**Selection Sort**

public class SelectionSort{

public static void selectionSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

}

public static void main(String[] args){

int[] arr = {64, 25, 12, 22, 11};

selectionSort(arr);

System.out.println("Sorted array is");

for (int num : arr){

System.out.print(num + " ");

}

}

}

Output:



Worst-case time complexity: \( O(n^2) \) – Happens when the array is in reverse order. It always checks every element in the array, regardless of its initial order.

Best-case time complexity: \( O(n^2) \) – Even if the array is already sorted, Selection Sort still performs the same number of comparisons.

Average-case time complexity: \( O(n^2) \) – Same as the worst-case because the number of comparisons remains consistent across all scenarios.

Space complexity: \( O(1) \) – It sorts the array in-place, requiring only a constant amount of extra space.

So, Selection Sort is inefficient for large datasets, especially compared to more advanced algorithms like Merge Sort or Quick Sort, but it has the advantage of being space-efficient with \( O(1) \) space complexity.