sorted arrays, and 30 inversely sorted arrays, with sizes ranging from 1 to 30. For each array size and type, the program must calculate both the number of comparisons and the execution time (in seconds) for each sorting algorithm. The results should be displayed neatly on the console and also saved in an Excel workbook

Test comparisons

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <chrono>
using namespace std;
using namespace std::chrono;
int bubble_sort_count(int arr[], int size) {
    int count = 0;
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            count++;
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
            }
        }
    }
    return count;
}
int insertion_sort(int arr[], int size) {
    int count = 0, j, key;
    for (int i = 1; i < size; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            count++;
```

```
                arr[j + 1] = arr[j];
                j--;
            }
            if (j >= 0) count++;
            arr[j + 1] = key;
        }
        return count;
    }
int selection_sort(int arr[], int size) {
        int count = 0, min_index;
        for (int i = 0; i < size - 1; i++) {
            min_index = i;
            for (int j = i + 1; j < size; j++) {
                count++;
                if (arr[j] < arr[min_index])
                    min_index = j;
            }
            swap(arr[i], arr[min_index]);
        }
        return count;
    }
void test_comparisons() {
        srand(time(0));
        ofstream fileComparisons("comparisons.csv");
        ofstream fileTimes("times.csv");
        fileComparisons << "Size,Type,Bubble,Selection,Insertion\n";
        fileTimes << "Size,Type,Bubble,Selection,Insertion\n";
        for (int size = 1; size <= 30; size++) {
            vector<string> types = { "Random", "Sorted", "Reverse" };
            for (string type : types) {
                long long bubbleCmp = 0, selectCmp = 0, insertCmp = 0;
                double bubbleTime = 0, selectTime = 0, insertTime = 0;
                for (int test = 0; test < 30; test++) {
                    vector<int> arr(size);
                    for (int i = 0; i < size; i++) arr[i] = rand() % 100;
```

```
                if (type == "Sorted") sort(arr.begin(), arr.end());
                if (type == "Reverse") sort(arr.begin(), arr.end(),
greater<int>());

                vector<int> tmp = arr;
                auto s = high_resolution_clock::now();
                bubbleCmp += bubble_sort_count(tmp.data(), size);
                auto e = high_resolution_clock::now();
                bubbleTime += duration<double>(e - s).count();

                tmp = arr;
                s = high_resolution_clock::now();
                selectCmp += selection_sort(tmp.data(), size);
                e = high_resolution_clock::now();
                selectTime += duration<double>(e - s).count();

                tmp = arr;
                s = high_resolution_clock::now();
                insertCmp += insertion_sort(tmp.data(), size);
                e = high_resolution_clock::now();
                insertTime += duration<double>(e - s).count();
            }

            fileComparisons << size << "," << type << ","
                << bubbleCmp / 30 << ","
                << selectCmp / 30 << ","
                << insertCmp / 30 << "\n";

            fileTimes << size << "," << type << ","
                << bubbleTime / 30.0 << ","
                << selectTime / 30.0 << ","
                << insertTime / 30.0 << "\n";
        }
    }
    cout << "Results saved to CSV files.\n";
}
```

```
int main() {
    test_comparisons();
    return 0;
}
```