Task 1: Analysis of Sorting Algorithms

Overview

This report examines the operational mechanics and performance characteristics of three fundamental sorting algorithms: **Bubble Sort**, **Selection Sort**, and **Insertion Sort**. Each algorithm's efficiency is evaluated based on its time and space complexity.

1. Bubble Sort

Bubble Sort functions by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the incorrect order. This process is repeated until the list is sorted.

- **Time Complexity**: The algorithm's time complexity is Big O (n^2) in the average and worst cases, making it inefficient for large datasets. Its best-case performance is Big O (n) which occurs if the array is already sorted.
- **Space Complexity**: The algorithm sorts the elements in place giving it a space complexity of Big O (1).
- **Observations**: The provided test results on an array of 30 elements align with this theoretical analysis.
    - **Best Case (Sorted Array)**: Required only 29 comparisons and a negligible execution time (0 ms).
    - **Worst Case (Reverse-Sorted Array)**: The number of comparisons jumped to 435, and the execution time was 2 ms.

2. Selection Sort

This algorithm divides the input into a sorted and an unsorted section. It repeatedly finds the minimum element from the unsorted section and swaps it with the first element of that section, effectively expanding the sorted portion.

- **Time Complexity**: Selection Sort has a consistent time complexity of Big O (n^2) in all scenarios (best, average, and worst). This is because it must always scan the entire unsorted section to find the minimum element, regardless of the initial order.
- **Space Complexity**: As an **in-place** sorting algorithm, its space complexity Big O (1).
- **Observations**: The provided test results on an array of 30 elements align with this theoretical analysis.
    - **Best Case (Sorted Array)**: Required 435 comparisons and an execution time (2ms).
    - **Worst Case (Reverse-Sorted Array)**: The number of comparisons remained 435, and the execution time was 1 ms.

3. Insertion Sort

Insertion Sort builds the final sorted array one item at a time. It iterates through the input elements and inserts each element into its correct position within the already sorted part of the array. To do this, it shifts all larger elements one position to the right.

- **Time Complexity**: The worst-case and average-case time complexity is Big O (n^2). However, it is highly efficient for lists that are already substantially sorted, achieving a best-case time complexity of Big O (n).
- **Space Complexity**: It operates **in-place**, resulting in a space complexity of Big O (1).
- **Observations**: The provided test results on an array of 30 elements align with this theoretical analysis.
    - **Best Case (Sorted Array)**: Required only 29 comparisons and a negligible execution time (0 ms).
    - **Worst Case (Reverse-Sorted Array)**: The number of comparisons jumped to 435, and the execution time was 2 ms.

4. Summary:

As a summary, here is a table to sum up the average run time and number of comparisons across the different arrays and different array orders.

Average time:

	Sorted	Reverse sorted	Random
Bubble sort	0 ms	0.4 ms	0.76 ms
Selection sort	0.4 ms	0.2 ms	0.4 ms
Insertion sort	0 ms	0.13 ms	0.06 ms

Average number of comparisons:

	Sorted	Reverse sorted	Random
Bubble sort	15	150	150
Selection sort	150	150	150
Insertion sort	15	150	86

As the table suggests, the worst algorithm with the highest average time and number of comparisons is selection sort as it's mechanism is to compare the whole array without any optimizations.

4. Additional functions

To complete this analysis there has to be some helper functions including:

- **void sortedData():** populates the 2D array with sorted elements.
- **void reverseSorted():** reverses the order of the sorted 2D array.
- **void randPopulating():** populates the 2D array using the rand() func from cstdlib library.

The complexity of all these algorithms are Big O(1) although they run by a nested loop to populate the array, but the number of operations is constant.