Selection Sort

- Selection sort is a simple and intuitive comparison-based sorting algorithm.
- It works by dividing the input array into two parts: the sorted part and the unsorted part.
- The sorted part is gradually built by repeatedly selecting the minimum (or maximum) element from the unsorted part and placing it at the correct position in the sorted part.
- This process continues until the entire array is sorted.

Key Steps:

1. Find the minimum (or maximum) element in the unsorted part of the array.
2. Swap the minimum element with the first element of the unsorted part, placing it in its correct position in the sorted part.
3. Expand the sorted part by moving the boundary one element ahead.
4. Repeat steps 1-3 until the entire array is sorted.

Key Points:

- **Time Complexity: Selection sort has a time complexity of O(n^2) in all cases, which makes it inefficient for large datasets.**
- **Space Complexity: It is an in-place sorting algorithm, meaning it doesn't require additional memory beyond the input array.**
- **Stability: Selection sort is not stable, meaning that the relative order of equal elements may change during the sorting process.**
- **Comparison Count: The number of comparisons performed by selection sort is the same regardless of the initial order of the elements.**

Here's a step-by-step explanation of how selection sort works

Example Array: [5, 3, 8, 2, 1]

Step 1:

- Initially, the entire array is unsorted.
- The minimum element is found in the unsorted part, which is 1.
- Swap the minimum element with the first element of the unsorted part.
- Updated array: [1, 3, 8, 2, 5]

Step 2:

- The first element is now sorted.
- Find the minimum element in the remaining unsorted part, which is 2.
- Swap the minimum element with the second element of the unsorted part.
- Updated array: [1, 2, 8, 3, 5]

Step 3:

- The first two elements are now sorted.
- Find the minimum element in the remaining unsorted part, which is 3.
- Swap the minimum element with the third element of the unsorted part.
- Updated array: [1, 2, 3, 8, 5]

Step 4:

- The first three elements are now sorted.
- Find the minimum element in the remaining unsorted part, which is 5.
- Swap the minimum element with the fourth element of the unsorted part.
- Updated array: [1, 2, 3, 5, 8]

Step 5:

- The entire array is now sorted.

Approach 1: Selection Sort - Iterative Approach

1. The **selectionSort** function implements the selection sort algorithm in an iterative manner.
2. It takes a reference to a vector of integers (**vector<int> &arr**) as input.
3. The outer loop runs from index 0 to **arr.size() - 1** to divide the array into sorted and unsorted parts.
4. Inside the outer loop, a variable **arrayMinIndex** is initialized with the current index **i**, assuming it to be the index of the minimum element in the unsorted part.
5. A nested loop starts from **i + 1** and iterates till the end of the array to find the actual minimum element in the unsorted part.
6. In each iteration of the nested loop, it checks if the element at the current index **j** is smaller than the assumed minimum element at **arrayMinIndex**.
7. If a smaller element is found, the **arrayMinIndex** is updated with the new minimum index **j**.

8. After the nested loop completes, the minimum element in the unsorted part is found.
9. The minimum element is then swapped with the first element of the unsorted part, placing it in its correct sorted position.
10. This process continues with the outer loop incrementing **i**, expanding the sorted part and reducing the unsorted part of the array.
11. Finally, the array is sorted in ascending order.

Approach 2: Selection Sort - Recursive Approach

1. The **selectionSortRecursive** function implements the selection sort algorithm using recursion.
2. It takes a reference to a vector of integers (**vector<int> &arr**) and an optional parameter **index** as input.
3. The base case of the recursive function is when the index reaches the last element of the array (**index == arr.size() - 1**).
4. Inside the function, a variable **arrayMinIndex** is initialized with the current index **index**, assuming it to be the index of the minimum element in the unsorted part.
5. A loop starts from **index + 1** and iterates till the end of the array to find the actual minimum element in the unsorted part.
6. In each iteration of the loop, it checks if the element at the current index **j** is smaller than the assumed minimum element at **arrayMinIndex**.
7. If a smaller element is found, the **arrayMinIndex** is updated with the new minimum index **j**.
8. After the loop completes, the minimum element is found in the unsorted part.
9. The minimum element is then swapped with the first element of the unsorted part, placing it in its correct sorted position.
10. The function calls itself recursively with the next index (**index + 1**), allowing the algorithm to sort the remaining unsorted part.
11. This recursion continues until the base case is reached, and the entire array is sorted in ascending order.