14. Sorting Algorithms Selection, Bubble, and Insertion Sort

Table of Contents

Selection Sort

Concept: Repeatedly select the smallest element from the unsorted portion and swap it with the first unsorted element.
Time Complexity: $O(n^2)$ in all cases (Best/Avg/Worst).
Use Case: Small datasets or when memory writes are expensive.

Steps:

1. Find the minimum element in the unsorted array.
2. Swap it with the first unsorted element.
3. Repeat for the remaining unsorted portion.

```cpp
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        int minIdx = i;
        for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        swap(arr[i], arr[minIdx]);
    }
}
```

Bubble Sort

Concept: Repeatedly swap adjacent elements if they are in the wrong order.
Time Complexity:

- Worst/Avg: $O(n^2)$
- Best (Optimized): $O(n)$ (if array is already sorted).
  Use Case: Simple implementation, small datasets, or nearly sorted arrays.

Steps:

1. Compare adjacent elements. Swap if `arr[j] > arr[j+1]`.
2. After each pass, the largest element "bubbles" to the end.
3. Optimize with early termination if no swaps occur.

```cpp
void bubbleSort(int arr[], int n) {
    bool didSwap;
    for (int i = 0; i < n-1; i++) {
        didSwap = false;
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
                didSwap = true;
            }
        }
        if (!didSwap) break; // Early exit if sorted
    }
}
```

Insertion Sort

Concept: Build the sorted array by inserting each element into its correct position.
Time Complexity:

- Worst/Avg: $O(n^2)$

- Best: O(n) (if array is sorted).
  Use Case: Small datasets, nearly sorted arrays, or online sorting.

Steps:

1. Start from the second element (index 1).
2. Compare with previous elements and shift them right if larger.
3. Insert the current element into its correct position.

```cpp
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i-1;
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

Key Comparisons

Algorithm	Best Case	Avg/Worst Case	Space	Stability	Use Case
Selection	$O(n^2)$	$O(n^2)$	O(1)	No	Small data, minimal swaps
Bubble	O(n)	$O(n^2)$	O(1)	Yes	Simple logic, nearly sorted
Insertion	O(n)	$O(n^2)$	O(1)	Yes	Small data, online sorting

Key Notes

1. **Selection Sort**:
    - Always $O(n^2)$ (no early termination).
    - Minimizes swaps (useful when writes are costly).
    - Unstable (swaps can change order of equal elements).
2. **Bubble Sort**:
    - Stable (equal elements retain order).
    - Optimized with `didSwap` flag for early exit.
    - Least efficient in practice for large datasets.
3. **Insertion Sort**:
    - Stable and adaptive (efficient for nearly sorted data).
    - Used in hybrid algorithms like Timsort.
    - Ideal for online sorting (data arriving sequentially).
4. **Placement Interviews**:
    - Focus on understanding time complexity and trade-offs.
    - Practice writing code for all three from scratch.
    - Explain optimization steps (e.g., early termination in Bubble Sort).