	Bansilal Ramnath Agarwal Charitable Trust's Vishwakarma Institute of Information Technology Department of Artificial Intelligence and Data Science	
Student Name: Mihir Umesh Patwardhan		
Class: TY	Division: C	Roll No: 373040
Semester: 5th	Academic Year: 2023 - 24	
Subject Name & Code: Design and Analysis of Algorithms, ADUA31202		
Title of Assignment: Implementation the following algorithm using Divide & Conquer method. (a)Merge sort (b) Quick Sort Also display execution time for different size of input and perform the analysis.		
Date of Performance: 18/08/2023	Date of Submission: 02/09/2023	

Quick Sort:

Quick Sort is a widely used and efficient sorting algorithm that follows a divide-and-conquer approach. It works by selecting a pivot element from the array and partitioning the other elements into two sub-arrays, one with elements less than the pivot and the other with elements greater than the pivot. This partitioning step is performed recursively on the sub-arrays, effectively sorting them. Quick Sort's key feature is its ability to sort in-place, minimizing the need for extra memory. However, its performance can degrade if the pivot selection isn't balanced, leading to worst-case scenarios.

To address this, various implementations use techniques like choosing a randomized pivot or employing the "three-way partitioning" strategy. Quick Sort's average and best-case time complexity of O (n log n) make it a powerful sorting solution for large datasets.

Merge Sort:

Merge Sort is a well-known sorting algorithm that employs a divide-and-conquer strategy for sorting arrays. It starts by dividing the array into smaller, equal-sized sub-arrays until each sub-array contains only one element. Then, it merges these sub-arrays in a sorted manner, combining them back into larger sub-arrays until the entire array is sorted. The merging step is crucial, as it ensures that the sub-arrays are combined in a way that maintains the overall sorted order.

Merge Sort's primary advantage is its consistent performance, with a guaranteed worst-case time complexity of O (n log n), making it suitable for sorting large datasets. However, it does require additional memory space for the merging process, which can be a drawback in memory-constrained environments. Despite this, Merge Sort's stability and predictable runtime make it a reliable choice for situations where stability and worst-case performance are essential.

Mihir. Patwardhan
C-373040
17/08/23 22111346

Design and Analysis of Algorithm
Assignment 2

* Problem Statement:-
Implementation of the following algorithm using Divide and Conquer method.
a. Merge Sort
b. Quick Sort

* Objective:-
To learn about the two different divide & conquer methods for sorting.

* Algorithm:-
◇ a. Quick Sort
1. Divide an array into subarrays by selecting a pivot element. While dividing the array, the pivot element should be positioned such that all elements less than pivot are to the left side and all elements greater than pivot are to the right side.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point elements are already sorted. Finally elements are combined to form a sorted array.

eg:- 8 7 6 1 0 9 $\boxed{2}$ pivot
↑

1. Compare first element with pivot element
8 > 2, a second pointer is set for 8.

2. Now pivot element is compared with other elements if an element lesser than pivot is encountered then the element is swapped with the greater element found earlier.

8 7 6 1 0 9 $\boxed{2}$ pivot

second pointer

8 7 6 1 0 9 $\boxed{2}$ pivot

second pointer

8 7 6 1 0 9 $\boxed{2}$ pivot

second pointer

1 < 2 ∴ Swap with second pointer.

1 7 6 8 0 9 $\boxed{2}$ pivot

3. Now again we check for second pointer.

1 7 6 8 0 9 $\boxed{2}$ pivot

second pointer

4. We repeat steps 2 & 3 until second ~~pointer~~ last element is reached. At that point we swap the pivot element and the second pointer.

1 7 6 8 0 9 $\boxed{2}$ pivot

second pointer

1 0 6 8 7 9 $\boxed{2}$ pivot

1 0 6 8 7 9 $\boxed{2}$ pivot

second pointer

∴ Second last element is checked so swap pivot & second pointer.

1 0 2 8 7 9 6

5. Divide the subarrays. Choose pivot elements again for the left & right sub-part separately and repeat from step 2 until each subarray is formed of a single element. ~~Then~~ The array is sorted by the end these steps.

1 0 $\boxed{2}$ 8 7 9 $\boxed{6}$ // Dividing

0 1 2 6 7 9 8 // Swapping

0 1 2 6 7 9 8 // looking for sub-array

0 1 2 6 7 9 $\boxed{8}$ // Dividing

0 1 2 6 7 8 9 // Swapping

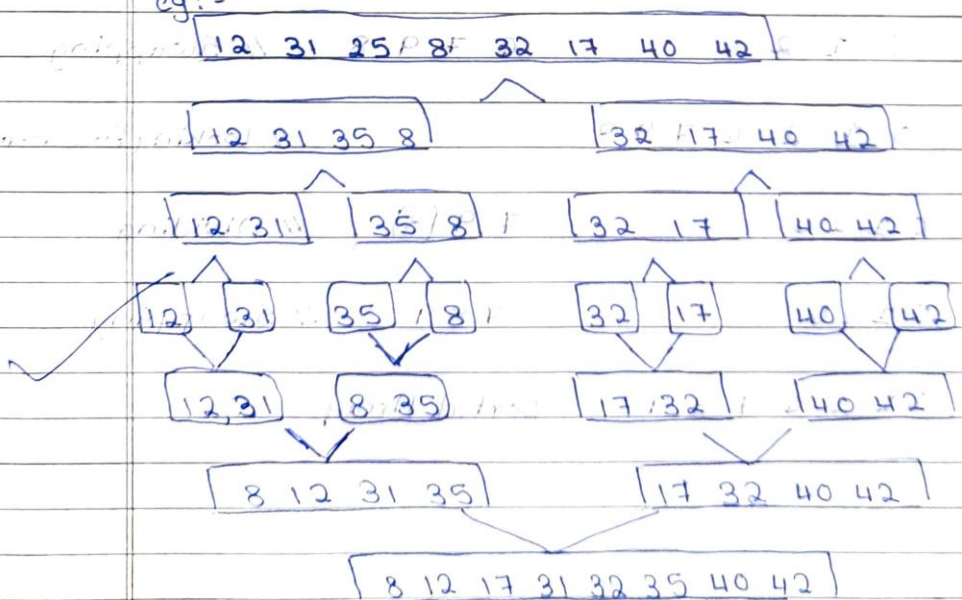
0 1 2 6 7 8 9 Sorted Array

Page No.
Date

b. Merge Sort :-

1. On the given array assign a beg to a[0] and end to a[n]. Find out the mid by taking mean of beg & end.
2. From the mid split the array into 2 parts. Repeat Step 1 until we have subarrays of only one element each.
3. At the end of step 2 we start merging the elements in the order which we split them in. The elements are sorted white merging together.
4. ~~Complete~~ Repeat the step 3 until all elements converge into one single array. By then array will be sorted.

eg:-



* Conclusion:-

We have learned the implementation of 2 Divide & Conquer methods 1. Quick Sort and 2. Merge Sort with the help of python program.

C

BKadam
18/8/2023

QUICK SORT:

```
def sort(array):
    if len(array) <= 1:
        return array

    pivot = array[-1]
    left = []
    middle = []
    right = []

    for num in array:
        if num < pivot:
            left.append(num)
        elif num == pivot:
            middle.append(num)
        else:
            right.append(num)
    #print(left,right,middle)                    Use if we need to see all steps
    return sort(left) + middle + sort(right)
#-------------------------------------------------------------------------------

A = [3, 6, 8, 10, 1, 2, 14, 5, 7, 12, 30]
sorted_A = sort(A)
print("Sorted Array:", sorted_A)

B = [3]
sorted_B = sort(B)
print("Sorted Array:", sorted_B)
```

OUTPUT:

```
Sorted Array: [1, 2, 3, 5, 6, 7, 8, 10, 12, 14, 30]
Sorted Array: [3]
Time taken for execution -0.0056130886
```

MERGE SORT:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    beg = arr[:mid]
    end = arr[mid:]
    left = merge_sort(beg)
    right = merge_sort(end)
    return merge(left, right)

def merge(i, j):
    result = []
    left, right = 0, 0
    while left < len(i) and right < len(j):
        if i[left] < j[right]:
            result.append(i[left])
            left += 1
        else:
            result.append(j[right])
            right += 1
    result.extend(i[left:])
    result.extend(j[right:])
    return result

a = [12,31,25,8,32,17,42,40]
Sorted = merge_sort(a)
print("Original array :",a)
print("Sorted array :",Sorted)
```

OUTPUT:

```
Original array : [12, 31, 25, 8, 32, 17, 42, 40]
Sorted array : [8, 12, 17, 25, 31, 32, 40, 42]
Time taken for execution -0.0035190582
```

Analysis of Program:

1. **Quick Sort**
   a. **Time Complexity**

      Best Case Complexity: O (n*log n)

      Worst Case Complexity: O (n*n)

      Average Case Complexity: O (n*log n)

   b. **Space Complexity**

      The space complexity of quick sort is O (n).

2. **Merge Sort**
   c. **Time Complexity**

      Best Case Complexity: O (n*log n)

      Worst Case Complexity: O (n*log n)

      Average Case Complexity: O (n*log n)

   d. **Space Complexity**

      The space complexity of merge sort is O (n).

We can see that Quick sort has higher worst-case complexity (n*n) than merge sort (n*log n). From the time complexity analysis and the time take for execution checked in the program as well we can say that merge sort is better in terms of time.