**-: Merge Sort / Quick Sort :-**

**Git Help**

* [**https://www.coursera.org/learn/developer-data-structures-and-algorithms/supplement/rJiSm/version-control-with-github**](https://www.coursera.org/learn/developer-data-structures-and-algorithms/supplement/rJiSm/version-control-with-github)

**Junit Installation on your Machine:-**

* [**https://www.coursera.org/learn/developer-data-structures-and-algorithms/supplement/Zae2e/setting-up-junit**](https://www.coursera.org/learn/developer-data-structures-and-algorithms/supplement/Zae2e/setting-up-junit)

import java.util.Arrays;

public class MergeSortExample {

    public static void main(String[] args) {

        int[] myArray = {64, 34, 25, 12, 22, 11, 90, 5};

        mergeSort(myArray, 0, myArray.length - 1);

        System.out.println("Sorted array: " + Arrays.toString(myArray));

    }

    public static void mergeSort(int[] arr, int left, int right) {

        if (left < right) {

            int mid = left + (right - left) / 2; // Find the middle index

            // Recursively sort the left and right halves

            mergeSort(arr, left, mid);

            mergeSort(arr, mid + 1, right);

            // Merge the sorted halves

            merge(arr, left, mid, right);

        }

    }

    public static void merge(int[] arr, int left, int mid, int right) {

        // Sizes of two subarrays

        int n1 = mid - left + 1;

        int n2 = right - mid;

        // Create temporary arrays

        int[] leftArray = new int[n1];

        int[] rightArray = new int[n2];

        // Copy data to temp arrays

        System.arraycopy(arr, left, leftArray, 0, n1);

        System.arraycopy(arr, mid + 1, rightArray, 0, n2);

        // Merge the temp arrays back into arr[left...right]

        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {

            if (leftArray[i] <= rightArray[j]) {

                arr[k] = leftArray[i];

                i++;

            } else {

                arr[k] = rightArray[j];

                j++;

            }

            k++;

        }

        // Copy remaining elements of leftArray, if any

        while (i < n1) {

            arr[k] = leftArray[i];

            i++;

            k++;

        }

        // Copy remaining elements of rightArray, if any

        while (j < n2) {

            arr[k] = rightArray[j];

            j++;

            k++;

        }

    }

}

**Example Input:-**

int[] arr = {38, 27, 43, 3, 9, 82, 10};

mergeSort(arr, 0, arr.length - 1);

**Step 1: Recursive Splitting**

**The array is recursively divided into two halves.**

**Recursive Splitting:-**

mergeSort(arr, 0, 6) // Left = 0, Right = 6, Mid = (0+6)/2 = 3

├── mergeSort(arr, 0, 3) // Left half

│ ├── mergeSort(arr, 0, 1) // Left half

│ │ ├── mergeSort(arr, 0, 0) // Single element {38} (STOP)

│ │ ├── mergeSort(arr, 1, 1) // Single element {27} (STOP)

│ │ ├── **merge**(arr, 0, 0, 1) // Merge {38} and {27} → {27, 38}

│ │

│ ├── mergeSort(arr, 2, 3) // Right half

│ │ ├── mergeSort(arr, 2, 2) // Single element {43} (STOP)

│ │ ├── mergeSort(arr, 3, 3) // Single element {3} (STOP)

│ │ ├── **merge**(arr, 2, 2, 3) // Merge {43} and {3} → {3, 43}

│ │

│ ├── **merge**(arr, 0, 1, 3) // Merge {27, 38} and {3, 43} → {3, 27, 38, 43}

│

├── mergeSort(arr, 4, 6) // Right half

│ ├── mergeSort(arr, 4, 5) // Left half

│ │ ├── mergeSort(arr, 4, 4) // Single element {9} (STOP)

│ │ ├── mergeSort(arr, 5, 5) // Single element {82} (STOP)

│ │ ├── **merge**(arr, 4, 4, 5) // Merge {9} and {82} → {9, 82}

│ │

│ ├── mergeSort(arr, 6, 6) // Single element {10} (STOP)

│ ├── **merge**(arr, 4, 5, 6) // Merge {9, 82} and {10} → {9, 10, 82}

│

├── **merge**(arr, 0, 3, 6) // Merge {3, 27, 38, 43} and {9, 10, 82} → {3, 9, 10, 27, 38, 43, 82}

**Step 2: Merging Steps**

**First Merge Calls**

* merge(arr, 0, 0, 1): Merges {38} and {27} → {27, 38}
* merge(arr, 2, 2, 3): Merges {43} and {3} → {3, 43}
* merge(arr, 4, 4, 5): Merges {9} and {82} → {9, 82}

**Second Merge Calls**

* merge(arr, 0, 1, 3): Merges {27, 38} and {3, 43} → {3, 27, 38, 43}
* merge(arr, 4, 5, 6): Merges {9, 82} and {10} → {9, 10, 82}

**Final Merge Call**

* merge(arr, 0, 3, 6): Merges {3, 27, 38, 43} and {9, 10, 82} → {3, 9, 10, 27, 38, 43, 82}

### ****Full Merge Process****

Now, let's see how the merge function works through all recursive calls:

| **Merge Call** | **Left Array** | **Right Array** | **Merged Result** |
| --- | --- | --- | --- |
| merge(arr, 0, 0, 1) | [38] | [27] | [27, 38] |
| merge(arr, 2, 2, 3) | [43] | [3] | [3, 43] |
| merge(arr, 0, 1, 3) | [27, 38] | [3, 43] | [3, 27, 38, 43] |
| merge(arr, 4, 4, 5) | [9] | [82] | [9, 82] |
| merge(arr, 4, 5, 6) | [9, 82] | [10] | [9, 10, 82] |
| merge(arr, 0, 3, 6) | [3, 27, 38, 43] | [9, 10, 82] | [3, 9, 10, 27, 38, 43, 82] |

### ****Example Walkthrough****

Let's take an example where **merge(arr, 0, 1, 3)** is called during merge sort.

#### ****Before Merging****

The array arr = {27, 38, 3, 43, 9, 82, 10}  
At this step, we are merging **[27, 38]** and **[3, 43]**.

#### ****Temporary Arrays****

leftArray = [27, 38] (n1 = 2)

rightArray = [3, 43] (n2 = 2)

i = 0, j = 0, k = 0

### ****Step-by-step Execution of**** while (i < n1 && j < n2)

| **Step** | **i** | **j** | **leftArray[i]** | **rightArray[j]** | **Comparison** | **Resulting arr** | **k** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 27 | 3 | 3 < 27 | {3, \_, \_, \_, \_, \_, \_} | 1 |
| 2 | 0 | 1 | 27 | 43 | 27 < 43 | {3, 27, \_, \_, \_, \_, \_} | 2 |
| 3 | 1 | 1 | 38 | 43 | 38 < 43 | {3, 27, 38, \_, \_, \_, \_} | 3 |

At this point, i = 2 (end of leftArray), so the first loop stops.

### ****Step-by-step Execution of**** while (i < n1)

Since i = 2 (end of leftArray), this **does not run**.

### ****Step-by-step Execution of**** while (j < n2)

| **Step** | **j** | **rightArray[j]** | **Resulting arr** |
| --- | --- | --- | --- |
| 1 | 1 | 43 | {3, 27, 38, 43, \_, \_, \_} |

At this point, **both arrays are merged**.

public class Main {

    public static void main(String[] args) {

        int[] myArray = {64, 34, 25, 12, 22, 11, 90, 5};

        quicksort(myArray, 0, myArray.length - 1);

        System.out.print("Sorted array: ");

        for (int value : myArray) {

            System.out.print(value + " ");

        }

    }

    public static void quicksort(int[] array, int low, int high) {

        if (low < high) {

            int pivotIndex = partition(array, low, high);

            quicksort(array, low, pivotIndex - 1);

            quicksort(array, pivotIndex + 1, high);

        }

    }

    public static int partition(int[] array, int low, int high) {

        int pivot = array[high];

        int i = low - 1;

        for (int j = low; j < high; j++) {

            if (array[j] <= pivot) {

                i++;

                int temp = array[i];

                array[i] = array[j];

                array[j] = temp;

            }

        }

        int temp = array[i + 1];

        array[i + 1] = array[high];

        array[high] = temp;

        return i + 1;

    }

}

**Given Array**

int[] myArray = {64, 34, 25, 12, 22, 11, 90, 5};

quicksort(myArray, 0, myArray.length - 1);

**Step 1: Initial Call**

quicksort(arr, 0, 7)

 **Subarray:** [64, 34, 25, 12, 22, 11, 90, 5]

 **Pivot = 5 (arr[7])**

 **Initial i = -1** (low - 1)

#### ****Partitioning Process****

We loop j from 0 to 6 and swap elements ≤ 5.

| **j** | **arr[j]** | **arr[j] ≤ 5?** | **Action** | **i** | **Updated Array** |
| --- | --- | --- | --- | --- | --- |
| 0 | 64 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |
| 1 | 34 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |
| 2 | 25 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |
| 3 | 12 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |
| 4 | 22 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |
| 5 | 11 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |
| 6 | 90 | ❌ No | No swap | -1 | [64, 34, 25, 12, 22, 11, 90, 5] |

**Final Swap (Placing Pivot 5)**

* Swap arr[i+1] (64) with arr[7] (5)
* **New Array:** [5, 34, 25, 12, 22, 11, 90, 64]
* **Pivot Position = 0**

**Step 2: Recursive Calls**

* quicksort(arr, 0, -1) → **Base Case, ignored**
* quicksort(arr, 1, 7)

**Step 3: Sorting Right Half**

quicksort(arr, 1, 7)

 **Subarray:** [34, 25, 12, 22, 11, 90, 64]

 **Pivot = 64 (arr[7])**

 **Initial i = 0**

#### ****Partitioning Process****

| **j** | **arr[j]** | **arr[j] ≤ 64?** | **Action** | **i** | **Updated Array** |
| --- | --- | --- | --- | --- | --- |
| 1 | 34 | ✅ Yes | i++, Swap arr[i] with arr[j] | 1 | [5, 34, 25, 12, 22, 11, 90, 64] |
| 2 | 25 | ✅ Yes | i++, Swap arr[i] with arr[j] | 2 | [5, 34, 25, 12, 22, 11, 90, 64] |
| 3 | 12 | ✅ Yes | i++, Swap arr[i] with arr[j] | 3 | [5, 34, 25, 12, 22, 11, 90, 64] |
| 4 | 22 | ✅ Yes | i++, Swap arr[i] with arr[j] | 4 | [5, 34, 25, 12, 22, 11, 90, 64] |
| 5 | 11 | ✅ Yes | i++, Swap arr[i] with arr[j] | 5 | [5, 34, 25, 12, 22, 11, 90, 64] |
| 6 | 90 | ❌ No | No swap | 5 | [5, 34, 25, 12, 22, 11, 90, 64] |

**Final Swap (Placing Pivot 64)**

* Swap arr[i+1] (90) with arr[7] (64)
* **New Array:** [5, 34, 25, 12, 22, 11, 64, 90]
* **Pivot Position = 6**

**Step 4: Recursive Calls**

* quicksort(arr, 1, 5) → Sort [34, 25, 12, 22, 11]
* quicksort(arr, 7, 7) → **Base Case, ignored**

**Step 5: Sorting [34, 25, 12, 22, 11]**

**quicksort(arr, 1, 5)**

 **Pivot = 11 (arr[5])**

 **Initial i = 0**

#### ****Partitioning Process****

| **j** | **arr[j]** | **arr[j] ≤ 11?** | **Action** | **i** | **Updated Array** |
| --- | --- | --- | --- | --- | --- |
| 1 | 34 | ❌ No | No swap | 0 | [5, 34, 25, 12, 22, 11, 64, 90] |
| 2 | 25 | ❌ No | No swap | 0 | [5, 34, 25, 12, 22, 11, 64, 90] |
| 3 | 12 | ❌ No | No swap | 0 | [5, 34, 25, 12, 22, 11, 64, 90] |
| 4 | 22 | ❌ No | No swap | 0 | [5, 34, 25, 12, 22, 11, 64, 90] |

**Final Swap (Placing Pivot 11)**

* Swap arr[i+1] (34) with arr[5] (11)
* **New Array:** [5, 11, 25, 12, 22, 34, 64, 90]
* **Pivot Position = 1**

**Step 6: Recursive Calls**

* quicksort(arr, 1, 0) → **Base Case, ignored**
* quicksort(arr, 2, 5) → Sort [25, 12, 22, 34]

## **Final Steps**

Following the same **partitioning process**, we continue sorting [25, 12, 22, 34], which results in:

**Sorted array: [5, 11, 12, 22, 25, 34, 64, 90]**

**Summary of Recursive Calls**

**quicksort(arr, 0, 7)**

**├── quicksort(arr, 0, -1) ❌ Ignored**

**├── quicksort(arr, 1, 7)**

**│ ├── quicksort(arr, 1, 5)**

**│ │ ├── quicksort(arr, 1, 0) ❌ Ignored**

**│ │ ├── quicksort(arr, 2, 5) → Sort [25, 12, 22, 34]**

**│ ├── quicksort(arr, 7, 7) ❌ Ignored**

**Bubble Sort:-**

    public static void bubbleSortDescending(int[] arr) {

        int n = arr.length;

        for (int counter1 = 0; counter1 < n - 1; counter1++) { //pass counter1

            for (int counter2 = 0; counter2 < n - 1 - counter1; counter2++) {

                if (arr[counter2] < arr[counter2 + 1]) { // Change comparison to descending order

                    int temp = arr[counter2];

                    arr[counter2] = arr[counter2 + 1];

                    arr[counter2 + 1] = temp;

                }

            }

        }

    }