Java Sorting algorithms

==========

1. **Bubble Sort**

✅ Introduction:

Repeatedly compares adjacent elements and swaps them if they're in the wrong order.

🔁 Flow:

- Loop through the array multiple times.
- Swap adjacent elements if needed.
- Stop when no swaps are made.

📊 Complexity:

- Time: Worst & Avg: $O(n^2)$, Best: O(n)
- Space: O(1)
- Stable: ✅

🎯 Use Case:

Small or nearly sorted arrays; educational purposes.

💻 Java Code:

```
void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
```

```
        }
      }
    }
}
```

🔹 2. **Selection Sort**

✅ Introduction:

Selects the smallest element and places it at the beginning.

🔁 Flow:

- Find the minimum element in the unsorted part.
- Swap it with the first unsorted element.
- Repeat for all positions.

📊 Complexity:

- Time: $O(n^2)$
- Space: O(1)
- Stable: ❌

🎯 Use Case:

Simple sorting when stability isn't required.

💻 Java Code:

```
void selectionSort(int[] arr) {
   int n = arr.length;
   for (int i = 0; i < n - 1; i++) {
      int minIdx = i;
      for (int j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIdx]) {
            minIdx = j;
```

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

🔹 3. **Insertion Sort**

✅ Introduction:

Builds the sorted array one element at a time.

🔁 Flow:

- Start from the second element.
- Compare with previous elements.
- Shift larger elements and insert the current one.

📊 Complexity:

- Time: Worst & Avg: $O(n^2)$, Best: $O(n)$
- Space: $O(1)$
- Stable: ✅

🎯 Use Case:

Efficient for small or nearly sorted arrays.

💻 Java Code:

```
void insertionSort(int[] arr) {
   for (int i = 1; i < arr.length; i++) {
      int key = arr[i];
```

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

🔹 4. **Merge Sort**

✅ Introduction:

Divide-and-conquer algorithm that splits and merges arrays.

🔄 Flow:

- Divide array into halves.
- Recursively sort each half.
- Merge sorted halves.

📊 Complexity:

- Time: O(n log n)
- Space: O(n)
- Stable: ✅

🎯 Use Case:

Large datasets, linked lists, stable sorting.

💻 Java Code:

```
void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
```

```
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

void merge(int[] arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];

    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

🔹 5. **Quick Sort**

✅ Introduction:

Uses a pivot to partition the array and sort recursively.

🔄 Flow:

- Choose a pivot.
- Partition array around pivot.
- Recursively sort left and right parts.

📊 Complexity:

- Time: Worst: $O(n^2)$, Avg: O(n log n)
- Space: O(log n)
- Stable: ❌

🎯 Use Case:

Fastest general-purpose sort; large datasets.

💻 Java Code:

```
void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
```

```
            int temp = arr[i]; arr[i] = arr[j]; arr[j] = temp;
        }
    }
    int temp = arr[i + 1]; arr[i + 1] = arr[high]; arr[high] = temp;
    return i + 1;
}
```

🔹 6. **Counting Sort**

✅ Introduction:

Counts occurrences of each value and builds the sorted array.

🔁 Flow:

- Count frequency of each element.
- Compute cumulative count.
- Place elements in sorted order.

📊 Complexity:

- Time: O(n + k)
- Space: O(k)
- Stable: ✅

🎯 Use Case:

Sorting integers in a known, small range.

💻 Java Code:

```
void countingSort(int[] arr) {
    int max = Arrays.stream(arr).max().getAsInt();
    int[] count = new int[max + 1];

    for (int num : arr) count[num]++;
```

```
    int index = 0;
    for (int i = 0; i < count.length; i++) {
        while (count[i]-- > 0) {
            arr[index++] = i;
        }
    }
}
```