Bubble Sort

- Bubble sort is a simple comparison-based sorting algorithm.
- It repeatedly compares adjacent elements and swaps them if they are in the wrong order.
- The largest (or smallest, depending on the sorting order) element "bubbles" to the end of the array in each pass.
- Bubble sort is easy to understand and implement, but it is not the most efficient sorting algorithm for large datasets.
- **Time Complexity:**
    - **Average Case: O(n^2)**
    - **Worst Case: O(n^2)**
    - **Best Case: O(n) when the array is already sorted (with an optimized implementation)**
- **Space Complexity: O(1) - Bubble sort sorts the array in-place without requiring additional memory.**
- **Stable Sorting**: Bubble sort is a stable sorting algorithm, which means it maintains the relative order of equal elements.
- It is an in-place sorting algorithm, meaning it does not require extra memory beyond the input array.
- Bubble sort can be optimized by introducing a flag to track whether any swaps were made in a pass, terminating the algorithm early if the array is already sorted.
- Recursive implementation of bubble sort is possible, but it is generally less efficient than the iterative version and other sorting algorithms.

Remember, while bubble sort is a good algorithm to understand sorting concepts, it is not the most efficient choice for large arrays. It's important to be familiar with more efficient sorting algorithms such as quicksort, merge sort, or heapsort for practical use.

- Bubble sort is a stable sorting algorithm, meaning it preserves the relative order of elements with equal values. If there are two elements with the same value, the one that appears earlier in the original array will also appear earlier in the sorted array.
- Bubble sort is a comparison-based sorting algorithm, which means it relies on comparing elements to determine their order.
- The number of passes required by bubble sort to sort an array is equal to the number of elements in the array minus one.

- Bubble sort works well for small-sized arrays or partially sorted arrays, but it becomes inefficient for large-sized arrays due to its quadratic time complexity.
- In the worst-case scenario, where the input array is in reverse sorted order, bubble sort requires the maximum number of comparisons and swaps.
- Bubble sort can be easily implemented in various programming languages due to its simplicity and straightforward logic.
- Although bubble sort is not the most efficient sorting algorithm, it can be useful in certain scenarios, such as when the input array is already almost sorted or when simplicity and ease of implementation are prioritized over performance.
- Bubble sort is an adaptive algorithm, meaning it can take advantage of pre-sorted or partially sorted arrays, reducing the number of comparisons and swaps required.
- Bubble sort is often used as a teaching tool to introduce the concept of sorting algorithms and the fundamental idea of comparing and swapping elements.

Here's the step-by-step visualization of the optimized bubble sort algorithm with the given array [7, 2, 4, 1, 5]:

1. Pass 1:
    - Compare 5 and 2: Swap (5, 2) -> [2, 5, 8, 1, 3]
    - Compare 5 and 8: No swap -> [2, 5, 8, 1, 3]
    - Compare 8 and 1: Swap (8, 1) -> [2, 5, 1, 8, 3]
    - Compare 8 and 3: Swap (8, 3) -> [2, 5, 1, 3, 8]
    - After the first pass: [2, 5, 1, 3, 8]
2. Pass 2:
    - Compare 2 and 5: No swap -> [2, 5, 1, 3, 8]
    - Compare 5 and 1: Swap (5, 1) -> [2, 1, 5, 3, 8]
    - Compare 5 and 3: Swap (5, 3) -> [2, 1, 3, 5, 8]
    - After the second pass: [2, 1, 3, 5, 8]
3. Pass 3:
    - Compare 2 and 1: Swap (2, 1) -> [1, 2, 3, 5, 8]
    - Compare 2 and 3: No swap -> [1, 2, 3, 5, 8]
    - After the third pass: [1, 2, 3, 5, 8]
4. Pass 4:

- Compare 1 and 2: No swap -> [1, 2, 3, 5, 8]
- After the fourth pass: [1, 2, 3, 5, 8]

Since there were no more swaps in the last pass, the array is sorted.

Approach 1: Function to perform bubble sort iteratively

- The **bubbleSort** function implements the bubble sort algorithm iteratively. It takes a reference to a vector **arr** and the size of the array **n** as parameters.
- The outer loop **for(int i = 0; i < n; i++)** represents the number of passes. It iterates from 0 to **n-1**, indicating the current pass.
- The **isSwapped** variable is used to track whether any swaps were made during a pass. It is initially set to **false**.
- The inner loop **for(int j = 0; j < n - i - 1; j++)** represents the comparisons and swaps within each pass. It iterates from 0 to **n-i-1**, comparing adjacent elements and swapping them if they are in the wrong order.
- If a swap occurs during a pass, the **isSwapped** flag is set to **true**.
- If no swaps were made during a pass, it means the array is already sorted, and the loop is terminated early using the **break** statement.

Approach 2: Function to perform bubble sort Recursively

- The **bubbleSortRecursive** function implements the bubble sort algorithm recursively. It takes a reference to a vector **arr** and the size of the array **n** as parameters.
- The base case for the recursion is **if(n == 1) return;**. If there is only one element remaining, the function returns.
- The **isSwapped** variable is used to track whether any swaps were made during a pass. It is initially set to **false**.
- The inner loop **for(int j = 0; j < n - 1; j++)** performs the comparisons and swaps, similar to the iterative approach.
- If a swap occurs during a pass, the **isSwapped** flag is set to **true**.
- If no swaps were made during a pass, it means the array is already sorted, and the function returns.
- The function then recursively calls itself with a reduced size of the array (**n-1**), continuing the sorting process.