EXPERIMENT NO 3

Title: Implement Merge Sort algorithm using Divide and Conquer Approach and analyze its complexity using recursion tree method.

Aim: - To implement Merge Sort Algorithm using Divide and Conquer and find its complexity.

Lab Outcomes : CSL401.1: Ability to implement the algorithm using divide and conquer approach and also analyze the complexity of various sorting algorithms.

Theory:-

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub problems, we state each sub problem as sorting a sub array $A[p \ldots r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurs through sub problems.

To sort $A[p \ldots r]$:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p \ldots r]$ into two subarrays $A[p \ldots q]$ and $A[q + 1 \ldots r]$, each containing about half of the elements of $A[p \ldots r]$. That is, q is the halfway point of $A[p \ldots r]$.

2. Conquer Step

Conquer by recursively sorting the two sub arrays $A[p \ldots q]$ and $A[q + 1 \ldots r]$.

3. Combine Step

Combine the elements back in $A[p \ldots r]$ by merging the two sorted sub arrays $A[p \ldots q]$ and $A[q + 1 \ldots r]$ into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, *and* r).

Note that the recursion bottoms out when the sub array has just one element, so that it is trivially sorted.

Merging

What remains is the MERGE procedure. The following is the input and output of the MERGE procedure.

INPUT: Array A and indices p, q, r such that $p \leq q \leq$ r and sub array A [p ... q] is sorted and sub array A [$q + 1$ ... r] is sorted. By restrictions on p, q, r, neither sub array is empty.

OUTPUT: The two sub arrays are merged into a single sorted sub array in A [p ... r].

We implement it so that it takes Θ (n) time, where $n = r - p + 1$, which is the number of elements being merged.

Idea behind Linear Time Merging

Think of two piles of cards, each pile is sorted and placed face-up on a table with the smallest cards on top. We will merge these into a single sorted pile, face-down on the table.

A basic step:

- Choose the smaller of the two top cards.
- Remove it from its pile, thereby exposing a new top card.
- Place the chosen card face-down onto the output pile.
- Repeatedly perform basic steps until one input pile is empty.
- Once one input pile empties, just take the remaining input pile and place it face-down onto the output pile.

Each basic step should take constant time, since we check just the two top cards. There are at most n basic steps, since each basic step removes one card from the input piles, and we started with n cards in the input piles. Therefore, this procedure should take $\Theta(n)$ time.

Now the question is do we actually need to check whether a pile is empty before each basic step? The answer is no, we do not. Put on the bottom of each input pile a special sentinel card. It contains a special value that we use to simplify the code. We use ∞, since that's guaranteed to lose to any other value. The only way that ∞ cannot lose is when both piles have ∞ exposed as their top cards. But when that happens, all the no sentinel cards have already been placed into the output pile. We know in advance that there are exactly $r - p + 1$ no sentinel cards so stop once we have performed $r - p + 1$ basic steps. Never need to check for sentinels, since they will always lose. Rather than even counting basic steps, just fill up the output array from index p up through and including index r .

The pseudo code of the MERGE procedure is as follow:

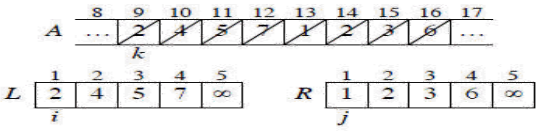
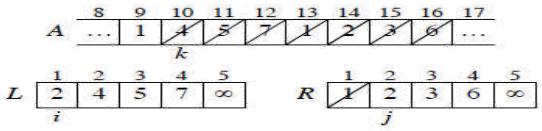
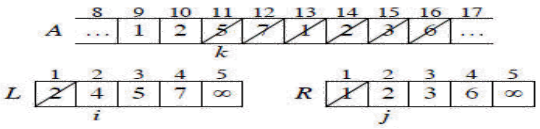
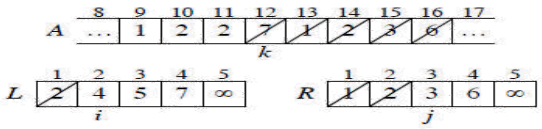
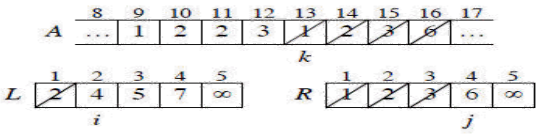
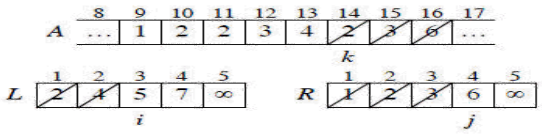
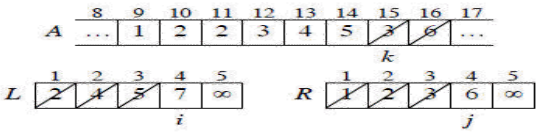
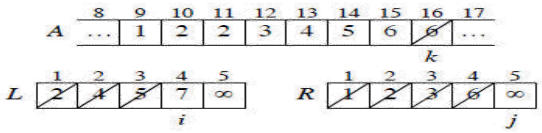
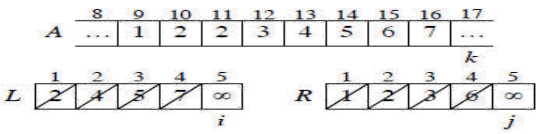
MERGE (A, p, q, r)

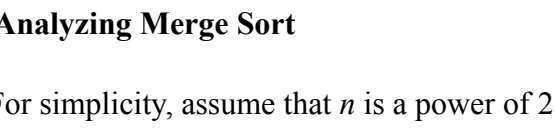
1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. Create arrays L[0 . . $n_1 + 1$] and R[0 . . $n_2 + 1$]
4. FOR $i \leftarrow 0$ TO n_1

```
5.          DO L[i] ← A[p + i]
6.      FOR j ← 0 TO n2
7.          DO R[j] ← A[q + j+1 ]
8.      L[n1 + 1] ← ∞
9.      R[n2 + 1] ← ∞
10.     i ← 0
11.     j ← 0
12.     FOR k ← p TO r
13.         DO IF L[i ] ≤ R[ j]
14.               THEN A[k] ← L[i]
15.                    i ← i + 1
16.               ELSE A[k] ← R[j]
17.                    j ← j + 1
```

Example [from CLRS-Figure 2.3]: A call of MERGE (A, 9, 12, 16). **Read the following figure row by row**. That is how we have done in the class.

- The first part shows the arrays at the start of the "for $k \leftarrow p$ to r" loop, where A[p . . q] is copied into L[1 . . n_1] and A[q + 1 . . r] is copied into R[1 . . n_2].
- Succeeding parts show the situation at the start of successive iterations.
- Entries in A with slashes have had their values copied to either L or R and have not had a value copied back in yet. Entries in L and R with slashes have been copied back into A.
- The last part shows that the sub arrays are merged back into A [p . . . r], which is now sorted, and that only the sentinels (∞) are exposed in the arrays L and R.]



Analyzing Merge Sort

For simplicity, assume that n is a power of 2 so that each divide step yields two sub problems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

- Divide: Just compute q as the average of p and r, which takes constant time i.e. $\Theta(1)$.
- Conquer: Recursively solve 2 sub problems, each of size $n/2$, which is $2T(n/2)$.
- Combine: MERGE on an n-element sub array takes $\Theta(n)$ time.

Summed together they give a function that is linear in n, which is $\Theta(n)$. Therefore, the recurrence for merge sort running time is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 , \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 . \end{cases}$$

Solving the Merge Sort Recurrence

By the master theorem in CLRS-Chapter 4 (page 73), we can show that this recurrence has the solution

$T(n) = \Theta(n \lg n)$.

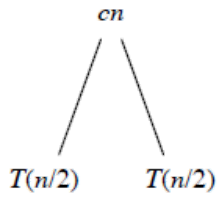
Reminder: lg n stands for $\log_2 n$.

Compared to insertion sort [$\Theta(n^2)$ worst-case time], merge sort is faster. Trading a factor of n for a factor of lg n is a good deal. On small inputs, insertion sort may be faster. But for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sorts.

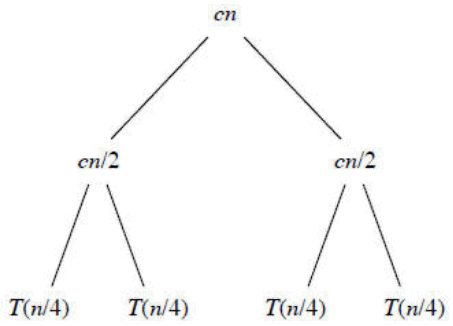
Recursion Tree

We can understand how to solve the merge-sort recurrence without the master theorem. There is a drawing of a recursion tree on page 35 in CLRS, which shows successive expansions of the recurrence.

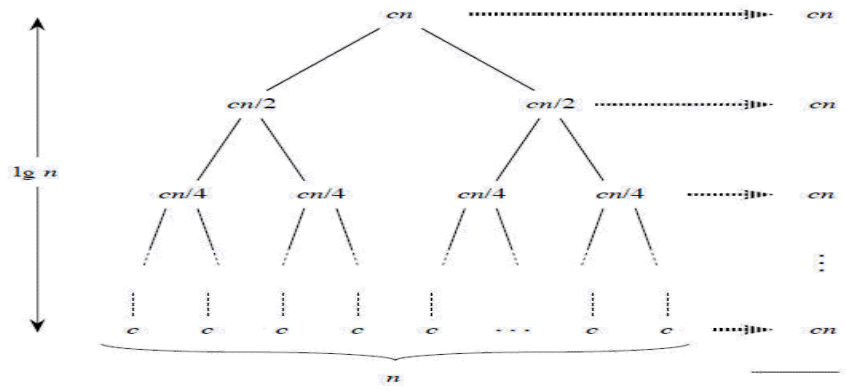
The following figure (Figure 2.5b in CLRS) shows that for the original problem, we have a cost of cn, plus the two sub problems, each costing T (n/2).



The following figure (Figure 2.5c in CLRS) shows that for each of the size-n/2 sub problems, we have a cost of cn/2, plus two sub problems, each costing T (n/4).



The following figure (Figure: 2.5d in CLRS) tells to continue expanding until the problem sizes get down to 1.



In the above recursion tree, each level has cost *cn*.

- The top level has cost *cn*.
- The next level down has 2 sub problems, each contributing cost *cn*/2.
- The next level has 4 sub problems, each contributing cost *cn*/4.
- Each time we go down one level, the number of subproblems doubles but the cost per sub problem halves. Therefore, cost per level stays the same.

The height of this recursion tree is lg *n* and there are lg *n* + 1 levels

Conclusion: - Thus we have implemented the program of merge sort algorithm with its complexities using Divide and Conquer approach. The program was successfully compiled & executed, and the output was verified. We have also analyzed the algorithm with its worst case and best case complexities.