CodInClub
Powered By BridgeLabz

# 1. Bubble Sort - Sort Student Marks

**Problem Statement:**
A school maintains student marks in an array. Implement **Bubble Sort** to sort the student marks in **ascending order**.

**Hint:**

- Traverse through the array multiple times.
- Compare adjacent elements and swap if needed.
- Repeat the process until no swaps are required.

```
public class BubbleSortMarks {
    public static void main(String[] args) {
        int[] marks = {45, 32, 89, 76, 12};
        int n = marks.length;
        for (int i = 0; i < n - 1; i++) {
            boolean swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (marks[j] > marks[j + 1]) {
                    int temp = marks[j];
                    marks[j] = marks[j + 1];
                    marks[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) break;
        }
        for (int i = 0; i < n; i++) System.out.print(marks[i] + " ");
    }
}
```

---

# 2. Insertion Sort - Sort Employee IDs

**Problem Statement:**
A company stores **employee IDs** in an unsorted array. Implement **Insertion Sort** to sort the employee IDs in **ascending order**.

**Hint:**

- Divide the array into sorted and unsorted parts.
- Pick an element from the unsorted part and insert it into its correct position in the sorted part.
- Repeat for all elements.

```
public class InsertionSortEmpID {
    public static void main(String[] args) {
        int[] empIDs = {104, 102, 109, 101, 108};
        int n = empIDs.length;
        for (int i = 1; i < n; i++) {
            int key = empIDs[i], j = i - 1;
            while (j >= 0 && empIDs[j] > key) {
                empIDs[j + 1] = empIDs[j];
                j--;
            }
            empIDs[j + 1] = key;
        }
        for (int i = 0; i < n; i++) System.out.print(empIDs[i] + " ");
    }
}
```

---

# 3. Merge Sort - Sort an Array of Book Prices

**Problem Statement:**
A bookstore maintains a list of book prices in an array. Implement **Merge Sort** to sort the prices **in ascending order**.

**Hint:**

- **Divide** the array into two halves recursively.
- **Sort** both halves individually.

- **Merge** the sorted halves by comparing elements.

```
public class MergeSortBooks {
    public static void mergeSort(int[] arr, int l, int r) {
        if (l >= r) return;
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }

    public static void merge(int[] arr, int l, int m, int r) {
        int n1 = m - l + 1, n2 = r - m;
        int[] L = new int[n1], R = new int[n2];
        for (int i = 0; i < n1; i++) L[i] = arr[l + i];
        for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) arr[k++] = (L[i] <= R[j]) ? L[i++] :
R[j++];
        while (i < n1) arr[k++] = L[i++];
        while (j < n2) arr[k++] = R[j++];
    }

    public static void main(String[] args) {
        int[] prices = {300, 150, 200, 500, 100};
        mergeSort(prices, 0, prices.length - 1);
        for (int i = 0; i < prices.length; i++) System.out.print(prices[i]
+ " ");
    }
}
```

---

# 4. Quick Sort - Sort Product Prices

**Problem Statement:**
An e-commerce company wants to display product prices in **ascending order**. Implement **Quick Sort** to sort the product prices.

**Hint:**

- Pick a **pivot** element (first, last, or random).
- **Partition** the array such that elements smaller than the pivot are on the left and larger ones are on the right.
- Recursively apply Quick Sort on left and right partitions.

```
public class QuickSortProducts {
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int p = partition(arr, low, high);
            quickSort(arr, low, p - 1);
            quickSort(arr, p + 1, high);
        }
    }

    public static int partition(int[] arr, int low, int high) {
        int pivot = arr[high], i = low - 1;
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                int t = arr[i]; arr[i] = arr[j]; arr[j] = t;
            }
        }
        int t = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = t;
        return i + 1;
    }

    public static void main(String[] args) {
        int[] prices = {400, 250, 120, 700, 310};
        quickSort(prices, 0, prices.length - 1);
        for (int i = 0; i < prices.length; i++) System.out.print(prices[i]
+ " ");
    }
}
```

# 5. Selection Sort - Sort Exam Scores

**Problem Statement:**
A university needs to sort students' **exam scores** in ascending order. Implement **Selection Sort** to achieve this.

**Hint:**

- Find the **minimum element** in the array.
- Swap it with the first unsorted element.
- Repeat the process for the remaining elements.

```
public class SelectionSortScores {
    public static void main(String[] args) {
        int[] scores = {77, 65, 89, 55, 92};
        int n = scores.length;
        for (int i = 0; i < n - 1; i++) {
            int min = i;
            for (int j = i + 1; j < n; j++) {
                if (scores[j] < scores[min]) min = j;
            }
            int temp = scores[i];
            scores[i] = scores[min];
            scores[min] = temp;
        }
        for (int i = 0; i < n; i++) System.out.print(scores[i] + " ");
    }
}
```

---

# 6. Heap Sort - Sort Job Applicants by Salary

**Problem Statement:**
A company receives job applications with different **expected salary demands**. Implement **Heap Sort** to sort these salary demands in **ascending order**.

**Hint:**

- Build a **Max Heap** from the array.
- Extract the largest element (root) and place it at the end.
- Reheapify the remaining elements and repeat until sorted.

```
public class HeapSortSalaries {
    public static void heapify(int[] arr, int n, int i) {
        int largest = i, l = 2 * i + 1, r = 2 * i + 2;
        if (l < n && arr[l] > arr[largest]) largest = l;
        if (r < n && arr[r] > arr[largest]) largest = r;
        if (largest != i) {
            int temp = arr[i]; arr[i] = arr[largest]; arr[largest] = temp;
            heapify(arr, n, largest);
        }
    }

    public static void heapSort(int[] arr) {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i);
        for (int i = n - 1; i >= 0; i--) {
            int t = arr[0]; arr[0] = arr[i]; arr[i] = t;
            heapify(arr, i, 0);
        }
    }

    public static void main(String[] args) {
        int[] salaries = {50000, 30000, 70000, 45000, 60000};
        heapSort(salaries);
        for (int i = 0; i < salaries.length; i++)
System.out.print(salaries[i] + " ");
    }
}
```

---

# 7. Counting Sort - Sort Student Ages

**Problem Statement:**
A school collects students' **ages** (ranging from 10 to 18) and wants them sorted. Implement **Counting Sort** for this task.

**Hint:**

- Create a **count array** to store the frequency of each age.
- Compute cumulative frequencies to determine positions.
- Place elements in their correct positions in the output array.

```
public class CountingSortAges {
    public static void countingSort(int[] arr, int maxVal) {
        int[] count = new int[maxVal + 1];
        for (int i = 0; i < arr.length; i++) count[arr[i]]++;
        int index = 0;
        for (int i = 0; i <= maxVal; i++) {
            while (count[i]-- > 0) arr[index++] = i;
        }
    }

    public static void main(String[] args) {
        int[] ages = {14, 17, 10, 12, 15, 18, 11, 13};
        countingSort(ages, 18);
        for (int i = 0; i < ages.length; i++) System.out.print(ages[i] + "
");
    }
}
```