Sorting Algorithms in Java

Sorting algorithms are techniques used to arrange elements of a list or array in a specific order (ascending or descending). Java provides multiple ways to implement these algorithms, including built-in libraries and custom implementations.

Common Sorting Algorithms

1. Bubble Sort

- **What**: Repeatedly compares adjacent elements and swaps them if they are in the wrong order.
- **Why**: Simple to implement but inefficient for large datasets.
- **How It Works**: Pass through the array multiple times until it's sorted.
- **Code**:

```
public class BubbleSort {
    public static 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 arr[j] and arr[j+1]
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
            }
        }
    }
}
```

2. Selection Sort

- **What**: Finds the smallest (or largest) element in the unsorted part of the array and moves it to the sorted part.
- **Why**: Reduces the number of swaps compared to Bubble Sort.
- **Code**:

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

3. Insertion Sort

- **What**: Builds the sorted array one element at a time by picking elements and placing them in their correct position.
- **Why**: Works well for small or partially sorted datasets.
- **Code**:

```
public class InsertionSort {
    public static 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

- **What**: Divides the array into halves, sorts each half, and then merges them back together.
- **Why**: Efficient for large datasets due to its divide-and-conquer approach.
- **Code**:

```
public class MergeSort {
    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    private static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
        int[] leftArr = new int[n1];
        int[] rightArr = new int[n2];
        System.arraycopy(arr, left, leftArr, 0, n1);
        System.arraycopy(arr, mid + 1, rightArr, 0, n2);

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

5. Quick Sort

- **What**: Uses a pivot to partition the array and sort each partition recursively.
- **Why**: One of the fastest sorting algorithms for large datasets.
- **Code**:

```
public class QuickSort {
    public static 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);
        }
    }

    private static 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. Built-in Sorting in Java

- Java provides a built-in sorting method using the `Arrays.sort()` or `Collections.sort()` for arrays and lists respectively.
- Example:

```
import java.util.Arrays;

public class BuiltInSort {
    public static void main(String[] args) {
        int[] arr = {5, 2, 9, 1, 5, 6};
        Arrays.sort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```

Key Points to Remember

- **Bubble Sort**: Simple but inefficient, O(n²) for worst-case.
- **Selection Sort**: Fewer swaps, O(n²) time complexity.
- **Insertion Sort**: Good for small or partially sorted datasets, O(n²) worst-case.
- **Merge Sort**: Stable and efficient, O(n log n) complexity.
- **Quick Sort**: Fast but not stable, O(n log n) on average.
- **Built-in Methods**: Use `Arrays.sort()` for arrays and `Collections.sort()` for lists.

Let me know if you want a detailed explanation of any algorithm or need help with its implementation!