Analysis of Algorithm

Practical no 4 : Merge Sort

Code :

import java.util.Arrays;

public class Mergesort {

    public static void mergeSort(int[] arr) {

        if (arr.length < 2)

            return;

        int mid = arr.length / 2;

        int[] left = Arrays.copyOfRange(arr, 0, mid);

        int[] right = Arrays.copyOfRange(arr, mid, arr.length);

        mergeSort(left);

        mergeSort(right);

        merge(arr, left, right);

    }

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

        int i = 0, j = 0, k = 0;

        while (i < left.length && j < right.length) {

            if (left[i] <= right[j]) {

                arr[k++] = left[i++];

            } else {

                arr[k++] = right[j++];

            }

        }

        while (i < left.length)

            arr[k++] = left[i++];

        while (j < right.length)

            arr[k++] = right[j++];

    }

    public static void main(String[] args) {

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

        System.out.println("Original array: " + Arrays.toString(arr));

        mergeSort(arr);

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

    }

}

Output :

Original array: [38, 27, 43, 3, 9, 82, 10]

Sorted array: [3, 9, 10, 27, 38, 43, 82]

Analysis :

1. Algorithm Overview:

Merge Sort is a divide and conquer algorithm. It works by:

* Dividing the input array into two halves,
* Recursively sorting each half,
* Merging the two sorted halves into a single sorted array.

2. Code Explanation:

mergeSort(int[] arr) Function:

* Base Case: The first if statement checks if the array's length is less than 2. An array with 0 or 1 element is trivially sorted, so the function simply returns without doing anything.
* Splitting the Array: The array is split into two halves:
  + left is the subarray from index 0 to mid - 1.
  + right is the subarray from index mid to arr.length - 1.
* Recursive Sorting: The method recursively calls mergeSort() on both halves (left and right).
* Merging: After sorting both halves, the function calls the merge() function to combine them into a single sorted array.

merge(int[] arr, int[] left, int[] right) Function:

* Merging Sorted Halves: The merge() function merges two sorted arrays left and right into the original array arr by comparing the elements of both arrays.
* Merging Process: Two pointers, i and j, track the positions in left and right, respectively. The smallest element between left[i] and right[j] is placed in arr[k].
  + If left[i] <= right[j], then left[i] is added to arr[k] and i is incremented.
  + If right[j] < left[i], then right[j] is added to arr[k] and j is incremented.
* Handling Leftovers: After comparing all elements, there may be some elements left in either left or right. The remaining elements are added to arr[k].

main(String[] args) Method:

* Initial Array: The main method initializes an array arr with some values.
* Merge Sort Execution: The mergeSort(arr) method is called to sort the array.
* Output: The sorted array is printed after the mergeSort operation.

3. Time Complexity Analysis:

* Divide Step: The array is divided into two halves in each recursive call. This step occurs log(n) times because the array is split in half with each recursion.
* Merge Step: In each level of recursion, every element from the array is processed exactly once when merging the two halves. Merging an array of length n takes O(n) time.

Time Complexity: O(n log n)

4. Space Complexity Analysis:

* Auxiliary Space: The space used by merge sort is mainly due to the extra arrays created for storing the left and right halves of the array during each merge operation.
* At each level of recursion, we create two subarrays: left and right. The total number of elements across all levels of recursion is n. Thus, the space complexity is proportional to the input array size.

Space Complexity: O(n)

5. Best, Worst, and Average Case Scenarios:

* Best Case: Even if the input array is already sorted, the algorithm will still split and merge the array. Therefore, the best-case time complexity is O(n log n).
* Worst Case: In the worst case (array sorted in reverse order), merge sort still divides the array recursively and merges in O(n log n) time.
* Average Case: The average time complexity for merge sort is also O(n log n), regardless of the initial arrangement of the array elements.