Arrays_Sorting Techniques:Day 4

By Kavuri Santosh Kumar

In Java, there are several sorting techniques that can be used to sort data. These can be categorized into built-in sorting methods and custom implementations of sorting algorithms.

---

1. Built-in Sorting Techniques

Java provides the following built-in methods for sorting:

a. Arrays.sort()

- Used to sort arrays (primitive and object types).
- Implements **Dual-Pivot QuickSort** for primitive types and **TimSort** for objects.
- Example:
- int[] arr = {5, 3, 8, 6};
- Arrays.sort(arr); // Sorts the array in ascending order.

b. Collections.sort()

- Used to sort collections like List.
- Implements **MergeSort** or **TimSort** depending on the underlying implementation.
- Example:
- List<Integer> list = Arrays.asList(5, 3, 8, 6);
- Collections.sort(list); // Sorts the list in ascending order.

c. List.sort()

- A method introduced in Java 8.
- Can sort a list based on a custom comparator.
- Example:
- list.sort((a, b) -> a - b); // Sorts using a custom comparator.

---

2. Custom Sorting Algorithms

Java allows you to implement common sorting algorithms manually. Some commonly used sorting techniques are:

a. Bubble Sort

- Repeatedly swaps adjacent elements if they are in the wrong order.
- Time Complexity: O(n2) in the worst case.

b. Selection Sort

- Selects the smallest element from the unsorted part and swaps it with the first unsorted element.
- Time Complexity: O(n2)

c. Insertion Sort

- Builds the sorted portion of the array one element at a time by inserting elements in their correct position.
- Time Complexity: O(n2).

d. Merge Sort

- Divides the array into halves, recursively sorts them, and then merges them.
- Time Complexity: O(nlogn)

e. Quick Sort

- Selects a "pivot" element, partitions the array around the pivot, and recursively sorts the partitions.
- Time Complexity: O(nlogn) on average.

f. Heap Sort

- Converts the array into a heap structure and repeatedly extracts the maximum (or minimum) element.
- Time Complexity: O(nlogn).

g. Radix Sort

- A non-comparative algorithm that sorts based on the digits of numbers, processing from least significant to most significant digit.
- Time Complexity: O(nk) where k is the number of digits.

h. Counting Sort

- Counts the occurrences of each element and uses this information to place elements in sorted order.
- Time Complexity: O(n+k) where k is the range of the input.

i. Bucket Sort

- Divides the elements into buckets and sorts each bucket individually, then concatenates them.
- Time Complexity: O(n+k).

j. Shell Sort

- A variation of insertion sort that uses a gap sequence to improve performance.
- Time Complexity: O(nlog2n).

---

3. Parallel Sorting

Arrays.parallelSort()

- Introduced in Java 8, it uses a parallel version of MergeSort or QuickSort to utilize multiple CPU cores for faster sorting.
- Example:
- int[] arr = {5, 3, 8, 6};
- Arrays.parallelSort(arr); // Uses parallelism for faster sorting.

Summary of Sorting Techniques:

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble Sort	O(n)	O(n2)	O(n2)	O(1)
Selection Sort	O(n2)	O(n2)	O(n2)	O(1)
Insertion Sort	O(n)	O(n2)	O(n2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n2)	O(logn)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

These sorting techniques offer a wide range of choices depending on the type of data, performance requirements, and memory constraints.

Bubble sort:

Question 1:

Write a program to sort an array of integers in ascending order. The program should accept the number of elements in the array and the elements themselves as input, and then display the sorted array as output. Using Bubble sort.?

Input Format:

1. An integer N representing the size of the array ($1 \leq N \leq 100$).
2. N space-separated integers representing the elements of the array.

Output Format:

A single line of space-separated integers representing the sorted array.

Sample Input:

5
34 12 7 89 56

Sample Output:

7 12 34 56 89

Explanation:

The input array [34, 12, 7, 89, 56] is sorted in ascending order to produce [7, 12, 34, 56, 89].

Selection Sort:

Question 2:

Write a program to sort an array of integers in ascending order. The program should accept the number of elements in the array and the elements themselves as input, and then display the sorted array as output. Use **Selection Sort**.

Input Format:

1. An integer N representing the size of the array ($1 \leq N \leq 100$)
2. N space-separated integers representing the elements of the array.

Output Format:

A single line of space-separated integers representing the sorted array.

Sample Input:

5
34 12 7 89 56

Sample Output:

7 12 34 56 89

Explanation:

The input array [34, 12, 7, 89, 56] is sorted in ascending order using the **Selection Sort** algorithm to produce [7, 12, 34, 56, 89].

Insertion Sort:

Question 2:

Write a program to sort an array of integers in ascending order. The program should accept the number of elements in the array and the elements themselves as input, and then display the sorted array as output. Use **Insertion Sort**.

Input Format:

1. An integer N representing the size of the array ($1 \leq N \leq 100$).
2. N space-separated integers representing the elements of the array.

Output Format:

A single line of space-separated integers representing the sorted array.

Sample Input:

5
34 12 7 89 56

Sample Output:

7 12 34 56 89

Explanation:

The input array [34, 12, 7, 89, 56] is sorted in ascending order using the **Insertion Sort** algorithm to produce [7, 12, 34, 56, 89].

Bubble sort

Code:

```
import java.util.Arrays;
class Main {
    public static void main(String[] args) {
        int arr[]={4,2,1,3,5};
        int temp = 0;
        // Bubble sort algorithm
        for(int i = 0; i < arr.length; i++) {
            for(int j = i + 1; j < arr.length; j++) {
                if(arr[j] < arr[i]) {
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }
        // Printing sorted array
        for(int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");  // Print array elements in one line
        }
    }
}
```

Selection sort:

Code:

```
import java.util.Arrays;
class Main {
    public static void main(String[] args) {
        int arr[] = {4, 2, 1, 3, 5};
        // Selection Sort algorithm
        for (int i = 0; i < arr.length - 1; i++) {
            // Find the index of the minimum element in the unsorted portion
            int minIndex = i;
            for (int j = i + 1; j < arr.length; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;  // Update minIndex
                }
            }
            // Swap the found minimum element with the first unsorted element
            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
        // Printing the sorted array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");  // Print array elements in one line
        }
    }
}
```

Insertion sort:

Code:

```
import java.util.Arrays;

class Main {
    public static void main(String[] args) {
        int arr[] = {4, 2, 1, 3, 5};
        // Insertion Sort algorithm
        for (int i = 1; i < arr.length; i++) {
            int key = arr[i];  // The current element to be inserted
            int j = i - 1;
            // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];  // Shift element to the right
                j = j - 1;
            }
            arr[j + 1] = key;  // Insert the key into its correct position
        }
        // Printing the sorted array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");  // Print array elements in one line
        }
    }
}
```