Sorting Algorithms with Tracing and Explanations

Here are the sorting algorithms with added tracing and explanations to make them more readable and understandable:

Bubble Sort

```
/**
 * Bubble Sort Algorithm
 * Time Complexity: O(n^2) worst case, O(n) best case (when already sorted)
 * Space Complexity: O(1)
 *
 * Algorithm:
 * 1. Compare adjacent elements and swap if they are in wrong order
 * 2. After each pass, the largest element bubbles up to its correct position
 * 3. Repeat until no more swaps are needed
 */
public static void bubbleSort(int[] arr) {
    System.out.println("Starting Bubble Sort...");
    System.out.println("Initial array: " + Arrays.toString(arr));

    for (int i = 1; i < arr.length; i++) {
        boolean swap = false;
        System.out.println("\nPass " + i + ":");

        for (int j = 0; j < arr.length - i; j++) {
            System.out.print("  Comparing elements at " + j + " and " + (j+1) + ": " +
arr[j] + " vs " + arr[j+1]);

            if (arr[j] > arr[j + 1]) {
                // Swap elements without temporary variable
                arr[j] = arr[j + 1] + arr[j] - (arr[j + 1] = arr[j]);
                swap = true;
                System.out.println(" -> Swapped");
            } else {
                System.out.println(" -> No swap");
            }
        }

        System.out.println("Array after pass " + i + ": " + Arrays.toString(arr));

        if (!swap) {
            System.out.println("No swaps in this pass. Array is sorted.");
            break;
        }
    }

    System.out.println("\nFinal sorted array: " + Arrays.toString(arr));
}
```

Selection Sort

```java
/**
 * Selection Sort Algorithm
 * Time Complexity: O(n^2) in all cases
 * Space Complexity: O(1)
 *
 * Algorithm:
 * 1. Find the minimum element in the unsorted portion
 * 2. Swap it with the first element of the unsorted portion
 * 3. Repeat until the entire array is sorted
 */
public static void selectionSort(int[] arr) {
    System.out.println("Starting Selection Sort...");
    System.out.println("Initial array: " + Arrays.toString(arr));

    int n = arr.length;
    for(int i = 0; i < n; i++) {
        System.out.println("\nPass " + (i+1) + ":");
        System.out.println("Looking for minimum in unsorted portion [" + i + "..." + (n-1) +
"]");

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

        System.out.println("Minimum found at index " + curr + " (value = " + arr[curr] +
")");

        if(curr != i) {
            // Swap elements without temporary variable
            arr[i] = arr[curr] + arr[i] - (arr[curr] = arr[i]);
            System.out.println("Swapped with element at index " + i);
        } else {
            System.out.println("Element already in correct position");
        }

        System.out.println("Array after pass " + (i+1) + ": " + Arrays.toString(arr));
    }

    System.out.println("\nFinal sorted array: " + Arrays.toString(arr));
}
```

Insertion Sort

```java
/**
 * Insertion Sort Algorithm
 * Time Complexity: O(n^2) worst case, O(n) best case (when already sorted)
 * Space Complexity: O(1)
 *
 * Algorithm:
 * 1. Start from the second element (key)
 * 2. Compare with previous elements and shift them right if they are greater than key
 * 3. Insert the key in its correct position
 * 4. Repeat for all elements
 */
public static void insertionSort(int[] arr) {
    System.out.println("Starting Insertion Sort...");
    System.out.println("Initial array: " + Arrays.toString(arr));

    for(int i = 1; i < arr.length; i++) {
        System.out.println("\nPass " + i + ":");
        System.out.println("Processing element at index " + i + " (value = " + arr[i] +
")");

        int key = arr[i];
        int j = i-1;

        System.out.println("  Finding correct position for " + key + " in sorted portion
[0..." + (i-1) + "]");

        while(j >= 0 && arr[j] > key) {
            System.out.println("    Shifting element " + arr[j] + " from index " + j + " to
" + (j+1));
            arr[j+1] = arr[j];
            j--;
        }

        arr[j+1] = key;
        System.out.println("  Inserted " + key + " at index " + (j+1));
        System.out.println("Array after pass " + i + ": " + Arrays.toString(arr));
    }

    System.out.println("\nFinal sorted array: " + Arrays.toString(arr));
}
```

Merge Sort

```java
/**
 * Merge Sort Algorithm
 * Time Complexity: O(n log n) in all cases
 * Space Complexity: O(n)
 *
 * Algorithm:
```

```
 * 1. Divide the array into two halves recursively until single elements remain
 * 2. Merge the divided arrays in sorted order
 */
public static void divide(int[] arr, int left, int right) {
    System.out.println("\nDivide: left = " + left + ", right = " + right);
    if(right > left) {
        int mid = left + (right - left)/2;
        System.out.println("  Splitting at mid = " + mid);

        divide(arr, left, mid);
        divide(arr, mid+1, right);

        System.out.println("  Merging from " + left + " to " + right);
        conquer(arr, left, mid, right);
    }
}

public static void conquer(int[] arr, int left, int mid, int right) {
    System.out.println("    Conquer: left = " + left + ", mid = " + mid + ", right = " +
right);
    System.out.println("    Subarrays to merge:");
    System.out.println("      Left: " + arrayToString(arr, left, mid));
    System.out.println("      Right: " + arrayToString(arr, mid+1, right));

    int firstHalf = left;
    int secondHalf = mid+1;
    int[] mergeArray = new int[right - left + 1];
    int mergeArrayIndex = 0;

    while(firstHalf <= mid && secondHalf <= right) {
        if(arr[firstHalf] <= arr[secondHalf]) {
            mergeArray[mergeArrayIndex++] = arr[firstHalf++];
        } else {
            mergeArray[mergeArrayIndex++] = arr[secondHalf++];
        }
    }

    while(firstHalf <= mid) {
        mergeArray[mergeArrayIndex++] = arr[firstHalf++];
    }

    while(secondHalf <= right) {
        mergeArray[mergeArrayIndex++] = arr[secondHalf++];
    }

    for(int i = 0; i < mergeArray.length; i++, left++) {
        arr[left] = mergeArray[i];
    }

    System.out.println("    Merged array: " + arrayToString(arr, left - mergeArray.length,
right));
}
```

```java
// Helper method to print subarrays
private static String arrayToString(int[] arr, int start, int end) {
    StringBuilder sb = new StringBuilder("[");
    for(int i = start; i <= end; i++) {
        sb.append(arr[i]);
        if(i < end) sb.append(", ");
    }
    sb.append("]");
    return sb.toString();
}
```

Quick Sort

```java
/**
 * Quick Sort Algorithm
 * Time Complexity: O(n log n) average case, O(n^2) worst case
 * Space Complexity: O(log n) due to recursion stack
 *
 * Algorithm:
 * 1. Choose a pivot element (here we use the last element)
 * 2. Partition the array such that elements < pivot are on left and elements > pivot are on
right
 * 3. Recursively sort the left and right partitions
 */
public static int partition(int[] arr, int low, int high) {
    System.out.println("\nPartitioning from " + low + " to " + high);
    System.out.println("  Pivot element: " + arr[high]);

    int pivotIndex = low - 1;
    for(int j = low; j <= high - 1; j++) {
        System.out.print("  Comparing element at " + j + " (" + arr[j] + ") with pivot: ");
        if(arr[j] < arr[high]) {
            pivotIndex++;
            System.out.println("Swap with " + pivotIndex);
            // Swap elements without temporary variable
            arr[pivotIndex] = arr[j] + arr[pivotIndex] - (arr[j] = arr[pivotIndex]);
        } else {
            System.out.println("No swap");
        }
    }

    pivotIndex++;
    // Swap pivot to its correct position
    arr[pivotIndex] = arr[high] + arr[pivotIndex] - (arr[high] = arr[pivotIndex]);

    System.out.println("  Pivot placed at index " + pivotIndex);
    System.out.println("  Array after partitioning: " + arrayToString(arr, low, high));

    return pivotIndex;
```

```java
}

public static void quickSort(int[] arr, int low, int high) {
    System.out.println("\nQuickSort: low = " + low + ", high = " + high);
    if(high > low) {
        int pivotIndex = partition(arr, low, high);
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
```

These implementations now include:

1. Clear algorithm descriptions at the top of each method
2. Detailed tracing output showing each step of the sorting process
3. Helper methods for better visualization
4. Comments explaining the key operations
5. Proper variable naming for better readability
6. Consistent formatting

To use any of these sorting algorithms with tracing, simply call them with your array:

```java
int[] arr = {5, 2, 9, 1, 5, 6};
bubbleSort(arr.clone());  // Using clone() to preserve original array
selectionSort(arr.clone());
insertionSort(arr.clone());

// For recursive sorts:
int[] mergeArr = arr.clone();
System.out.println("\nStarting Merge Sort...");
divide(mergeArr, 0, mergeArr.length - 1);
System.out.println("Final sorted array: " + Arrays.toString(mergeArr));

int[] quickArr = arr.clone();
System.out.println("\nStarting Quick Sort...");
quickSort(quickArr, 0, quickArr.length - 1);
System.out.println("Final sorted array: " + Arrays.toString(quickArr));
```