Green University of Bangladesh

Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering

Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)

Lab Report NO: 06

Course Title : Algorithm Lab
Course Code : CSE 204
Section : D - 9

Lab Experiment Name : Implement Merge Sort and Quick Sort Algorithm.

Student Details

Name		ID
1.	Jahidul Islam	221002504

Lab Date : 04/12/2023
Submission Date : 08/12/2023
Course Teacher's Name : Md. Abu Rahman Refat

Lab Report Status

Marks: ………………………………… Signature:…………………
Comments:…………………………………… Date:…………………………

1. TITLE:

Implement Merge Sort and Quick Sort Algorithm.

2. OBJECTIVES/AIM:

The primary objectives or aim of this lab experiment are:

- **Understanding Merge and Quick Sort Algorithm:** To comprehend the principles behind the Merge Sort algorithm, a divide-and-conquer sorting technique.

- **Implementation Skills:** To enhance programming skills by implementing the Merge Sort, Quick Sort algorithm in Java.

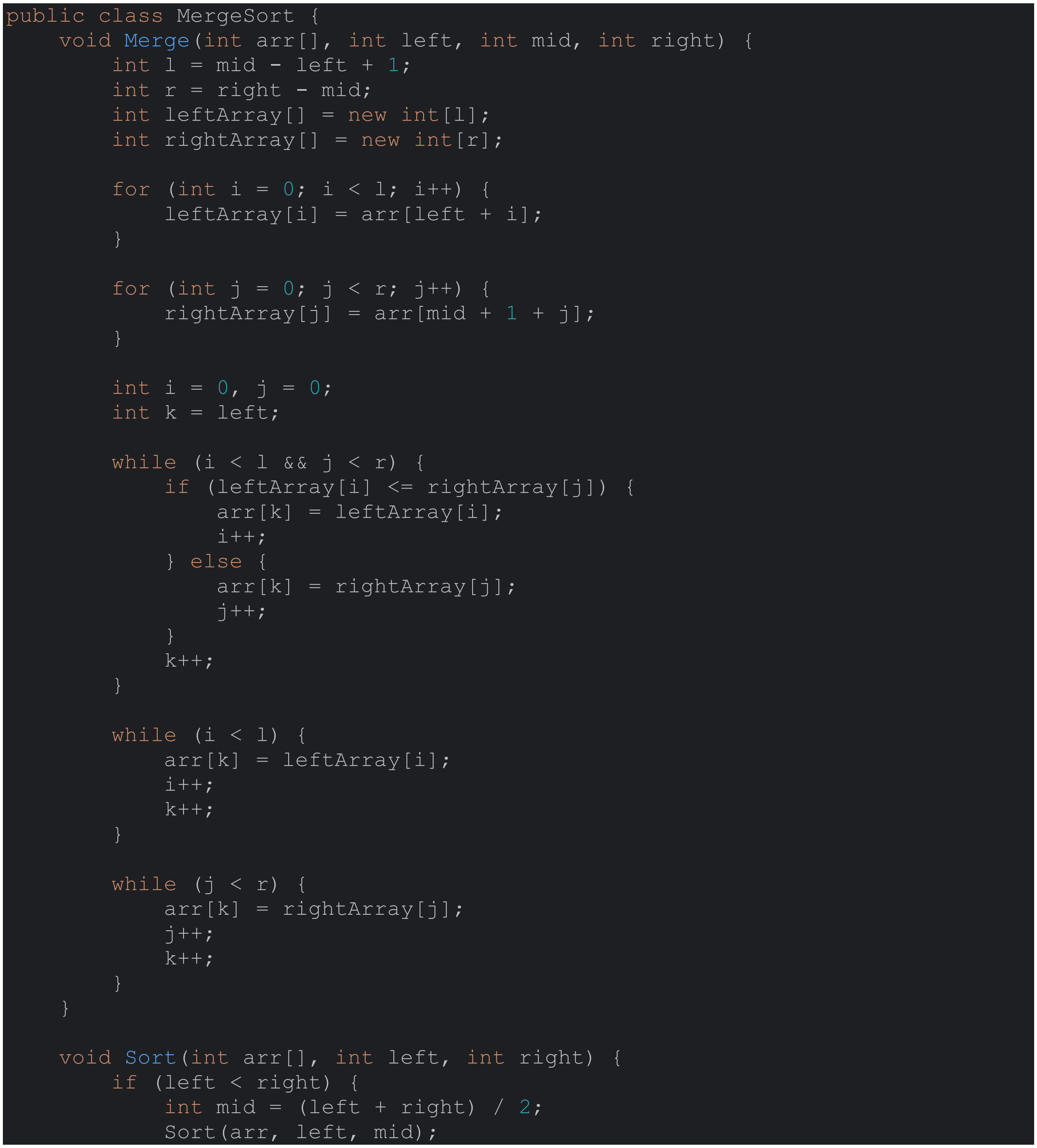
- **Analyzing Time Complexity:** To observe and analyze the time complexity of Merge Sort , Quick Sort concerning different input sizes.

3. PROCEDURE:

- **Understanding Merge Sort, Quick Sort Algorithm:**
  1. Study the theoretical concepts of the Merge Sort**,** Quick Sort algorithm, focusing on its divide-and-conquer strategy.
- **Code Implementation:**
  2. Java program to implement the Merge Sort**,** Quick Sort algorithm based on theoretical understanding.
  3. Verify the correctness of the implementation through step-by-step code walkthrough.
- **Testing and Debugging:**
  4. Test the implementation with various input sizes and cases to ensure the algorithm works correctly.
  5. Debug and address any issues encountered during testing.
- **Performance Analysis:**
  6. Analyze the time complexity of Merge Sort**,** Quick Sort by measuring the execution time for different input sizes.
  7. Record and document the results for further analysis.

4. IMPLEMENTATION:

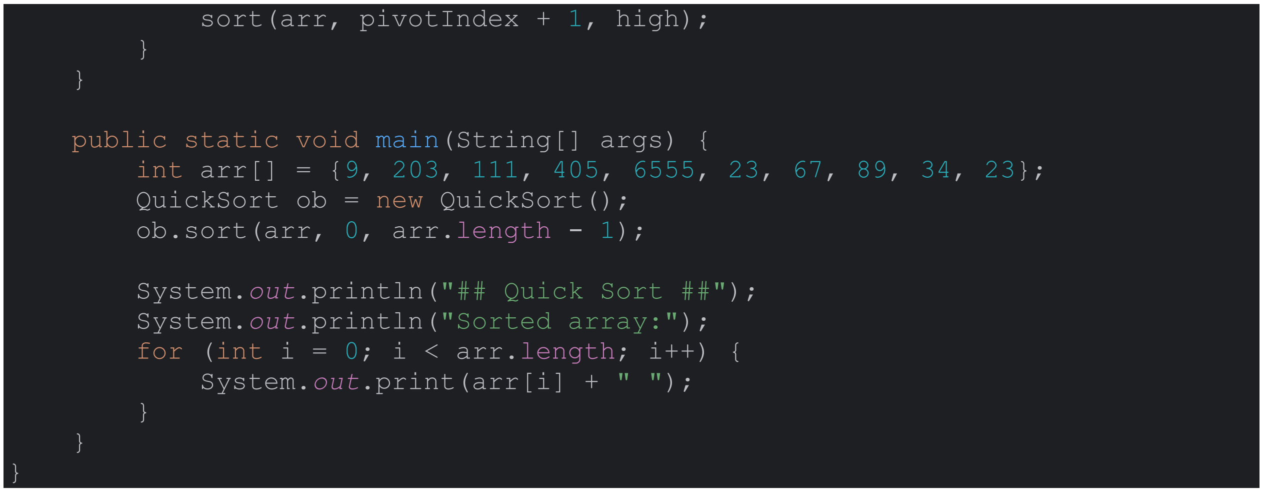
Code in Java: MergeShot



```java
public class MergeSort {
    void Merge(int arr[], int left, int mid, int right) {
        int l = mid - left + 1;
        int r = right - mid;
        int leftArray[] = new int[l];
        int rightArray[] = new int[r];

        for (int i = 0; i < l; i++) {
            leftArray[i] = arr[left + i];
        }

        for (int j = 0; j < r; j++) {
            rightArray[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0;
        int k = left;

        while (i < l && j < r) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k] = leftArray[i];
                i++;
            } else {
                arr[k] = rightArray[j];
                j++;
            }
            k++;
        }

        while (i < l) {
            arr[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < r) {
            arr[k] = rightArray[j];
            j++;
            k++;
        }
    }

    void Sort(int arr[], int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            Sort(arr, left, mid);
            Sort(arr, mid + 1, right);
            Merge(arr, left, mid, right);
        }
    }

    public static void main(String[] args) {
        int arr[] = {90, 23, 101, 45, 65, 23, 67, 89, 34, 23};
        MergeSort ob = new MergeSort();
        ob.Sort(arr, 0, arr.length - 1);

        System.out.println("Sorted array:");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

Code in Java: Quick Sort

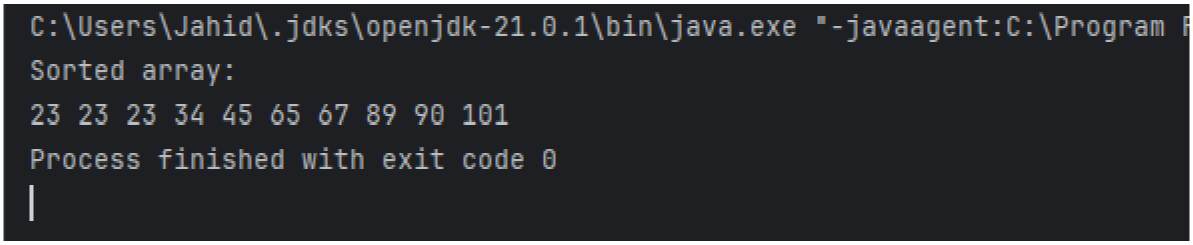
```
public class QuickSort {
    // Function to partition the array and return the pivot index
    int partition(int arr[], int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Swap arr[i+1] and arr[high] (pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }

    // Function to perform Quick Sort
    void sort(int arr[], int low, int high) {
        if (low < high) {
            // Find pivot element such that elements smaller than pivot are
on the left, and larger on the right
            int pivotIndex = partition(arr, low, high);

            // Recursively sort the sub-arrays
            sort(arr, low, pivotIndex - 1);
```



```
            sort(arr, pivotIndex + 1, high);
        }
    }

    public static void main(String[] args) {
        int arr[] = {9, 203, 111, 405, 6555, 23, 67, 89, 34, 23};
        QuickSort ob = new QuickSort();
        ob.sort(arr, 0, arr.length - 1);

        System.out.println("## Quick Sort ##");
        System.out.println("Sorted array:");
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}
```

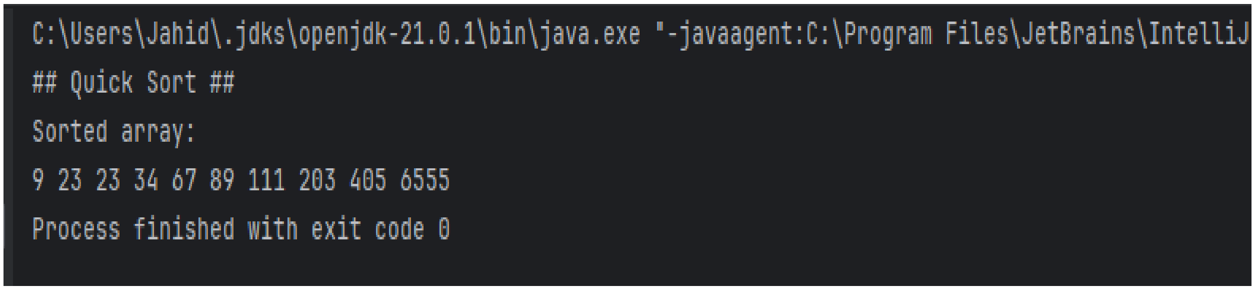
5. TEST RESULT / OUTPUT:

- Objective 1: Merge Sort



```
C:\Users\Jahid\.jdks\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program F
Sorted array:
23 23 23 34 45 65 67 89 90 101
Process finished with exit code 0
```

- Objective 2: Quick Sort



```
C:\Users\Jahid\.jdks\openjdk-21.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ
## Quick Sort ##
Sorted array:
9 23 23 34 67 89 111 203 405 6555
Process finished with exit code 0
```

6. DISCUSSION AND ANALYSIS:

Objective 1: Merge SOrt

a) **Algorithm Complexity:**
   1. Discuss the time complexity of the Merge Sort algorithm and how it aligns with the theoretical expectations (O(n log n)).
   2. Consider the space complexity and any auxiliary space required during the sorting process.

b) **Correctness and Verification:**
   3. Confirm the correctness of the implementation by comparing the sorted output with the expected results.
   4. Discuss any challenges faced during the debugging process and how they were resolved.

c) **Performance Analysis:**
   5. Present and analyze the performance results, including the execution time for different input sizes.
   6. Discuss any observations or patterns in the performance data.

d) **Advantages and Limitations:**
   7. Discuss the advantages of Merge Sort, such as its stability, predictable performance, and suitability for large datasets.
   8. Address any limitations or scenarios where Merge Sort may not be the most efficient choice.

- Objective 2: Quick Sort

1. **Algorithm Complexity:** Average-case time complexity of O(n log n), worst case: O(n^2)
2. **Partitioning:** The efficiency of Quick Sort heavily depends on the partitioning process. The chosen pivot influences the balance of the sub-arrays.
3. **In-Place Sorting:** The swapping of elements is done in the same array. Quick Sort is an in-place sorting algorithm
4. **Stability:** Quick Sort is not a stable sorting algorithm. order of equal elements may not be preserved during sorting.

7. OverAll Insights:

- **Choosing the Right Algorithm:**
  - The choice between Merge Sort and Quick Sort depends on specific requirements and constraints.
  - Merge Sort's predictability and stability make it suitable for general use, while Quick Sort's efficiency shines in scenarios with large datasets.
- **Pivot Selection in Quick Sort:**
  - The performance of Quick Sort is highly dependent on pivot selection. Strategies such as randomized or median-of-three pivots can mitigate worst-case scenarios

8. CONCLUSION:

For scenarios where stability, predictability, and consistent performance across various input sizes are crucial, Merge Sort stands as a reliable choice. In cases where efficiency is paramount, and memory constraints are significant, Quick Sort, with careful pivot selection, offers a powerful solution. Ultimately, the choice between the two algorithms should be made based on the specific requirements of the task at hand.