A picture containing text, tableware, plate, dishware

Description automatically generated

**Design and Analysis of Algorithms**

**Program**  BSAI

**Semester**  6th

**Department**  Computer Science

**Submitted to** Sir Waqar Ahmed

**Group Members and Registration No.**

Kanza Kashaf 22-NTU-CS-1350

Hassaan Raza 22-NTU-CS-1362

Sorting Algorithms Report

## Introduction

Sorting is a fundamental operation in computer science that arranges the elements of an array in a specific order (ascending or descending). This report examines four sorting algorithms implemented in the project:

1. Insertion Sort
2. Quick Sort
3. Heap Sort
4. Counting Sort

The program allows for dynamic input, measures execution time for sorting, and displays the results interactively.

## 1. Insertion Sort

### Working

Insertion Sort is a simple, comparison-based algorithm. It builds the sorted array one element at a time by comparing and inserting each element into its correct position.

### Steps

1. Start with the first element, considering it sorted.
2. Take the next element and compare it with the elements in the sorted portion.
3. Insert it in the correct position by shifting elements.
4. Repeat until all elements are sorted.

### Complexity

* Best Case: O(n) (already sorted array)
* Worst Case: O(n^2) (reverse sorted array)
* Average Case: O(n^2)

## 2. Quick Sort

### Working

Quick Sort is a divide-and-conquer algorithm. It selects a pivot and partitions the array into two subarrays: elements smaller than the pivot and elements greater than the pivot. It then recursively sorts the subarrays.

### Steps

1. Pick a pivot element.
2. Partition the array around the pivot.
3. Recursively apply Quick Sort on the left and right partitions.

### Complexity

* Best Case: O (n log n)
* Worst Case: O (n^2) (when the pivot divides the array poorly, e.g., smallest or largest element)
* Average Case: O (n log n)

## 3. Heap Sort

### Working

Heap Sort uses a binary heap data structure to repeatedly extract the maximum element and rebuild the heap.

### Steps

1. Build a max-heap from the input array.
2. Extract the maximum element (root) and swap it with the last element.
3. Reduce the heap size and rebuild the heap.
4. Repeat until the heap is empty.

### Complexity

* Best Case: O (n log n)
* Worst Case: O (n log n)
* Average Case: O (n log n)

## 4. Counting Sort

### Working

Counting Sort is a non-comparison-based sorting algorithm that works by counting the frequency of each element and placing them at their correct positions.

### Steps

1. Count the occurrences of each element in a frequency array.
2. Compute the cumulative frequency.
3. Place each element in its sorted position using the cumulative frequency array.

### Complexity

* Time Complexity: O (n + k), where k is the range of input values.
* Space Complexity: O (n + k)

# Performance Analysis

Sorting time of these Algorithms on different input sizes:

## First Observation:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Insertion Sort | Quick Sort | Heap Sort | Counting Sort |
| 10 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 50 | 0.0333 | 0.0000 | 0.0333 | 0.0000 |
| 100 | 0.0667 | 0.0000 | 0.0333 | 0.0667 |
| 1000 | 1.5333 | 0.2667 | 0.7000 | 0.0667 |
| 5000 | 10.0000 | 0.5333 | 1.0333 | 0.1333 |
| 10000 | N/A (Too large) | 1.0000 | 2.0667 | 0.1667 |
| 100000 | N/A (Too large) | 12.5000 | 30.3333 | 2.2000 |
| 1000000 | N/A (Too large) | 228.3000 | 446.5000 | N/A (Too large) |

## Second Obseration:

Here is the data presented in the table format as requested:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Size | Insertion Sort | Quick Sort | Heap Sort | Counting Sort |
| 10 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 50 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| 100 | 0.0000 | 0.0333 | 0.1000 | 0.0000 |
| 1000 | 1.3000 | 0.4000 | 0.5000 | 0.0000 |
| 5000 | 9.3000 | 0.3667 | 0.9333 | 0.0667 |
| 10000 | N/A (Too large) | 0.8667 | 2.2000 | 0.1000 |
| 100000 | N/A (Too large) | 11.0667 | 28.3000 | 1.9667 |
| 1000000 | N/A (Too large) | 211.3333 | 444.0333 | N/A (Too large) |

# Analyze the Data

## Observations:

### Insertion Sort:

* + Performs well for small sizes but becomes inefficient quickly. It cannot handle sizes above **5000**.

### Quick Sort:

* + Performs well across all input sizes. Execution time increases logarithmically as size grows.

### Heap Sort:

* + Handles larger inputs but is slower compared to Quick Sort.

### Counting Sort:

* + Performs extremely well when the range of values is small, but it cannot handle very large input sizes like **1,000,000**.