REGENT EDUCATION & RESEARCH FOUNDATION

———————————O———————————

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY

Design and Analysis of Algorithm

Name :Soumadeep Kr. Dey

Roll :26371023043

Registration :232630510043

Paper : MCAN-303

Stream :MCA

Year : 2nd

Semester :3rd

Session :2023-25

INDEX

Introduction :

Certainly! Below is an outline for a document on **Merge Sort** that spans five pages, including sections on the introduction, algorithm, working example, analysis, and conclusion.

Merge Sort Algorithm

Introduction

Sorting algorithms are fundamental in computer science and are used in a variety of applications. One of the most efficient comparison-based sorting algorithms is **Merge Sort**. Developed by **John von Neumann** in 1945, Merge Sort uses the divide-and-conquer paradigm to sort an array or list.

Merge Sort is a stable, comparison-based algorithm with a time complexity of **O(n log n)**, which makes it more efficient than algorithms like bubble sort, insertion sort, and selection sort, particularly for large datasets. In this document, we will explore how Merge Sort works, analyze its complexity, and examine a practical example of sorting.

How Merge Sort Works

Merge Sort follows a recursive process where the input array is split into smaller subarrays, sorted individually, and then merged together. The three main steps of Merge Sort are:

1. **Divide:**

   The array is recursively divided into two equal halves until each subarray contains a single element or is empty.

2. **Conquer (Sort):**

   Once the array is divided, the subarrays are sorted recursively using the same method.

3. **Merge:**

   After sorting, the sorted subarrays are merged back into one sorted array.

Pseudocode of Merge Sort

python

MergeSort(arr):

iflen(arr) <= 1:

```
returnarr

mid = len(arr) // 2

left_half = MergeSort(arr[:mid])

right_half = MergeSort(arr[mid:])

return Merge(left_half, right_half)

Merge(left, right):

sorted_array = []

i, j = 0, 0

while i <len(left) and j <len(right):

if left[i] < right[j]:

sorted_array.append(left[i])

        i += 1

else:

sorted_array.append(right[j])

        j += 1

sorted_array.extend(left[i:])

sorted_array.extend(right[j:])

returnsorted_array
```

In the pseudocode above, `MergeSort` recursively splits the array into smaller parts, and `Merge` combines them back into a sorted array.

Working Example

Let’s walk through a detailed example to illustrate how Merge Sort works.

Consider the following array that needs to be sorted:

[38, 27, 43, 3, 9, 82, 10 \]

Step 1: Divide the Array

Initially, the array is divided into two subarrays until each subarray has only one element.

Step 1: [38, 27, 43, 3, 9, 82, 10]

Step 2: [38, 27, 43, 3] and [9, 82, 10]

Step 3: [38, 27] and [43, 3] and [9] and [82, 10]

Step 4: [38] and [27] and [43] and [3] and [9] and [82] and [10]

Step 2: Conquer (Sort Each Subarray)

Each of these subarrays is now merged into a sorted array:

Step 5: [27, 38] and [3, 43] and [9] and [10, 82]

Step 3: Merge the Sorted Arrays

Finally, the sorted subarrays are merged into a single sorted array:

Step 6: [3, 27, 38, 43] and [9, 10, 82]

Step 7: [3, 9, 10, 27, 38, 43, 82]

After following these steps, the array is sorted in ascending order.

Time and Space Complexity

Time Complexity

The time complexity of Merge Sort is **O(n log n)** in all cases—best, average, and worst. This is due to the way the array is split and merged:

- The array is divided in half recursively, creating log(n) levels of recursion.

- At each level, merging takes **O(n)** time since we compare all elements once.

Thus, the overall complexity is **O(n log n)**, making it efficient even for large datasets.

Space Complexity

The space complexity of Merge Sort is **O(n)** because it requires additional space for temporary arrays used in the merging process. Each time the array is split, new temporary subarrays are created to hold the values before they are merged back. Thus, Merge Sort is not an in-place sorting algorithm, unlike algorithms such as Quick Sort.

Advantages and Disadvantages

Advantages

1. **Stable Sort:**

Merge Sort is a stable algorithm, meaning that it preserves the relative order of equal elements.

2. **Consistent Time Complexity:**

Merge Sort consistently runs in O(n log n) time, regardless of the input array's order.

3. **Efficiency for Large Datasets:**

It performs better than quadratic-time algorithms (e.g., Bubble Sort or Selection Sort) when dealing with large datasets.

4. **External Sorting:**

Merge Sort is well-suited for sorting large datasets that don’t fit in memory because it works efficiently with external data (on disk).

Disadvantages

1. **Space Complexity:**

The additional memory requirement makes Merge Sort less efficient in terms of space compared to in-place algorithms like Quick Sort or Heap Sort.

2. **Performance on Small Datasets:**

For smaller datasets, Merge Sort may be slower than simpler algorithms such as Insertion Sort due to the overhead of recursion and merging.

Conclusion

Merge Sort is one of the most powerful sorting algorithms with a time complexity of **O(n log n)**. It uses the divide-and-conquer approach to recursively split and merge arrays. While its additional space requirements can be a disadvantage, Merge Sort’s stability and efficiency make it an excellent choice for sorting large datasets and external files. Moreover, its consistent performance across all types of input makes it a reliable choice when sorting is required.

In comparison with other algorithms, Merge Sort stands out due to its consistent time complexity and ability to handle large datasets, although it may not always be the best choice for smaller datasets due to its space and recursion.