Sorting Algorithms – Theory & Java Implementation

Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. After each pass, the largest element moves to the end like a bubble rising to the surface.

Time Complexity: O(n^2)
Space Complexity: O(1)
Stable: Yes

```
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]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Selection Sort

Selection Sort divides the array into a sorted and unsorted part. It repeatedly selects the minimum element from the unsorted part and places it at the beginning.

Time Complexity: O(n^2)
Space Complexity: O(1)
Stable: No

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

Insertion Sort

Insertion Sort works the way we sort playing cards in hand. Each element is placed in its correct position among the previously sorted elements.

Time Complexity: O(n^2) (Best: O(n))
Space Complexity: O(1)
Stable: Yes

```
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;
    }
}
```

Merge Sort

Merge Sort follows the Divide and Conquer approach. The array is divided into halves, sorted recursively, and then merged.

Time Complexity: O(n log n)
Space Complexity: O(n)
Stable: Yes

```
public static 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);
    }
}
```

Quick Sort

Quick Sort also uses Divide and Conquer. It selects a pivot element and partitions the array so that smaller elements are on one side and larger on the other.

Time Complexity: O(n log n) (Worst: $O(n^2)$)
Space Complexity: O(log n)
Stable: No

```
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);
    }
}
```