Practical 2: Sorting Arrays-II

Aim

Implement the **Merge sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

Merge Sort is one of the most popular sorting algorithms that is based on the principle of **Divide and Conquer** Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

A similar approach is used by merge sort where we divide an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

Algorithm

- Start
- Declare array and left, right, mid variable
- Perform merge function.

```
if left > right
        return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

- Stop

Program

```
import java.util.*;

class MergeSort {

    void merge(int arr[], int p, int q, int r) {

        int n1 = q - p + 1;

        int n2 = r - q;

        int L[] = new int[n1];

        int R[] = new int[n2];

        for (int i = 0; i < n1; ++i){

            L[i] = arr[p + i];

    }

        for (int j = 0; j < n2; ++j){

            R[j] = arr[q + 1 + j];

    }
```

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

    void sort(int arr[], int l, int r) {
        if (l < r) {
            int m = l + (r - l) / 2;

            sort(arr, l, m);
            sort(arr, m + 1, r);
            merge(arr, l, m, r);
        }
```

```
        }

        static void printArray(int arr[]) {
                int n = arr.length;
                for (int i = 0; i < n; ++i){
                        System.out.print(arr[i] + " ");
    }
                System.out.println();
        }

        public static void main(String args[]) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter the number of elements: ");
    int length = sc.nextInt();

    int[] array = new int[length];

    for(int k = 0; k < length; k++) {
        System.out.printf("Enter element %d: ", k+1);
        array[k] = sc.nextInt();
    }

                System.out.println("Given Array");
                printArray(array);

                MergeSort ob = new MergeSort();
                ob.sort(array, 0, array.length - 1);

                System.out.println("\nSorted array");
                printArray(array);
        }
}
```

Output

```
Enter the number of elements: 6
Enter element 1: 65
Enter element 2: 45
Enter element 3: 32
Enter element 4: 78
Enter element 5: 6
Enter element 6: 626
Given Array
65 45 32 78 6 626

Sorted array
6 32 45 65 78 626
```

Analysis of Algorithm

A merge sort consists of several passes over the input.

1st pass - merges segments of size 1

2nd pass - merges segments of size 2

ith pass - merges segments of size 2^{i-1}

Thus, the total number of passes is *logn*. As merge showed, we can merge two sorted segments in linear time, which means that each pass takes *O(n)* time. Since there are [$log_2 n$] passes, the total computing time is ***O(nlogn)***.

The time complexity of Merge Sort is *O(nlog(n))* in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Description	**Best Case**	**Average case**	**Worst case**
Complexity	O(nlog(n))	O(nlog(n))	O(nlog(n))

Aim

Implement the **Quick sort** algorithm for an array. Measure the execution time and the number of steps required to execute each algorithm in best case, worst case and average case.

Theory

Quicksort is a sorting algorithm based on the **divide and conquer** approach where an array is divided into subarrays by selecting a pivot element (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
    if (leftmostIndex < rightmostIndex)
        pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
        quickSort(array, leftmostIndex, pivotIndex - 1)
        quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
    set rightmostIndex as pivotIndex
    storeIndex <- leftmostIndex – 1
    for i <- leftmostIndex + 1 to rightmostIndex
    if element[i] < pivotElement
        swap element[i] and element[storeIndex]
        storeIndex++
    swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Program

```
import java.util.*;
class QuickSort {
    static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
```

```
    }
    static int partition(int[] arr, int low, int high) {

        int pivot = arr[high];
        int i = (low - 1);

        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }
    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);
        }
    }
    static void printArray(int[] arr, int size) {
        for (int i = 0; i < size; i++){
            System.out.print(arr[i] + " ");
  }
        System.out.println();
    }

public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.print("Enter the number of elements: ");
    int n = sc.nextInt();
```

```
        int[] array = new int[n];
        for(int k = 0; k < n; k++) {
           System.out.printf("Enter element %d: ", k+1);
           array[k] = sc.nextInt();
        }
                System.out.println("Given Array");
                printArray(array, n);
                quickSort(array, 0, n - 1);
                System.out.println("\nSorted array: ");
                printArray(array, n);
           }
}
```

Output:

```
Enter the number of elements: 5
Enter element 1: 4
Enter element 2: 617
Enter element 3: 7
Enter element 4: 54
Enter element 5: 21
Given Array
4 617 7 54 21

Sorted array:
4 7 21 54 617
```

Analysis of Algorithm

1. **Worst Case Complexity**

   It occurs when the pivot element picked is either the greatest or the smallest element. This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.

2. **Best Case Complexity**

   It occurs when the pivot element is always the middle element or near to the middle element.

3. **Average Case Complexity**

   It occurs when the above conditions do not occur.

Description	**Best Case**	**Average case**	**Worst case**
Complexity	O(nlog(n))	O(nlog(n))	$O(n^2)$