Java DSA Sorting Techniques - Complete Guide

Created for: College Freshers and Programming Beginners
Focus: Understanding sorting algorithms from basics to implementation

Table of Contents

1. Introduction to Sorting
2. Bubble Sort
3. Selection Sort
4. Insertion Sort
5. Merge Sort
6. Quick Sort
7. Heap Sort
8. Counting Sort
9. Comparison and When to Use What

1. Introduction to Sorting

What is Sorting?

Sorting means arranging data in a particular order - either ascending (small to large) or descending (large to small).

Real-world Example:
Imagine Suresh works at Indian Railways ticket counter in Pune. He has a list of passenger names that arrived randomly. To process them efficiently, he needs to sort them alphabetically. That's sorting!

Why Do We Need Sorting?

- **Searching becomes faster**: Finding a name in a sorted phone directory is much quicker
- **Data analysis**: LIC needs sorted policy numbers to generate reports
- **Better user experience**: Flipkart shows products sorted by price or rating
- **Efficiency**: Many algorithms work better with sorted data

Types of Sorting Algorithms

1. **Comparison-based**: Compare elements to sort (Bubble, Selection, Insertion, Merge, Quick, Heap)
2. **Non-comparison-based**: Use other techniques (Counting, Radix, Bucket)

2. Bubble Sort

The Concept

Bubble Sort works like bubbles rising in water. Larger elements "bubble up" to the end of the array by repeatedly comparing adjacent elements.

Real-world Analogy:
Think of Ramesh organizing his cricket cards by player ratings. He compares two adjacent cards at a time and swaps if the left one has a higher rating than the right one. He repeats this process until all cards are sorted.

How It Works

1. Compare first two elements
2. If first > second, swap them
3. Move to next pair and repeat
4. After one complete pass, the largest element reaches the end
5. Repeat for remaining elements

Step-by-Step Example

Let's sort: `[64, 34, 25, 12, 22]`

Pass 1:

- Compare 64 and 34 → Swap → `[34, 64, 25, 12, 22]`
- Compare 64 and 25 → Swap → `[34, 25, 64, 12, 22]`
- Compare 64 and 12 → Swap → `[34, 25, 12, 64, 22]`
- Compare 64 and 22 → Swap → `[34, 25, 12, 22, 64]`

Pass 2:

- Compare 34 and 25 → Swap → `[25, 34, 12, 22, 64]`
- Compare 34 and 12 → Swap → `[25, 12, 34, 22, 64]`
- Compare 34 and 22 → Swap → `[25, 12, 22, 34, 64]`

Continue until sorted: `[12, 22, 25, 34, 64]`

Visual Representation

```
Initial Array: [64, 34, 25, 12, 22]

Pass 1: (Bubble largest to end)
[64, 34, 25, 12, 22]  → Compare 64 & 34 → Swap
[34, 64, 25, 12, 22]  → Compare 64 & 25 → Swap
[34, 25, 64, 12, 22]  → Compare 64 & 12 → Swap
[34, 25, 12, 64, 22]  → Compare 64 & 22 → Swap
[34, 25, 12, 22, 64]  ✓ 64 is in correct position

Pass 2: (Bubble second largest to end-1)
[34, 25, 12, 22 | 64]  → Compare 34 & 25 → Swap
[25, 34, 12, 22 | 64]  → Compare 34 & 12 → Swap
```

```
[25, 12, 34, 22 | 64]  → Compare 34 & 22 → Swap
[25, 12, 22, 34 | 64]  ✓ 34 is in correct position

Pass 3:
[25, 12, 22 | 34, 64]  → Compare 25 & 12 → Swap
[12, 25, 22 | 34, 64]  → Compare 25 & 22 → Swap
[12, 22, 25 | 34, 64]  ✓ 25 is in correct position

Pass 4:
[12, 22 | 25, 34, 64]  → Compare 12 & 22 → No swap needed
[12, 22 | 25, 34, 64]  ✓ Array is sorted

Final: [12, 22, 25, 34, 64] ✓
```

Complete Dry Run Table

Array: [64, 34, 25, 12, 22]

Pass	i	j	Comparison	arr[j] > arr[j+1]	Action	Array After Step
1	0	0	64 vs 34	Yes	Swap	[34, 64, 25, 12, 22]
1	0	1	64 vs 25	Yes	Swap	[34, 25, 64, 12, 22]
1	0	2	64 vs 12	Yes	Swap	[34, 25, 12, 64, 22]
1	0	3	64 vs 22	Yes	Swap	[34, 25, 12, 22, 64]
2	1	0	34 vs 25	Yes	Swap	[25, 34, 12, 22, 64]
2	1	1	34 vs 12	Yes	Swap	[25, 12, 34, 22, 64]
2	1	2	34 vs 22	Yes	Swap	[25, 12, 22, 34, 64]
3	2	0	25 vs 12	Yes	Swap	[12, 25, 22, 34, 64]
3	2	1	25 vs 22	Yes	Swap	[12, 22, 25, 34, 64]
4	3	0	12 vs 22	No	No Swap	[12, 22, 25, 34, 64]

Total Comparisons: 10
Total Swaps: 9

Java Implementation

```java
public class BubbleSort {

    // Basic Bubble Sort
    public static void bubbleSort(int[] arr) {
        int n = arr.length;

        // Outer loop for passes
        for (int i = 0; i < n - 1; i++) {
```

```
            // Inner loop for comparisons
            for (int j = 0; j < n - i - 1; j++) {
                // Compare adjacent elements
                if (arr[j] > arr[j + 1]) {
                    // Swap if in wrong order
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }

    // Optimized Bubble Sort (stops if array is already sorted)
    public static void optimizedBubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

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

            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no swaps happened, array is sorted
            if (!swapped) {
                break;
            }
        }
    }

    // Helper method to print array
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    // Main method for testing
    public static void main(String[] args) {
        // Example: Sorting ages of students in Chennai college
        int[] studentAges = {22, 19, 21, 18, 20};

        System.out.println("Original array:");
        printArray(studentAges);

        bubbleSort(studentAges);
```

```
        System.out.println("Sorted array:");
        printArray(studentAges);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n) - when array is already sorted (optimized version)
- **Average Case**: O(n²)
- **Worst Case**: O(n²)
- **Space Complexity**: O(1) - sorting happens in place

When to Use Bubble Sort

- Small datasets (less than 50 elements)
- Teaching purposes (easy to understand)
- When simplicity is more important than efficiency
- Nearly sorted arrays (with optimization)

Real-world Use Case:
Mahesh's grocery store in Pune has only 20 items. He uses bubble sort logic mentally to arrange items by price on the shelf.

---

3. Selection Sort

The Concept

Selection Sort works by repeatedly finding the minimum element and placing it at the beginning.

Real-world Analogy:
Dinesh is organizing exam papers by roll numbers. He scans through all papers, finds the lowest roll number, puts it first. Then scans remaining papers, finds the next lowest, puts it second, and so on.

How It Works

1. Find the minimum element in unsorted portion
2. Swap it with the first element of unsorted portion
3. Move boundary of sorted and unsorted portions
4. Repeat until entire array is sorted

Step-by-Step Example

Let's sort: `[64, 25, 12, 22, 11]`

Pass 1: Find minimum (11), swap with first element

- `[11, 25, 12, 22, 64]`

Pass 2: Find minimum in remaining (12), swap with second element

- `[11, 12, 25, 22, 64]`

Pass 3: Find minimum in remaining (22), swap with third element

- `[11, 12, 22, 25, 64]`

Pass 4: Find minimum in remaining (25), already in place

- `[11, 12, 22, 25, 64]`

Visual Representation

```
Initial Array: [64, 25, 12, 22, 11]

Pass 1: Find minimum from entire array
[64, 25, 12, 22, 11]
 ↓   ↓   ↓   ↓   ↓   → Scan all, find min = 11 at index 4
[11, 25, 12, 22, 64]  → Swap 64 with 11
[✓ | 25, 12, 22, 64]  ✓ Position 0 sorted

Pass 2: Find minimum from index 1 onwards
[11 | 25, 12, 22, 64]
      ↓   ↓   ↓   ↓   → Scan remaining, find min = 12 at index 2
[11, 12, 25, 22, 64]  → Swap 25 with 12
[✓   ✓ | 25, 22, 64]  ✓ Position 1 sorted

Pass 3: Find minimum from index 2 onwards
[11, 12 | 25, 22, 64]
          ↓   ↓   ↓   → Scan remaining, find min = 22 at index 3
[11, 12, 22, 25, 64]  → Swap 25 with 22
[✓   ✓   ✓ | 25, 64]  ✓ Position 2 sorted

Pass 4: Find minimum from index 3 onwards
[11, 12, 22 | 25, 64]
              ↓   ↓   → Scan remaining, find min = 25 (already at index 3)
[11, 12, 22, 25, 64]  → No swap needed
[✓   ✓   ✓   ✓ | 64]  ✓ Position 3 sorted

Final: [11, 12, 22, 25, 64] ✓
```

Complete Dry Run Table

Array: [64, 25, 12, 22, 11]

Pass	i	Unsorted Portion	Find Min	Min Index	Min Value	Swap Elements	Array After Pass
1	0	[64, 25, 12, 22, 11]	Scan all	4	11	64 ↔ 11	[11, 25, 12, 22, 64]

Pass	i	Unsorted Portion	Find Min	Min Index	Min Value	Swap Elements	Array After Pass
2	1	[25, 12, 22, 64]	Scan from index 1	2	12	25 ↔ 12	[11, 12, 25, 22, 64]
3	2	[25, 22, 64]	Scan from index 2	3	22	25 ↔ 22	[11, 12, 22, 25, 64]
4	3	[25, 64]	Scan from index 3	3	25	No swap	[11, 12, 22, 25, 64]

Total Comparisons: 10
Total Swaps: 3 (Notice: Much fewer swaps than Bubble Sort!)

Java Implementation

```java
public class SelectionSort {

    public static void selectionSort(int[] arr) {
        int n = arr.length;

        // Outer loop to track sorted boundary
        for (int i = 0; i < n - 1; i++) {
            // Assume first element of unsorted part is minimum
            int minIndex = i;

            // Find the actual minimum in unsorted part
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap minimum element with first element of unsorted part
            if (minIndex != i) {
                int temp = arr[i];
                arr[i] = arr[minIndex];
                arr[minIndex] = temp;
            }
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example: Sorting LIC policy premiums
```

```
        int[] premiums = {5000, 3000, 7000, 2000, 4500};

        System.out.println("Original premiums:");
        printArray(premiums);

        selectionSort(premiums);

        System.out.println("Sorted premiums:");
        printArray(premiums);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n²)
- **Average Case**: O(n²)
- **Worst Case**: O(n²)
- **Space Complexity**: O(1)

When to Use Selection Sort

- Small datasets
- When memory write operations are costly (only n swaps)
- Simple implementation needed
- When auxiliary space is limited

Real-world Use Case:
Mukesh at Flipkart warehouse has to move 10 heavy boxes to the front in order of priority. He doesn't want to move boxes multiple times, so he uses selection sort approach - finds the highest priority box and moves it once to the front.

---

4. Insertion Sort

The Concept

Insertion Sort works like arranging playing cards in your hand. You pick one card at a time and insert it in the correct position among the already sorted cards.

Real-world Analogy:
Kamlesh is playing cards and picks up cards one by one from the deck. As he picks each card, he inserts it in the right position in his hand to keep cards sorted.

How It Works

1. Start with second element (first is already sorted)
2. Compare it with elements in sorted portion
3. Shift elements to right to make space
4. Insert element in correct position
5. Repeat for all elements

Step-by-Step Example

Let's sort: `[12, 11, 13, 5, 6]`

Initial: `[12] | 11, 13, 5, 6` (12 is sorted)

Step 1: Insert 11

- Compare 11 with 12, shift 12 right
- `[11, 12] | 13, 5, 6`

Step 2: Insert 13

- Compare 13 with 12, no shift needed
- `[11, 12, 13] | 5, 6`

Step 3: Insert 5

- Compare with 13, 12, 11 - shift all right
- `[5, 11, 12, 13] | 6`

Step 4: Insert 6

- Compare with 13, 12, 11 - shift these right
- `[5, 6, 11, 12, 13]`

Visual Representation

```
Initial Array: [12, 11, 13, 5, 6]

Step 0: [12] | 11, 13, 5, 6  ← 12 is already "sorted" (single element)

Step 1: Insert 11 into sorted portion [12]
[12] | 11, 13, 5, 6
 ↓
[12, 12] | 13, 5, 6  ← Shift 12 to right
[11, 12] | 13, 5, 6  ← Insert 11 at beginning

Step 2: Insert 13 into sorted portion [11, 12]
[11, 12] | 13, 5, 6
     ↓
[11, 12, 13] | 5, 6  ← 13 is already larger, no shift needed

Step 3: Insert 5 into sorted portion [11, 12, 13]
[11, 12, 13] | 5, 6
 ↓   ↓   ↓
[_, 11, 12, 13] | 6  ← Shift 13 to right
[_, _, 11, 12] | 6   ← Shift 12 to right
[_, _, _, 11] | 6    ← Shift 11 to right
[5, 11, 12, 13] | 6  ← Insert 5 at beginning

Step 4: Insert 6 into sorted portion [5, 11, 12, 13]
[5, 11, 12, 13] | 6
```

```
    ↓   ↓   ↓
[5, _, 11, 12]       ← Shift 13 to right
[5, _, _, 11]        ← Shift 12 to right
[5, _, _, _]         ← Shift 11 to right
[5, 6, 11, 12, 13]   ← Insert 6 at position 1

Final: [5, 6, 11, 12, 13] ✓
```

Complete Dry Run Table

Array: [12, 11, 13, 5, 6]

Step	i	key	j (start)	Compare	Shift/Action	Array State
1	1	11	0	11 < 12	Shift 12 right	[12, 12, 13, 5, 6]
1	1	11	-1	Stop	Insert 11	[11, 12, 13, 5, 6]
2	2	13	1	13 > 12	No shift	[11, 12, 13, 5, 6]
3	3	5	2	5 < 13	Shift 13 right	[11, 12, 13, 13, 6]
3	3	5	1	5 < 12	Shift 12 right	[11, 12, 12, 13, 6]
3	3	5	0	5 < 11	Shift 11 right	[11, 11, 12, 13, 6]
3	3	5	-1	Stop	Insert 5	[5, 11, 12, 13, 6]
4	4	6	3	6 < 13	Shift 13 right	[5, 11, 12, 13, 13]
4	4	6	2	6 < 12	Shift 12 right	[5, 11, 12, 12, 13]
4	4	6	1	6 < 11	Shift 11 right	[5, 11, 11, 12, 13]
4	4	6	0	6 > 5	Stop	-
4	4	6	0	-	Insert 6	[5, 6, 11, 12, 13]

Total Comparisons: 7
Total Shifts: 6

Java Implementation

```java
public class InsertionSort {

    public static void insertionSort(int[] arr) {
        int n = arr.length;

        // Start from second element
        for (int i = 1; i < n; i++) {
            // Element to be inserted in sorted portion
            int key = arr[i];
            int j = i - 1;
```

```
            // Move elements greater than key one position ahead
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            // Insert key at correct position
            arr[j + 1] = key;
        }
    }

    // Recursive version
    public static void recursiveInsertionSort(int[] arr, int n) {
        // Base case
        if (n <= 1) {
            return;
        }

        // Sort first n-1 elements
        recursiveInsertionSort(arr, n - 1);

        // Insert last element in sorted array
        int key = arr[n - 1];
        int j = n - 2;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example: Sorting train platform numbers at Chennai Central
        int[] platforms = {7, 2, 9, 1, 5, 3};

        System.out.println("Original platform numbers:");
        printArray(platforms);

        insertionSort(platforms);

        System.out.println("Sorted platform numbers:");
        printArray(platforms);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n) - when array is already sorted
- **Average Case**: O(n²)
- **Worst Case**: O(n²)
- **Space Complexity**: O(1)

When to Use Insertion Sort

- Small datasets
- Nearly sorted arrays (performs excellently)
- Online sorting (elements arrive one at a time)
- Stable sorting required

Real-world Use Case:
Nitesh maintains a sorted list of customer complaints by timestamp at his Pune customer service center. As new complaints arrive one by one, he uses insertion sort logic to add them to the right position.

---

5. Merge Sort

The Concept

Merge Sort follows the "Divide and Conquer" approach. It divides the array into smaller parts, sorts them, and then merges them back together.

Real-world Analogy:
Hitesh needs to sort 1000 exam papers by roll numbers. He divides them among 10 assistants, each sorts their portion, then they merge the sorted portions together.

How It Works

1. **Divide**: Split array into two halves
2. **Conquer**: Recursively sort both halves
3. **Combine**: Merge the two sorted halves

Step-by-Step Example

Let's sort: `[38, 27, 43, 3, 9, 82, 10]`

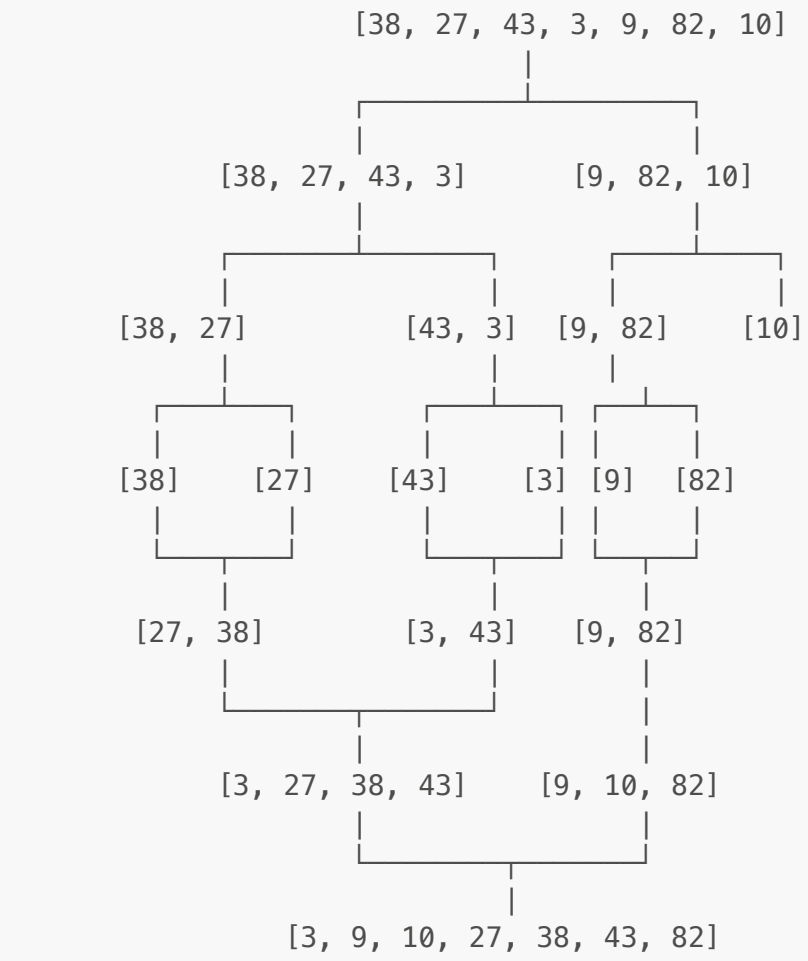
Divide Phase:

```
[38, 27, 43, 3, 9, 82, 10]
          ↓
[38, 27, 43, 3] | [9, 82, 10]
          ↓                ↓
[38, 27] | [43, 3]    [9, 82] | [10]
    ↓         ↓          ↓         ↓
[38] [27] [43] [3]    [9] [82]   [10]
```

Conquer Phase (Merging back):

```
[27, 38] | [3, 43]   [9, 82] | [10]
        ↓                   ↓
[3, 27, 38, 43]     [9, 10, 82]
            ↓
[3, 9, 10, 27, 38, 43, 82]
```

Visual Representation - Detailed




```
Legend:
│ ─  = Divide phase (going down)
└ ┬  = Merge phase (going up)
```

Complete Dry Run - Merge Process

Merging [27, 38] and [3, 43]:

```
Left array:   [27, 38]     i = 0
Right array:  [3, 43]      j = 0
Result:       [_, _, _, _]  k = 0
```

```
Step 1: Compare 27 and 3
  3 < 27 → result[0] = 3, j++
  Result: [3, _, _, _]

Step 2: Compare 27 and 43
  27 < 43 → result[1] = 27, i++
  Result: [3, 27, _, _]

Step 3: Compare 38 and 43
  38 < 43 → result[2] = 38, i++
  Result: [3, 27, 38, _]

Step 4: Left array exhausted
  Copy remaining from right: result[3] = 43
  Result: [3, 27, 38, 43] ✓
```

Merging [9, 82] and [10]:

```
Left array:   [9, 82]      i = 0
Right array:  [10]         j = 0
Result:       [_, _, _]    k = 0

Step 1: Compare 9 and 10
  9 < 10 → result[0] = 9, i++
  Result: [9, _, _]

Step 2: Compare 82 and 10
  10 < 82 → result[1] = 10, j++
  Result: [9, 10, _]

Step 3: Right array exhausted
  Copy remaining from left: result[2] = 82
  Result: [9, 10, 82] ✓
```

Final Merge [3, 27, 38, 43] and [9, 10, 82]:

```
Left array:   [3, 27, 38, 43]     i = 0
Right array:  [9, 10, 82]         j = 0
Result:       [_, _, _, _, _, _, _]  k = 0

Step 1: 3 < 9   → result[0] = 3,  i=1  → [3, _, _, _, _, _, _]
Step 2: 27 > 9  → result[1] = 9,  j=1  → [3, 9, _, _, _, _, _]
Step 3: 27 > 10 → result[2] = 10, j=2  → [3, 9, 10, _, _, _, _]
Step 4: 27 < 82 → result[3] = 27, i=2  → [3, 9, 10, 27, _, _, _]
Step 5: 38 < 82 → result[4] = 38, i=3  → [3, 9, 10, 27, 38, _, _]
Step 6: 43 < 82 → result[5] = 43, i=4  → [3, 9, 10, 27, 38, 43, _]
Step 7: Left exhausted, copy 82      → [3, 9, 10, 27, 38, 43, 82] ✓
```

Detailed Dry Run Table

Recursion Level	Array Segment	Action	Left Part	Right Part	Merged Result
Divide-1	[38,27,43,3,9,82,10]	Split	[38,27,43,3]	[9,82,10]	-
Divide-2	[38,27,43,3]	Split	[38,27]	[43,3]	-
Divide-3	[38,27]	Split	[38]	[27]	-
Merge-3	[38,27]	Merge	[38]	[27]	[27,38]
Divide-3	[43,3]	Split	[43]	[3]	-
Merge-3	[43,3]	Merge	[43]	[3]	[3,43]
Merge-2	[38,27,43,3]	Merge	[27,38]	[3,43]	[3,27,38,43]
Divide-2	[9,82,10]	Split	[9,82]	[10]	-
Divide-3	[9,82]	Split	[9]	[82]	-
Merge-3	[9,82]	Merge	[9]	[82]	[9,82]
Merge-2	[9,82,10]	Merge	[9,82]	[10]	[9,10,82]
Merge-1	[38,27,43,3,9,82,10]	Merge	[3,27,38,43]	[9,10,82]	[3,9,10,27,38,43,82]

Total Merge Operations: 6
Total Comparisons during Merging: 12

Java Implementation

```java
public class MergeSort {

    // Main merge sort function
    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            // Find middle point
            int mid = left + (right - left) / 2;

            // Sort first half
            mergeSort(arr, left, mid);

            // Sort second half
            mergeSort(arr, mid + 1, right);

            // Merge both halves
            merge(arr, left, mid, right);
        }
    }

    // Merge two sorted subarrays
    public static void merge(int[] arr, int left, int mid, int right) {
        // Find sizes of two subarrays
```

```
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temporary arrays
        int[] leftArr = new int[n1];
        int[] rightArr = new int[n2];

        // Copy data to temporary arrays
        for (int i = 0; i < n1; i++) {
            leftArr[i] = arr[left + i];
        }
        for (int j = 0; j < n2; j++) {
            rightArr[j] = arr[mid + 1 + j];
        }

        // Merge temporary arrays back
        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {
            if (leftArr[i] <= rightArr[j]) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of leftArr if any
        while (i < n1) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }

        // Copy remaining elements of rightArr if any
        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example: Sorting Flipkart order IDs
        int[] orderIds = {1045, 1012, 1089, 1003, 1067, 1034};
```

```
        System.out.println("Original order IDs:");
        printArray(orderIds);

        mergeSort(orderIds, 0, orderIds.length - 1);

        System.out.println("Sorted order IDs:");
        printArray(orderIds);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n log n)
- **Average Case**: O(n log n)
- **Worst Case**: O(n log n)
- **Space Complexity**: O(n) - needs temporary arrays

When to Use Merge Sort

- Large datasets
- When consistent O(n log n) performance is needed
- When stability is required
- Linked list sorting
- External sorting (data on disk)

Real-world Use Case:
Indian Railways uses merge sort approach to consolidate passenger lists from multiple booking sources (website, app, counters) because it guarantees consistent performance even with millions of bookings.

---

6. Quick Sort

The Concept

Quick Sort picks a pivot element and partitions the array around it - smaller elements to the left, larger to the right. Then it recursively sorts the partitions.

Real-world Analogy:
Ratnesh is organizing books in Pune library. He picks a book (pivot), puts all books with lower page numbers to the left shelf and higher to the right shelf. Then repeats this process for each shelf.

How It Works

1. Choose a pivot element (often last element)
2. Partition array around pivot
3. Recursively sort left partition
4. Recursively sort right partition

Partitioning Logic

All elements smaller than pivot move to left side, larger move to right side.

Step-by-Step Example

Let's sort: `[10, 7, 8, 9, 1, 5]` (pivot = 5)

Initial: `[10, 7, 8, 9, 1, 5]`

Partition:

- Compare 10 with 5 (pivot) → larger, stays right
- Compare 7 with 5 → larger, stays right
- Compare 8 with 5 → larger, stays right
- Compare 9 with 5 → larger, stays right
- Compare 1 with 5 → smaller, move to left
- Result after partition: `[1, 5, 8, 9, 7, 10]`

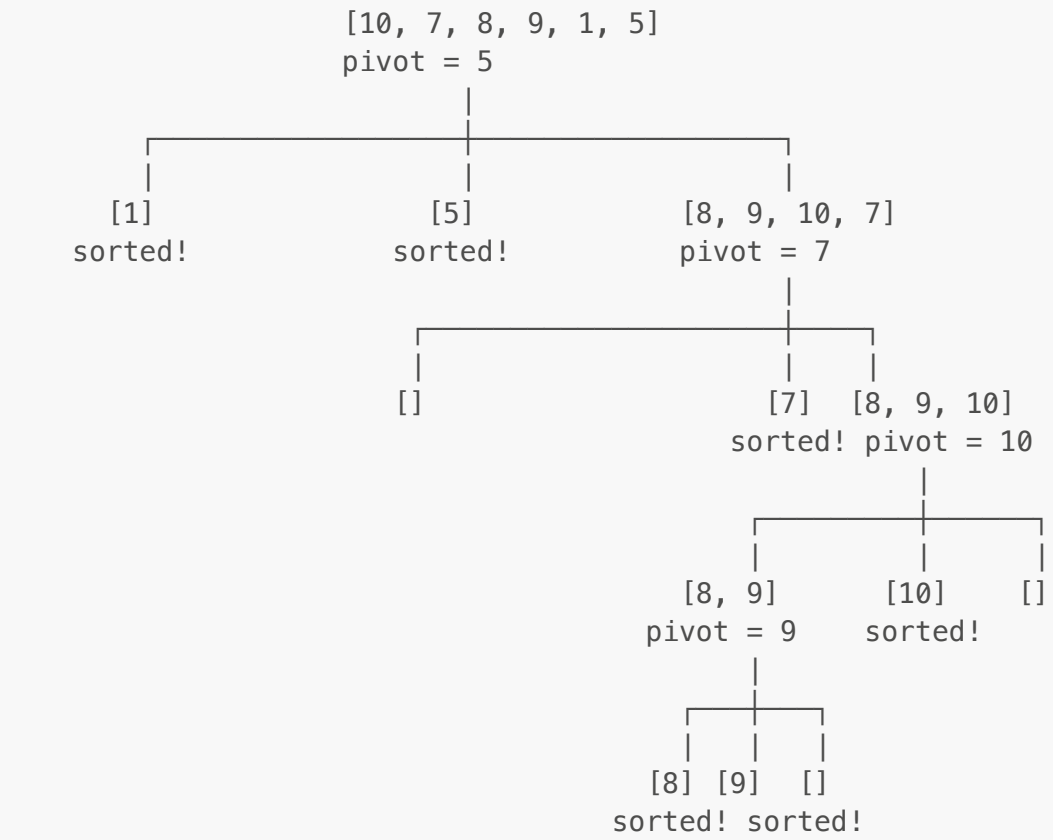
Recursively sort: `[1]` and `[8, 9, 7, 10]`

Continue until sorted: `[1, 5, 7, 8, 9, 10]`

Visual Representation - Partition Process

```
Initial Array: [10, 7, 8, 9, 1, 5]  (pivot = 5, last element)

Partition Process:
                        i
[10, 7, 8, 9, 1, 5]    i = -1 (points to position before smaller elements)
 ↑                      j = 0
 j

Compare 10 with pivot 5: 10 > 5 → Don't swap, j++
Compare 7 with pivot 5:  7 > 5 → Don't swap, j++
Compare 8 with pivot 5:  8 > 5 → Don't swap, j++
Compare 9 with pivot 5:  9 > 5 → Don't swap, j++
Compare 1 with pivot 5:  1 < 5 → i++, swap arr[i] with arr[j]

[1, 7, 8, 9, 10, 5]    i = 0, j = 4
 ↑              ↑
 i              j

Now place pivot in correct position: swap arr[i+1] with pivot
[1, 5, 8, 9, 10, 7]
    ↑
  pivot at correct position

After partition: [1] | 5 | [8, 9, 10, 7]
                Left  Pivot  Right
```

Complete Visual Tree




```
Final Sorted Array: [1, 5, 7, 8, 9, 10]
```

Detailed Partition Dry Run

Array: [10, 7, 8, 9, 1, 5], Pivot = 5

Step	j	arr[j]	Compare	i	Action	Array State
0	-	-	-	-1	Initialize	[10, 7, 8, 9, 1, 5]
1	0	10	10 > 5	-1	No swap	[10, 7, 8, 9, 1, 5]
2	1	7	7 > 5	-1	No swap	[10, 7, 8, 9, 1, 5]
3	2	8	8 > 5	-1	No swap	[10, 7, 8, 9, 1, 5]
4	3	9	9 > 5	-1	No swap	[10, 7, 8, 9, 1, 5]
5	4	1	1 < 5	0	i++, swap(0,4)	[1, 7, 8, 9, 10, 5]
6	5	-	End loop	0	Swap pivot to i+1	[1, 5, 8, 9, 10, 7]

Partition Index: 1 (pivot 5 is now at index 1)

Complete Quick Sort Dry Run

Initial: [10, 7, 8, 9, 1, 5]

Call #	Subarray	Low	High	Pivot	After Partition	Pivot Position

Call #	Subarray	Low	High	Pivot	After Partition	Pivot Position
1	[10, 7, 8, 9, 1, 5]	0	5	5	[1, 5, 8, 9, 10, 7]	1
2	[1]	0	0	-	[1] (single element)	-
3	[8, 9, 10, 7]	2	5	7	[7, 9, 10, 8]	2
4	[]	2	1	-	(empty subarray)	-
5	[9, 10, 8]	3	5	8	[8, 10, 9]	3
6	[]	3	2	-	(empty subarray)	-
7	[10, 9]	4	5	9	[9, 10]	4
8	[]	4	3	-	(empty subarray)	-
9	[10]	5	5	-	[10] (single element)	-

Final Result: [1, 5, 7, 8, 9, 10]

Total Partitions: 4
Total Comparisons: ~15
Total Swaps: 4

Java Implementation

```java
public class QuickSort {

    // Main quick sort function
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            // Find partition index
            int pi = partition(arr, low, high);

            // Sort elements before and after partition
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    // Partition function
    public static int partition(int[] arr, int low, int high) {
        // Choose rightmost element as pivot
        int pivot = arr[high];

        // Index of smaller element
        int i = low - 1;

        for (int j = low; j < high; j++) {
            // If current element is smaller than pivot
            if (arr[j] < pivot) {
                i++;
```

```
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Swap arr[i+1] and pivot
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    // Alternative: Random pivot selection for better performance
    public static int randomPartition(int[] arr, int low, int high) {
        // Generate random index between low and high
        int random = low + (int)(Math.random() * (high - low + 1));

        // Swap random element with last element
        int temp = arr[random];
        arr[random] = arr[high];
        arr[high] = temp;

        return partition(arr, low, high);
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example: Sorting LIC agent commission amounts
        int[] commissions = {4500, 2300, 6700, 1200, 5400, 3100};

        System.out.println("Original commissions:");
        printArray(commissions);

        quickSort(commissions, 0, commissions.length - 1);

        System.out.println("Sorted commissions:");
        printArray(commissions);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n log n)

- **Average Case**: O(n log n)
- **Worst Case**: O(n²) - when array is already sorted
- **Space Complexity**: O(log n) - recursion stack

When to Use Quick Sort

- Large datasets
- When average-case performance matters most
- In-place sorting needed (less memory)
- General-purpose sorting

Real-world Use Case:
Himesh at Chennai tech company uses quick sort in their database system because it's fast on average and doesn't need extra memory like merge sort.

---

7. Heap Sort

The Concept

Heap Sort uses a binary heap data structure. It first builds a max heap, then repeatedly extracts the maximum element and rebuilds the heap.

Real-world Analogy:
Gukesh manages a priority queue at Mumbai hospital emergency. He always treats the most critical patient first (max element), then reorganizes remaining patients by priority.

What is a Heap?

A heap is a complete binary tree where:

- **Max Heap**: Parent node is always greater than children
- **Min Heap**: Parent node is always smaller than children

How It Works

1. Build a max heap from input array
2. Swap root (maximum) with last element
3. Reduce heap size by 1
4. Heapify the root
5. Repeat steps 2-4 until heap size is 1

Array Representation of Heap

For element at index `i`:

- Left child: `2*i + 1`
- Right child: `2*i + 2`
- Parent: `(i-1)/2`

Visual Representation - Heap Building

```
Initial Array: [4, 10, 3, 5, 1]

Array as Tree Structure:
       4(0)
      /    \
    10(1)  3(2)
    /  \
   5(3) 1(4)

Indices in parentheses

Step 1: Build Max Heap (heapify from bottom-up, starting from last non-
leaf)
Last non-leaf node = (n/2 - 1) = (5/2 - 1) = 1

Heapify at index 1 (node 10):
       4
      /  \
    10    3
    / \
   5   1

Check: 10 > 5 and 10 > 1 → Already a valid heap at this node

Heapify at index 0 (node 4):
       4
      /  \
    10    3
    / \
   5   1

Check: 10 > 4 → Swap 4 and 10
      10
      /  \
     4    3
    / \
   5   1

Check node 4 again: 5 > 4 → Swap 4 and 5
      10
      /  \
     5    3
    / \
   4   1

Max Heap Built! Array: [10, 5, 3, 4, 1]
```

Heap Sort Process Visualization

```
Step 1: Max Heap [10, 5, 3, 4, 1]
       10
      /  \
     5    3
    / \
   4   1

Swap root with last element: [1, 5, 3, 4, 10]
Reduce heap size by 1

Heapify root:
       5
      /  \
     4    3
    /
   1       [10]

Step 2: [5, 4, 3, 1 | 10]
Swap root with last: [1, 4, 3, 5, 10]
Heapify:
       4
      /  \
     1    3     [5, 10]

Step 3: [4, 1, 3 | 5, 10]
Swap root with last: [3, 1, 4, 5, 10]
Heapify:
       3
      /
     1         [4, 5, 10]

Step 4: [3, 1 | 4, 5, 10]
Swap root with last: [1, 3, 4, 5, 10]
Heapify:
     1          [3, 4, 5, 10]

Step 5: [1 | 3, 4, 5, 10]
Single element, done!

Final Sorted Array: [1, 3, 4, 5, 10]
```

Complete Dry Run Table

Initial Array: [4, 10, 3, 5, 1]

Phase 1: Build Max Heap

Step	Node Index	Value	Left Child	Right Child	Action	Array After
1	1	10	5	1	10 > 5 and 10 > 1, OK	[4, 10, 3, 5, 1]
2	0	4	10	3	10 > 4, swap	[10, 4, 3, 5, 1]

Step	Node Index	Value	Left Child	Right Child	Action	Array After
3	1	4	5	1	5 > 4, swap	[10, 5, 3, 4, 1]

Max Heap Built: [10, 5, 3, 4, 1]

Phase 2: Extract Max and Sort

Iteration	Heap Size	Root	Last Element	Swap	Array After Swap	Heapify	Final Array
1	5	10	1	10↔1	[1, 5, 3, 4, 10]	[5, 4, 3, 1]	[5, 4, 3, 1, 10]
2	4	5	1	5↔1	[1, 4, 3, 5, 10]	[4, 1, 3]	[4, 1, 3, 5, 10]
3	3	4	3	4↔3	[3, 1, 4, 5, 10]	[3, 1]	[3, 1, 4, 5, 10]
4	2	3	1	3↔1	[1, 3, 4, 5, 10]	[1]	[1, 3, 4, 5, 10]
5	1	1	-	Done	[1, 3, 4, 5, 10]	-	[1, 3, 4, 5, 10]

Total Heapify Calls: 7
Total Comparisons: ~15
Total Swaps: 5

Heapify Process Detail

Example: Heapify at index 0 with array [1, 5, 3, 4]

```
Current: i=0, value=1
Left child: 2*0+1 = 1, value=5
Right child: 2*0+2 = 2, value=3

Step 1: Find largest among root, left, right
  - largest = 0 (value 1)
  - left (5) > root (1), so largest = 1
  - right (3) < largest (5), so largest stays 1

Step 2: Swap if largest is not root
  - largest (1) != root (0)
  - Swap arr[0] with arr[1]: [5, 1, 3, 4]

Step 3: Recursively heapify affected subtree
  - Heapify at index 1
  - Left child: 2*1+1 = 3, value=4
  - Right child: 2*1+2 = 4 (out of bounds)
```

```
  - 4 > 1, so swap: [5, 4, 3, 1]
  - index 3 is leaf, stop

Result: [5, 4, 3, 1] ✓ Valid max heap
```

Java Implementation

```java
public class HeapSort {

    // Main heap sort function
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // Extract elements from heap one by one
        for (int i = n - 1; i > 0; i--) {
            // Move current root to end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            // Heapify reduced heap
            heapify(arr, i, 0);
        }
    }

    // Heapify a subtree rooted at index i
    public static void heapify(int[] arr, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;

        // If left child is larger than root
        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }

        // If right child is larger than largest so far
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }

        // If largest is not root
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
```

```
            // Recursively heapify the affected subtree
            heapify(arr, n, largest);
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example: Sorting product ratings on Flipkart
        int[] ratings = {4, 2, 5, 1, 3};

        System.out.println("Original ratings:");
        printArray(ratings);

        heapSort(ratings);

        System.out.println("Sorted ratings:");
        printArray(ratings);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n log n)
- **Average Case**: O(n log n)
- **Worst Case**: O(n log n)
- **Space Complexity**: O(1)

When to Use Heap Sort

- When consistent O(n log n) is needed
- Memory is limited (in-place sorting)
- Priority queue implementations
- Finding kth largest/smallest elements

Real-world Use Case:
Jitesh's Bangalore traffic control system uses heap sort to manage traffic signal priorities based on vehicle density at different junctions.

---

8. Counting Sort

The Concept

Counting Sort is a non-comparison based sorting. It counts occurrences of each element and calculates positions.

Real-world Analogy:
Ramesh counts votes in Pune municipal election. Instead of comparing candidate names, he counts how many votes each candidate got and declares results based on counts.

How It Works

1. Find the range of input elements (min to max)
2. Create count array to store frequency
3. Modify count array to store actual positions
4. Build output array using counts
5. Copy output back to original array

Step-by-Step Example

Let's sort: `[4, 2, 2, 8, 3, 3, 1]`

Step 1: Count frequencies

- Count array: `[0, 1, 2, 2, 1, 0, 0, 0, 1]`
- Index represents number, value represents frequency

Step 2: Cumulative count (positions)

- Count array: `[0, 1, 3, 5, 6, 6, 6, 6, 7]`

Step 3: Build output array using counts

- Result: `[1, 2, 2, 3, 3, 4, 8]`

Visual Representation

```
Original Array: [4, 2, 2, 8, 3, 3, 1]

Step 1: Count Frequency of Each Element
Index:  0  1  2  3  4  5  6  7  8
Count: [0, 1, 2, 2, 1, 0, 0, 0, 1]
        ↑  ↑  ↑  ↑  ↑           ↑
       no  1  2  2  1          one
      zeros one two twos one    eight

Element Count Visualization:
1 → occurs 1 time   → ●
2 → occurs 2 times  → ●●
3 → occurs 2 times  → ●●
4 → occurs 1 time   → ●
8 → occurs 1 time   → ●

Step 2: Cumulative Count (Position Calculation)
Index:  0  1  2  3  4  5  6  7  8
```

```
Count: [0, 1, 3, 5, 6, 6, 6, 6, 7]
          ↑  ↑  ↑  ↑           ↑
         pos pos pos pos       pos
         for for for for       for
          1   2   3   4         8

This tells us:
- Elements ≤ 1 end at position 1
- Elements ≤ 2 end at position 3
- Elements ≤ 3 end at position 5
- Elements ≤ 4 end at position 6
- Elements ≤ 8 end at position 7

Step 3: Place Elements in Sorted Order (Right to Left)
Process [4, 2, 2, 8, 3, 3, 1] from right to left:

Element 1: count[1]=1, place at position 0
Output: [1, _, _, _, _, _, _]
count[1]=0

Element 3: count[3]=5, place at position 4
Output: [1, _, _, _, 3, _, _]
count[3]=4

Element 3: count[3]=4, place at position 3
Output: [1, _, _, 3, 3, _, _]
count[3]=3

Element 8: count[8]=7, place at position 6
Output: [1, _, _, 3, 3, _, 8]
count[8]=6

Element 2: count[2]=3, place at position 2
Output: [1, _, 2, 3, 3, _, 8]
count[2]=2

Element 2: count[2]=2, place at position 1
Output: [1, 2, 2, 3, 3, _, 8]
count[2]=1

Element 4: count[4]=6, place at position 5
Output: [1, 2, 2, 3, 3, 4, 8]
count[4]=5

Final Sorted Array: [1, 2, 2, 3, 3, 4, 8] ✓
```

Complete Dry Run Table

Array: [4, 2, 2, 8, 3, 3, 1]

Phase 1: Count Frequency

Element	Frequency	Count Array (after this element)
1	1	[0,1,0,0,0,0,0,0,0]
2	2	[0,1,2,0,0,0,0,0,0]
3	2	[0,1,2,2,0,0,0,0,0]
4	1	[0,1,2,2,1,0,0,0,0]
8	1	[0,1,2,2,1,0,0,0,1]

Phase 2: Cumulative Count

Index	Original Count	Cumulative Count	Meaning
0	0	0	No elements ≤ 0
1	1	1	1 element ≤ 1
2	2	3	3 elements ≤ 2
3	2	5	5 elements ≤ 3
4	1	6	6 elements ≤ 4
5	0	6	6 elements ≤ 5
6	0	6	6 elements ≤ 6
7	0	6	6 elements ≤ 7
8	1	7	7 elements ≤ 8

Phase 3: Build Sorted Array (Processing Right to Left)

Step	Element	Count[element]	Position	Output Array	Updated Count
1	1	1	0	[1,,,,,,]	count[1]=0
2	3	5	4	[1,,,,*3*,,_]	count[3]=4
3	3	4	3	[1,,,3,3,,]	count[3]=3
4	8	7	6	[1,,,3,3,_,8]	count[8]=6
5	2	3	2	[1,,*2*,*3*,*3*,,8]	count[2]=2
6	2	2	1	[1,2,2,3,3,_,8]	count[2]=1
7	4	6	5	[1,2,2,3,3,4,8]	count[4]=5

Total Time Complexity: O(n + k) where n=7, k=9
Space Used: O(n + k) = O(16)

Why Process Right to Left?

Processing right to left maintains **stability** - if two elements have the same value, their relative order is preserved. This is important when sorting complex objects where multiple fields matter.

Example showing stability:

```
Input: [2a, 3, 2b, 1]  (subscripts show original order)

If processed left to right:  [1, 2b, 2a, 3]  ✗ Order changed!
If processed right to left:  [1, 2a, 2b, 3]  ✓ Order preserved!
```

Java Implementation

```java
public class CountingSort {

    // Main counting sort function
    public static void countingSort(int[] arr) {
        int n = arr.length;

        // Find maximum element
        int max = arr[0];
        for (int i = 1; i < n; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }

        // Create count array
        int[] count = new int[max + 1];

        // Store count of each element
        for (int i = 0; i < n; i++) {
            count[arr[i]]++;
        }

        // Modify count array to store actual positions
        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
        }

        // Build output array
        int[] output = new int[n];
        for (int i = n - 1; i >= 0; i--) {
            output[count[arr[i]] - 1] = arr[i];
            count[arr[i]]--;
        }

        // Copy output to original array
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
```

```
    }

    // Simplified version for small range
    public static void simpleCountingSort(int[] arr) {
        int n = arr.length;
        int max = arr[0];

        for (int i = 1; i < n; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }

        int[] count = new int[max + 1];

        // Count occurrences
        for (int i = 0; i < n; i++) {
            count[arr[i]]++;
        }

        // Rebuild array
        int index = 0;
        for (int i = 0; i <= max; i++) {
            while (count[i] > 0) {
                arr[index++] = i;
                count[i]--;
            }
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        // Example: Sorting student marks (0-100 range)
        int[] marks = {78, 92, 67, 85, 92, 78, 95, 67};

        System.out.println("Original marks:");
        printArray(marks);

        countingSort(marks);

        System.out.println("Sorted marks:");
        printArray(marks);
    }
}
```

Time and Space Complexity

- **Best Case**: O(n + k) where k is range
- **Average Case**: O(n + k)
- **Worst Case**: O(n + k)
- **Space Complexity**: O(n + k)

When to Use Counting Sort

- When range of input is small (k is close to n)
- Elements are integers in a known range
- Need linear time complexity
- Stability is required

Limitations:

- Not suitable when range is very large
- Only works with non-negative integers (can be modified)
- Requires extra space

Real-world Use Case:
Suresh at Chennai exam center uses counting sort to grade answer sheets marked out of 100. Since the range (0-100) is small and fixed, counting sort is perfect and extremely fast.

---

9. Comparison and When to Use What

Quick Comparison Table

Algorithm	Best Case	Average Case	Worst Case	Space	Stable	Use Case
Bubble Sort	O(n)	O(n²)	O(n²)	O(1)	Yes	Small/nearly sorted
Selection Sort	O(n²)	O(n²)	O(n²)	O(1)	No	Memory writes costly
Insertion Sort	O(n)	O(n²)	O(n²)	O(1)	Yes	Small/nearly sorted
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)	Yes	Large datasets
Quick Sort	O(n log n)	O(n log n)	O(n²)	O(log n)	No	General purpose
Heap Sort	O(n log n)	O(n log n)	O(n log n)	O(1)	No	Consistent performance
Counting Sort	O(n+k)	O(n+k)	O(n+k)	O(n+k)	Yes	Small integer range

Stability in Sorting

A sorting algorithm is **stable** if two elements with equal values appear in the same order in sorted output as they appear in input.

Example:
Input: `[(5, "Suresh"), (3, "Ramesh"), (5, "Mahesh")]`

Stable sort preserves: `[(3, "Ramesh"), (5, "Suresh"), (5, "Mahesh")]`
Unstable might give: `[(3, "Ramesh"), (5, "Mahesh"), (5, "Suresh")]`

Decision Tree: Which Algorithm to Choose

```
Start
├─ Is data size < 50?
│  ├─ Yes → Is data nearly sorted?
│  │  ├─ Yes → Use Insertion Sort
│  │  └─ No → Use Selection Sort (if memory writes costly) or Bubble Sort
(for teaching)
│  └─ No → Continue
│
├─ Is data in small integer range (like 0–100)?
│  ├─ Yes → Use Counting Sort
│  └─ No → Continue
│
├─ Do you need stable sorting?
│  ├─ Yes → Use Merge Sort
│  └─ No → Continue
│
├─ Is memory very limited?
│  ├─ Yes → Use Heap Sort or Quick Sort
│  └─ No → Use Merge Sort or Quick Sort
│
└─ General case → Use Quick Sort (fastest on average)
```

Real-world Scenarios

Scenario 1: Chennai Railway Station - Daily Ticket Sorting

- Data size: 50,000 tickets
- Data type: Ticket numbers
- Requirement: Fast sorting
- **Best choice**: Quick Sort (fast, in-place)

Scenario 2: Pune School - Class Rank Calculation

- Data size: 60 students
- Data type: Marks (0-100)
- Requirement: Stable sorting
- **Best choice**: Counting Sort (linear time, stable)

Scenario 3: Bangalore Hospital - Patient Priority

- Data size: Variable
- Requirement: Always pick highest priority
- **Best choice**: Heap Sort (priority queue)

Scenario 4: Mumbai E-commerce - Product Sorting

- Data size: 1 million products
- Requirement: Stable, preserve order for equal prices
- **Best choice**: Merge Sort (stable, guaranteed O(n log n))

Scenario 5: Delhi Startup - User Dashboard

- Data size: 20 recent activities
- Nearly sorted: New activities added at end
- **Best choice**: Insertion Sort (O(n) for nearly sorted)

Happy Coding! Keep Sorting! Keep Learning!