**SORTING**

# Bubble sort

## Overview

Bubble Sort: A simple, basic sorting algorithm where the largest value "bubbles up" to the end of the list through multiple passes.

Part of the elementary sorting algorithms: These are straightforward but less efficient compared to advanced algorithms like Merge Sort or Quick Sort.

## How It Works

- Compare Adjacent Elements:

* Compare the first two elements.
* If the first element is larger than the second, swap them.
* Continue this process for all adjacent elements in the array.

- Bubble Up the Largest Value:

* After one pass, the largest value will be at the end of the array.
* Repeat the process for the remaining unsorted portion of the array.

- Repeat Until Sorted:

* Keep passing through the array until no swaps are needed.

## Time and Space Complexity

- Time Complexity:

* Best Case: O(n) when the array is already sorted.
* Average/Worst Case: O(n^2) due to nested loops.

- Space Complexity: O(1) no extra data structures used.

# Selection Sort

## Overview

Selection Sort: Another simple sorting algorithm that works by finding the smallest element in the list and placing it in its correct position, starting from the beginning.

Like Bubble Sort, it has a time complexity of O(n^2) and is not efficient for large datasets.

## How It Works

- Find the smallest element: Start from the first index and iterate through the array to find the smallest element.

- Swap the smallest element with the element at the current index.

- Repeat for remaining array: Move to the next index and repeat the process for the unsorted portion of the array.

- Continue until sorted: Repeat until the entire array is sorted.

## Time and Space Complexity

- Time Complexity: Best, Average, and Worst Case: O(n^2) due to nested loops.

- Space Complexity: O(1) no extra space used beyond the input array.

# Insertion Sort

## Overview

Insertion Sort: A sorting algorithm that works similarly to how people often sort cards in their hands.

It builds the sorted array one element at a time by inserting elements into their correct positions.

## How It Works

- Start with the second element:

* Assume the first element is sorted.
* Take the second element and compare it with the first.
* Insert it into the correct position.

- Continue with the next element:

* Take the next element and compare it with the elements in the sorted portion of the array.
* Shift larger elements one position to the right to make space for the new element.
* Insert the new element in its correct position.

- Repeat: Continue until all elements are placed in their correct positions.

## Time and Space Complexity

- Time Complexity:

* Best Case: O(n) when the array is nearly sorted.
* Average/Worst Case: O(n^2) due to nested loops.

- Space Complexity: O(1) no extra data structures used.

# Merge sort

## Overview

Merge Sort: A divide-and-conquer sorting algorithm.

Divides the array into smaller parts, sorts each part, and merges them back together in sorted order.

## How It Works

- Divide the Array: Recursively split the array into halves until each subarray has only one element.

- Merge the Subarrays: Compare the elements of two subarrays and merge them into a single sorted array.

## Time and Space Complexity

- Time Complexity: O(nlogn)

* Dividing the array takes O(logn).
* Merging the arrays takes O(n) for each level of recursion.

- Space Complexity: O(n) temporary arrays are used to store divided portions during the merge process.

## Implementation Skeleton

- Merge Sort Function:

* Base case: If the array has one element, return it.
* Recursively split the array into left and right halves.
* Merge the sorted halves using a helper function.

- Merge Function:

* Compare elements from the left and right subarrays.
* Add the smaller element to the result array.
* Append any remaining elements from both subarrays.

# Quick sort

## Overview

QuickSort is a divide-and-conquer algorithm that sorts an array by selecting a **pivot** and partitioning the array into two sub-arrays:

* Elements less than the pivot go to the left.
* Elements greater than the pivot go to the right.

This process is repeated recursively until the entire array is sorted.

## How It Works

- Choose a Pivot: Select a pivot element from the array. Common choices include the first element, the last element, or a random element.

- Partition the Array: Rearrange elements so that those smaller than the pivot are to its left and those larger are to its right.

- Recursively Apply QuickSort: Apply the same process to the left and right sub-arrays.

- Combine Results: Once all sub-arrays are sorted, the entire array becomes sorted.

## Time and Space Complexity

- Time Complexity:

* Best Case: O(nlogn) the array is split evenly at each step.
* Average Case: O(nlogn) random distribution of elements with balanced splits.
* Worst Case: O(n^2) pivot causes highly unbalanced splits (e.g., sorted arrays).

- Space Complexity: O(logn) due to recursive function calls.

A screenshot of a computer screen

Description automatically generated

Slide : Array sorting algorithms