CHAPTER 3

Divide and conquer approach

Now a day's most of the algorithms are recursive in nature. That means for solving a particular problem they call themselves repeatedly one or more times. All these algorithms follow the divide and conquer approach to accomplish the given task. In this approach whole problem is divided into several subproblems. These subproblems are similar to the original problem but smaller in size. All these subproblems are then solved recursively. The solutions obtained from these subproblems are then combined to create a solution to the original problem. Example of divide and conquer approach is binary search, merge sort, quick sort etc. At each level of the recursion the divide and conquer approach follows three steps:

Step 1: Divide

In this step whole problem is divided into several subproblems.

Step 2: Conquer

The subproblems are conquered by solving them recursively, only if they are small enough to be solved, otherwise step 1 is executed.

Step 3: Combine

Finally, the solutions obtained by the subproblems are combined to create solution to the original problem.

Advantage: It is an extremely efficient algorithm with respect to time.
Drawback: It requires a second array i.e. a temporary array as large as the original array.

Merge Sort

Merge Sort which is able to rearrange elements of a list in ascending order. Merge sort works as a divide-and-conquer algorithm. It recursively divide the list into two halves until one element left, and merge the already sorted two halves into a sorted one.

Merge sort uses a divide-and-conquer approach: **1) Divide:** Divide the array repeatedly into two halves **2) Conquer:** Stop dividing when there is single element left. By fact, single element is already sorted. I.e. Sort the two subsequences recursively using merge sort. **3) Combine:** Merges two already sorted sub arrays into one.	**Procedure for MERGESORT** MERGESORT(*A*, `low, high`) **if** `low < high` **then** `mid` ← ⌊(low + high)/2⌋ MERGESORT(*A*, `low, mid`) MERGESORT(*A*, `mid+1, high`) MERGE(*A*, `low, mid, high`) `end if` **Figure 9**

The recursion "bottoms out" when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

For accomplishing the whole task we are using two procedures MERGESORT and MERGE as shown in figure 9 and 10 respectively. In this section we are presenting the algorithm for procedure MERGESORT which takes the advantage of procedure MERGE. Procedure MERGE is used for combining the sublists. Given a sequence of n elements also called as keys A [1]… A [n], the procedure is to split them into two sets A[p…q] and A[q+1…r] with the lengths n_1 and n_2 respectively.

Figure 11 illustrates the operation of the procedure bottom-up when n is a power of 2. Let us consider an array A = [6, 3, 9, 1, 5, 4, 7, 2]. The algorithm consists of merging pairs of 1-item sequences to form sorted sequences of length 2, merging pairs of sequences of length 2 to form sorted sequences of length 4, and so on, until two sequences of length $n/2$ are merged to form the final sorted sequence of length n.

Procedure for MERGE

```
MERGE(A, low, mid, high)
l1 = low, l2 = mid + 1, i = low
while (l1 <= mid && l2 <= high)
        do if a[l1] <= a[l2]
            then b[i] ← a[l1]
                 l1← l1+ 1
                 i ← i + 1
           else
                b[i] ← a[l2]
                l2← l2+ 1
                i ← i + 1
             end if
end while
if l1 > mid
  then for k = l2 to high
          do b[i] ← a[k]
             i ← i + 1
       end for
end if


else
  then for k = l1 to mid
          do b[i] ← a[k]
             i ← i + 1
       end for
end if
[copy from array b to array a]
       for i ← low to high
             do a[i] ← b[i];
```

Figure 10

Prepared By
Sasmita Kumari Nayak

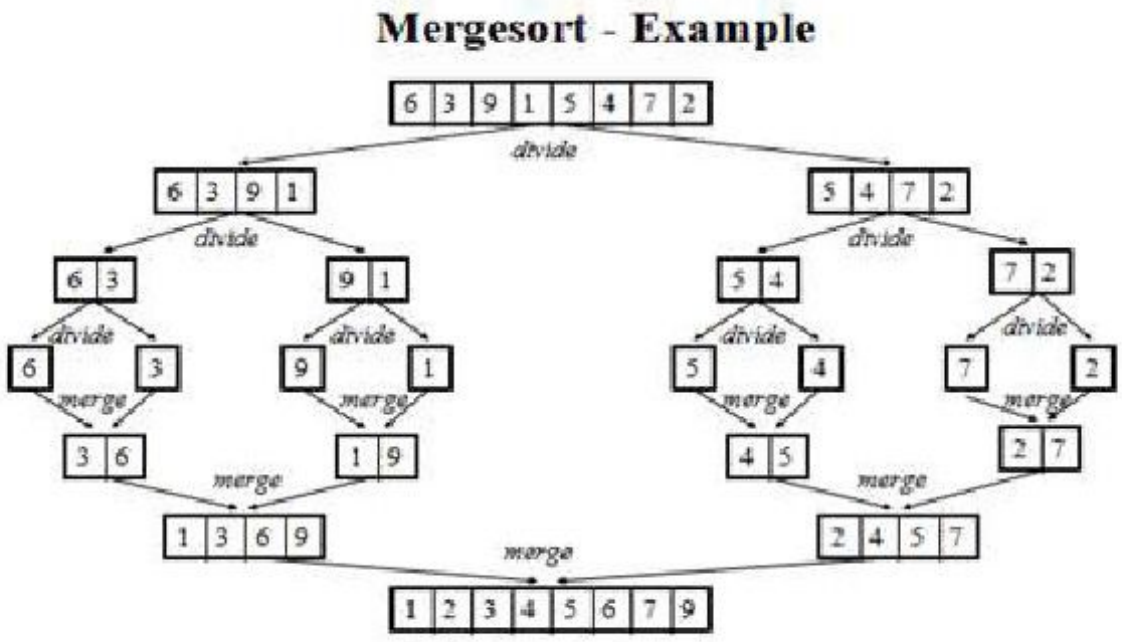

Figure 11

Mergesort – Example2

Let us consider another example of merge sort as shown in figure 12.

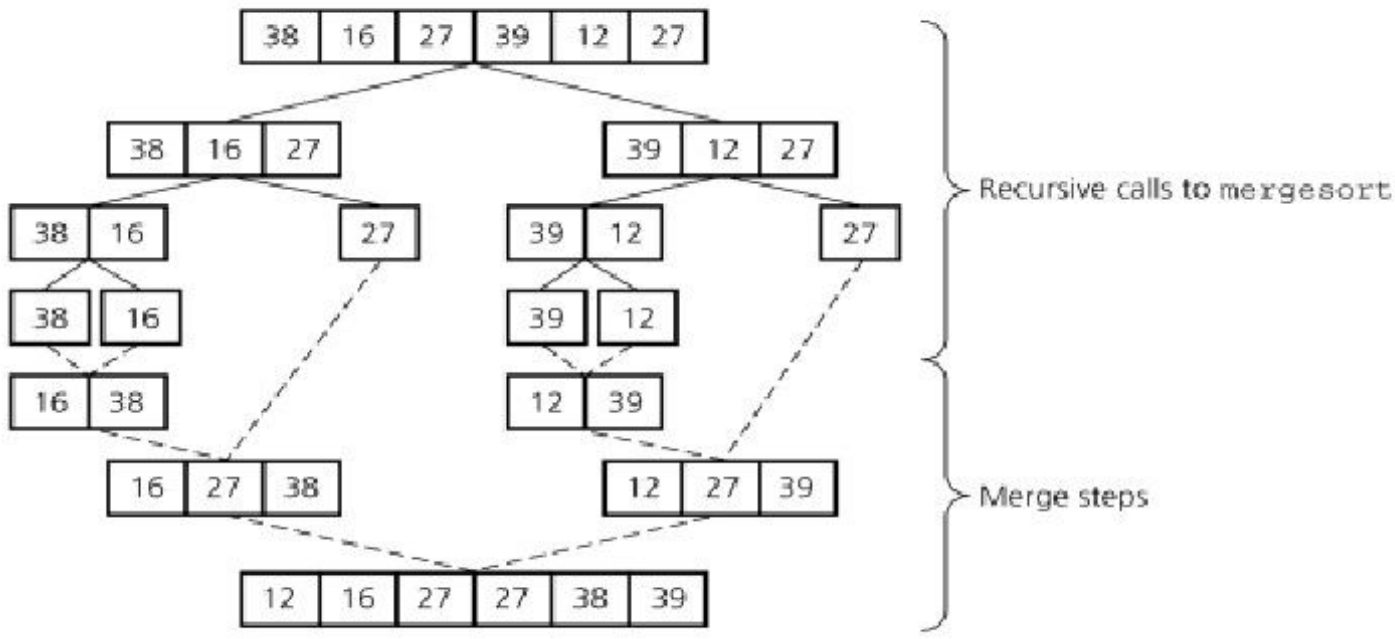

Figure 12

Analysis of merge sort

Although the pseudocode for MERGE-SORT works correctly when the number of elements is not even, our recurrence-based analysis is simplified if we assume that the original problem size is a power of 2. Each divide step then yields two subsequences of size exactly $n/2$. To set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements, we break down the running time as follows:

Prepared By
Sasmita Kumari Nayak

- **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$.
- **Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time.
- **Combine:** We have already noted that the MERGE procedure on an n-element subarray takes time $\Theta(n)$, so $C(n) = \Theta(n)$.

When we add the functions $D(n)$ and $C(n)$ for the merge sort analysis, we are adding a function that is $\Theta(n)$ and a function that is $\Theta(1)$. This sum is a linear function of n, that is, $\Theta(n)$. Adding it to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases} \qquad (2.1)$$

The worst case for merge sort occurs when we merge two sub arrays into one if the biggest and the second biggest elements are in two separated sub array. We shall see the "master theorem," which we can use to show that $T(n)$ is $\Theta(n \lg n)$, where $\lg n$ stands for $\log_2 n$. Because the logarithm function grows more slowly than any linear function, for large enough inputs, merge sort, with its $\Theta(n \lg n)$ running time, in the worst case.

Let us rewrite the above recurrence as

$$T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1, \end{cases} \qquad (2.2)$$

Where the constant c represents the time required to solve problems of size 1 as well as the time per array element of the divide and combine steps.

Prepared By
Sasmita Kumari Nayak

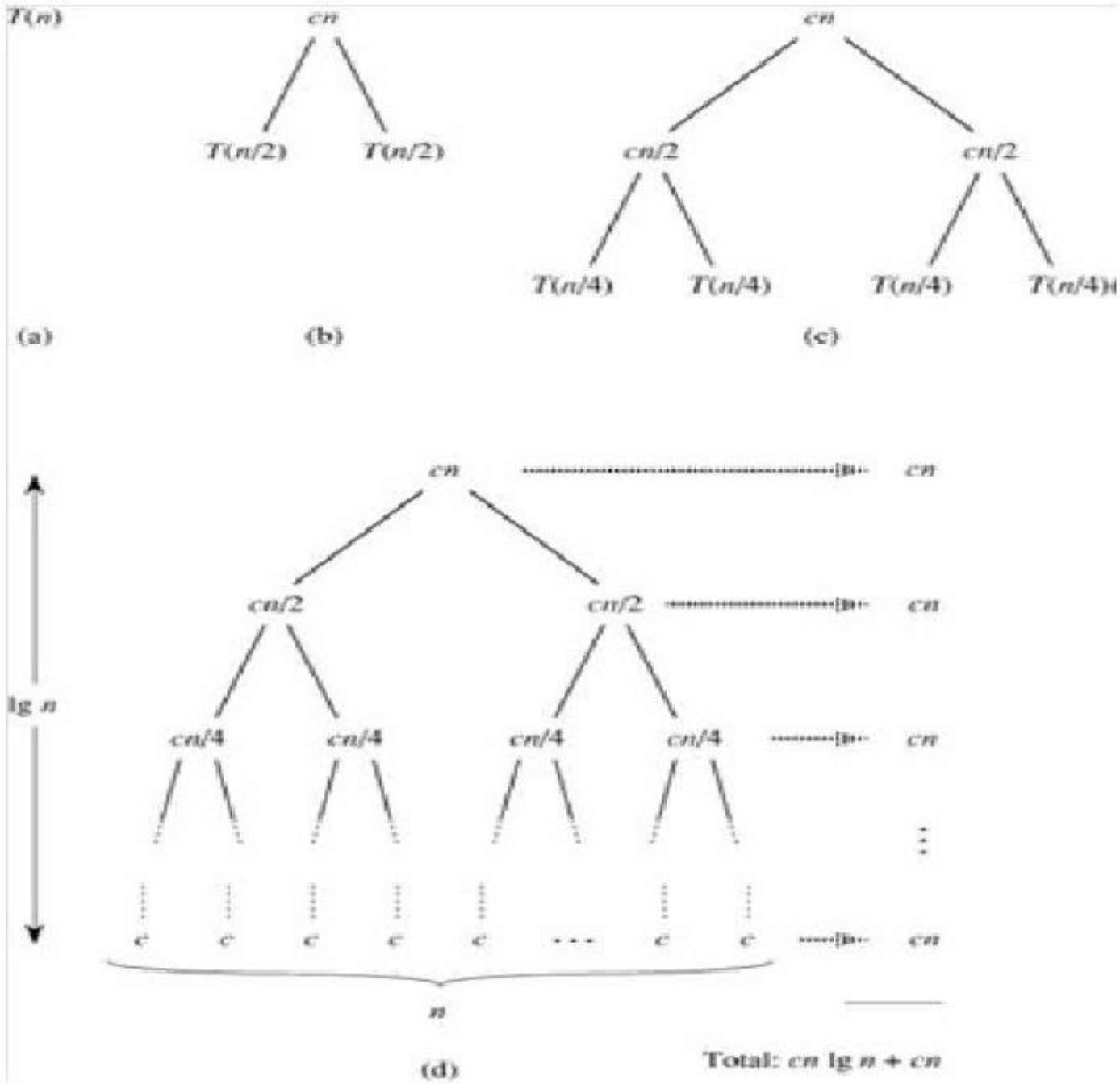

Figure 13

Figure 13 shows how we can solve the recurrence (2.2). For convenience, we assume that n is an exact power of 2. Part (a) of the figure shows $T(n)$, which in part (b) has been expanded into an equivalent tree representing the recurrence. The cn term is the root (the cost at the top level of recursion), and the two subtrees of the root are the two smaller recurrences $T(n/2)$. Part (c) shows this process carried one step further by expanding $T(n/2)$. The cost for each of the two subnodes at the second level of recursion is $cn/2$. We continue expanding each node in the tree by breaking it into its constituent parts as determined by the recurrence, until the problem sizes get down to 1, each with a cost of c. Part (d) shows the resulting tree.

Use recurrence tree method to find bound asymptotically. Ignoring the low-order term and the constant c gives the desired result of $\Theta(n \lg n)$.

Prepared By
Sasmita Kumari Nayak