
Unit 2 Divide and Conquer Technique

Divide & Conquer Technique

2.1	Introduction	1
2.2	Objectives	2
2.3	Recurrence Relation Formulation in Divide and Conquer Technique	2
2.4	Binary Search Algorithm	5
2.5	Sorting Algorithms	9
2.5.1	Merge Sort	9
2.5.2	Quick Sort	17
2.6	Integer Multiplication	25
2.7	Matrix Multiplication Algorithm	28
2.7.1	Straight forward method	28
2.7.2	Divide & Conquer Strategy for multiplication	29
2.8	Summary	36
2.9	Solution to Check Your Progress	37
2.10	Further Readings	38

2.0 Introduction

In Divide and conquer approach, the original problem is divided into two or more sub-problems recursively, till it is small enough to be solved easily. Each sub-problem is some fraction of the original problem. Next, the solutions of the sub-problems are combined together to generate the solution of the original problem. Figure 1 illustrates the working strategy of Divide and Conquer approach.

Many useful algorithms are recursive in nature such as Merge Sort, Quick Sort, Binary Search etc. All these algorithm will be examined in this unit. A large Integer multiplication and Strassen's matrix multiplication algorithm are also formulated as divide and conquer problems

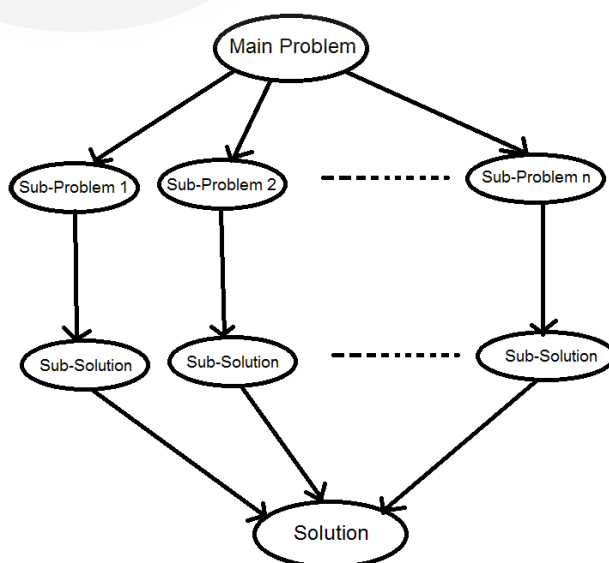


Figure 1: Working strategy of Divide and Conquer approach.

In Divide and Conquer approach, there are mainly three steps:

Step 1(Divide): The original problem is divided into one or more sub-problems in such a way that each sub-problem is equivalent to the original problem but its size is smaller than the original one. Further sub-division of each sub-problem is done till either it is directly solvable or it is impossible to perform sub-division which indicates that there is a direct solution of the sub-problem.

Step 2(Conquer): In this step, each smallest sub-problem is solved independently.

Step 3(Combine): The solution of each sub-problem is combined recursively to generate the solution of the original problem.

The unit is structured as follows. In section 2.2, we define a recurrence relation of divide & conquer approach in general and take Merge Sort and closest pair problems as examples to derive their recurrence relations. From section 2.3 to 2.6 we present several problems, derive their recurrence relations and provide solutions. Summary & solution to check your progress are given in 2.7 & 2.8 respectively.

2.1 Objectives

After reading this Unit, you will be able to:

- Define concepts of divide-and-conquer approach.
- Formulate the recurrence relation of problems that are solvable by Divide-and-Conquer approach
- Apply how divide-and-conquer approach is used to solve problems like Binary search, Quick-sort, Merge-sort, Integer Multiplication and Matrix multiplication.

2.2 Recurrence Relations Formulations in Divide and Conquer Approach

As many algorithms are recursive in nature, the computational complexity of such algorithms is defined by Divide and Conquer approach. In Divide and Conquer approach, the running time is equated as a recurrence relation which is based upon three steps:

- 1) Divide: Dividing the given problem into sub-problems.
- 2) Conquer: Each sub-problem solves itself by calling itself recursively. (Base case: If size of sub-problem is small enough, that solve it directly)
- 3) Combine: Each solution of the sub-problem is combined to obtain the original solution.

In general, the recurrence relation of a Divide and Conquer approach is presented as

Greedy Technique

$$T(n) = \begin{cases} \theta(1) & \text{if } n \leq C \\ a T(n/b) + f(n) & \end{cases}$$

where,

- $T(n)$ = Time required to solve a problem of size 'n'.
- a corresponds to the number of partitions made to a problem.
- $T(n/b)$ corresponds to the running time of solving each subproblem of size (n/b) .
- $f(n) = D(n) + C(n)$ – time required to divide the problem and combine the solutions respectively. If the problem size is small enough say, $n \leq C$ for some constant C , we have a best case which can be directly solved in a constant time: $\theta(1)$, otherwise, divide a problem of a size n into subproblems, each of $\left(\frac{1}{b}\right)$ size.

Example: Merge Sort algorithm closely follows the Divide-and-Conquer approach. which divides the array of n elements which requires to be sorted into two subarrays of $n/2$ elements each. Then, it sorts the two subarrays recursively. Finally it combines (merges) the two sorted subarrays to produce the result. The following procedure MERGE_SORT (A, p, r) sorts the elements in the subarray $A[p, \dots, r]$, where p and r are starting and end indexes of an array A . If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the divide step simply computes an index q that partitions $A[p, \dots, r]$ into two sub-arrays: $A[p, \dots, q]$ containing $\lfloor (n/2) \rfloor$ elements, and $A[q+1, \dots, r]$ containing $\lfloor (n/2) \rfloor$ elements. The pseudocode of the Merge Sort is given below in figure 2.

```

MERGE_SORT (A, p, r)
1. if (p < r)
2.   then  $q \leftarrow \lfloor (p+r)/2 \rfloor$       /*Divide
3.   MERGE_SORT (A, p, q) /*Conquer
4.   MERGE_SORT (A, q+1, r) /*Conquer
5.   MERGE (A, p, q, r)      /*Combine

```

Figure 2: Steps in merge sort algorithms

Problem1 To set up a recurrence $T(n)$ for MERGE SORT algorithm, we can note down the following points:

- Base Case: if $n=1$ (only one element in the array), the time will be $\theta(1)$.
- When there are $n>1$ elements, a running time can be derived as follows:
 - (1) Divide: Compute q as the middle of p and r , which takes constant time.
 - (2) Conquer: Two subproblems are solved recursively each of size $n/2$, which contributes $2T\left(\frac{n}{2}\right)$ to the running time.

(3) Combine: Merging of two sorted subarrays (for which we use MERGE (A, p, r) of an n element array) takes time $\theta(n)$, so $C(n) = \theta(n)$.

Thus $f(n) = D(n) + c(n) = \theta(1) + \theta(n) = \theta(n)$ which is a linear function of n.

Thus from all the above 3 steps, a recurrence relation for MERGE_SORT (A, l, P, r) in the worst case can be written as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n \geq 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & n > 1 \end{cases}$$

This algorithms will be explained in detailed in section 2.4.2

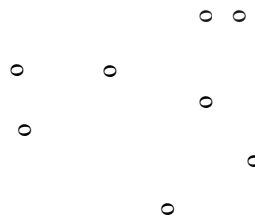
Once the recurrence relation is derived for a problem following divide & conquer technique, the next step would be solve it through any method such as Recursive tree or Substitution method or Master Method. We have $T(n) = \theta(n \log n)$.

Problem 2: Closest Pair Problem

Problem Definition- Given a set of n points in a plane. Find a pair of points which are closest together. If p1 and p2 are two points in the plane, then the Euclidean distance between p1 and p2 is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. We need to find the closest pair of points. In case the two points are having the same locations points, that pair is the closest with distance 0. What will be the total number of comparisons of pair of points? If there are n points, there will be $\frac{n(n-1)}{2}$ pair of points for comparisons.

The brute force method will take $O(n^2)$ time complexity. Since this approach requires exhaustive search, let us explore divide and conquer technique to reduce its time complexity.

The following figure illustrates a set of n points in P in a plane.



Assume all the points in P are sorted by x coordinates, it is possible to divide the points set P into two halves: the left half of P and the right half of P by drawing an imaginary vertical line as shown below in a figure

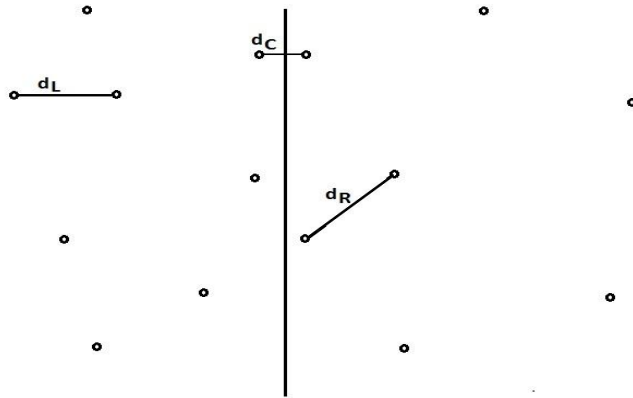


Figure 4: P is partitioned into two halves.

Shortest distances are shown as d_L and d_R in the left half and right half respectively.

The recurrence relation of the closest pair problem is similar to Merge Sort Problem i.e.,

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \theta(n) \\ &= O(n \log n) \end{aligned}$$

2.3 Binary Search

Binary search is a procedure of finding the location of an element in a sorted array. The iterative version of the algorithm is given in figure . The divide and conquer version of the algorithm proceeds as follows:

- 1) If the key is found at the middle position of the array, the algorithm terminates, otherwise
- 2) **Divide** the array into two sub-arrays recursively. If the key is smaller than the middle value, select the left part of the sub-arrays, otherwise (if it is larger) select the right part of :the sub-array. The process continues until the search interval is not empty or it cannot be broken further
- 3) **Conquer**(solve) the sub-array by determining whether the key is located in that sub-array.
- 4) Obtain the result

The recursive version of the algorithm is given in figure

Iterative Binary Search Algorithm

Input: A sorted array 'A' of size 'n' (in ascending order) and a element 'x' to be searched.

Output: Index of 'x' in 'A' if found, else return 0.

//**lowerIndex:** Index of the first element in the array, considered to search for 'x'.

//**upperIndex:** Index of the last element in the array, considered to search for 'x'.

//**middleIndex:** Index of the middle element in the array under consideration.

// **A[middleIndex]:** Value at the middle index of the array 'A'.

```

{
    lowerIndex = 0
    upperIndex = n
    while(lowerIndex <= upperIndex){
        middleIndex = ((lowerIndex + upperIndex)/2)
        if(A[middleIndex] == x) // if 'x' is found
            return middleIndex; // Index of 'x' in 'A'
        else if (x < A[middleIndex])
            upperIndex = middleIndex - 1;
        else
            lowerIndex = middleIndex + 1;
    }
    return -1 // if 'x' is not found
}

```

Figure 5: iterative_binary search Algorithm

Recursive Binary Search Algorithm

```

Recursive_Binary_Search( lowerIndex, upperIndex, A [ ], x )
{
    while (lowerIndex ≤ upperIndex)
        middleIndex = (lowerIndex + upperIndex)/2 ;
        if A[ middleIndex] = x
            return middleIndex ; /* terminate the program*/
        else if ( x < A[ middleIndex])
            Recursive_Binary_Search(lowerIndex,middleIndex-1,A[ ],x)
        else
            Recursive_Binary_Search(middleIndex+1, upperIndex, A [ ], x)
}

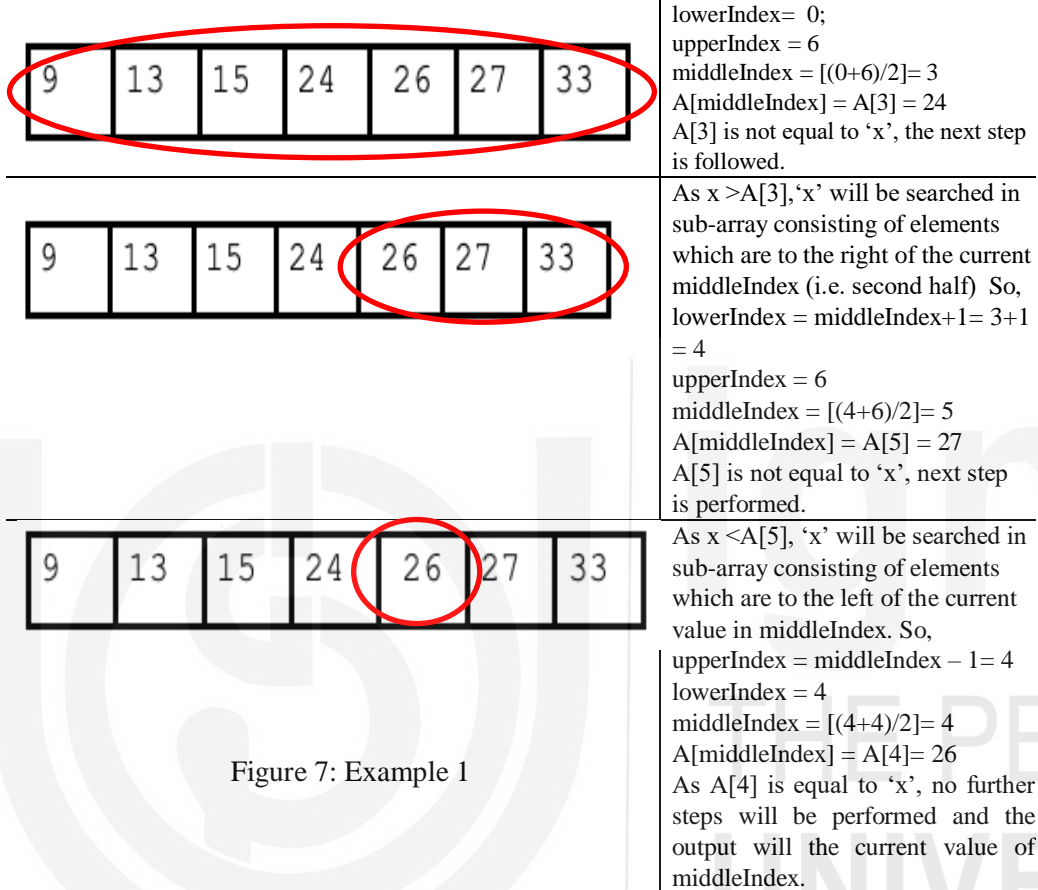
```

Figure 6: Recursive Binary Search

Let us apply the algorithm to the following problems in example 1 (Figure 7) & example 2 (Figure 8)

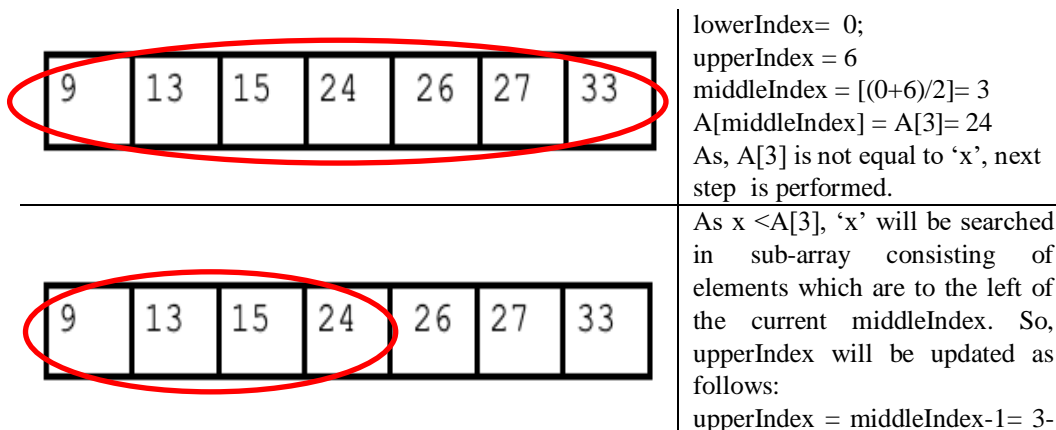
Example 1: Consider a sorted array $A=[9,13,15,24,26,27,33]$ and the element to be searched is '26'.

Solution: For the given array 'A', consider 'x' as 26. Following steps will be performed by binary search. Here, the circle denotes the elements considered in a particular iteration.



Example 2: Consider a sorted array $A = [9,13,15,24,26,27,33]$ and the element to be searched is '17'.

Solution: For the given array 'A', consider 'x' as 17. Following steps will be performed by binary search. Here, red circle denotes the elements considered in a particular iteration.



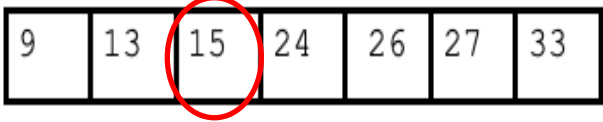

	<div>1 = 2 lowerIndex = 0 Now, middleIndex = [(0+2)/2]= 1 A[middleIndex] = A[1]= 13 As A[1] is not equal to 'x', next step is performed.</div>
<div></div>	<div>As $x > A[1]$, 'x' will be searched in sub-array consisting of elements which are to the right of the current value in middleIndex. So, lowerIndex will be updated as follows: lowerIndex = middleIndex + 1 = 2 upperIndex = 2 Now, middleIndex = [(2 + 2)/2]= 2 A[middleIndex] = A[2]= 15 As A[2] is not equal to 'x', next step is performed.</div>
<div></div>	<div>As $x > A[2]$, 'x' will be searched in sub-array consisting of elements which are to the right of the current value in middleIndex. So, lowerIndex will be updated as follows: lowerIndex = middleIndex + 1 = 3 upperIndex = 2 As lowerIndex > upperIndex, no further steps is performed. The output will be -1 which indicates 'x' is not in the array 'A'.</div>

Figure 8: Example 2

2.3.2 Analysis of Binary Search:

Method 1: Let us assume for the moment that the size of the array is a power of 2, say 2^k . Each time in the while loop, when we examine the middle element, we cut the size of the sub-array into half. So Before the 1st iteration size of the array is 2^k .

After the 1st iteration size of the sub-array of our interest is: 2^{k-1}

After the 2nd iteration size of the sub-array of our interest is: 2^{k-2}

.....

.....

After the kth iteration size of the sub-array of our interest is: $2^{k-k}=1$

So we stop after the next iteration. Thus we have at most $(k+1) = (\log n + 1)$ iterations.

Since with each iteration, we perform a constant amount of work: computing a mid point and few comparisons. So overall, for an array of size n, we perform $C \cdot (\log n + 1) = O(\log n)$ comparisons. Thus $T(n) = O(\log n)$

2.3.3 Recurrence Relation of Binary Search

Greedy Technique

The complexity of divide and conquer approach is defined by recurrence relation as follows:

$$T(n) = a T(n/b) + f(n)$$

As binary search follows divide and conquer approach and performs dividing the array but searching on only one sub-array in an iteration, the computational complexity of binary search technique can be defined as recurrence relation in the following form:

$$T(n) = T(n/2) + k$$

where, a, b, and f(n) are replaced with values 1, 2, k (constant less than n), respectively. K is constant value for divide and conquer operation on solving this recurrence relation by substitution method, the computational complexity for binary search is $O(\log n)$.

Check Your Progress 1

Question 1: What will be minimum number of comparisons required to find the minimum and the maximum of 100 numbers?

Question 2: Given an array $A = [45, 77, 89, 90, 94, 99, 100]$ and key = 100. What are the mid values of an array (corresponding array elements) generated in the first and second iterations?

2.4 Sorting Algorithms

It is process of rearranging the elements of a list in either ascending or descending order. For example, consider a list of elements $\langle a_1, a_2, a_3, \dots, a_n \rangle$ as input which needs to be sorted in ascending order, then the output of the sorting algorithm will be rearranging the list such that $\langle a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n \rangle$. Generally, there are number of sorting algorithms like bubble sort, merge sort, radix sort, quick sort and many more. In this section, we will cover Merge-Sort and Quick-Sort sorting algorithms as they are based on divide and conquer approach.

2.4.1 Merge-Sort

As discussed in the above section, Merge-Sort algorithm is a divide-and-conquer based sorting algorithm. It follows a divide and conquer approach to perform sorting. The algorithm in divide step repeatedly partitions an array into several sub arrays until each sub array consists of a single element. Then, it merges sub-problem solutions together according to following steps:

- **Divide** the array into two equal sub-arrays Compare the sub-array's first elements
- **Conquer**(solve) each sub-array by performing sorting operation. Apply the recursion unless the array is sufficiently small.
- **Combine**(Merge) the solutions into a single array

The following example illustrates the above steps:

Suppose the array has following numbers:

37 20 23 30 18 15 27 17

In divide step , the array is divided into two subarrays

37 20 23 30 and 18 15 27 17

In conquer step , we sort each subarray

20 23 30 37 and 15 17 18 27

In combine(merge) ,all the subarrays are merged in sorted order

15 17 18 20 23 27 30 37

Pseudo-code of the Merge-Sort

Algorithm 2: Merge-Sort (X, m, n)

```

if ( $m < n$ ) {
     $i = (m+n)/2$ ;           //Divide
    Merge-Sort ( $X, m, i$ );  //Conquer
    Merge-Sort ( $X, i+1, n$ ); //Conquer
    Merge ( $X, m, i, n$ );    //Combine
}
else {
    Already sorted
}
    
```

Figure 7: Merge Sort Algorithm

In figure 7, X is the given array of size $m \times n$. It first checks for the number of elements in X . If $m \Rightarrow n$, then the given array is sorted already as there will be at most one element only. Alternatively, it will first perform the Divide step by computing the middle index 'i' of X and then partitions it into two sub-arrays and calls the same algorithm on each sub-array recursively. Finally, the merge operation performs the Combine step. The pseudo-code to perform Merge operation is detailed in figure 8.

The following algorithm merges the two sorted subarrays $S1[]$ and $S2[]$ into $S[]$

Pseudo-code for Merge Operation

Problem Definition- Merge two sorted arrays $S1[]$ and $S2[]$ into one sorted array $S[]$

Inputs – positive integers m and n , sorted arrays $S1[]$ and $S2[]$, indexed from $1..m$ and $1..n$ respectively

Output- an array S indexed from 1 to $m+ n$ containing sorted values from $S1$ and $S2$

Merge($S1[], S2[], S[] \ m, n$)

{

int i, j, k

/* initialization of i, j, k */

```

i=1; j = 1;k=1
while ( i ≤ m && j ≤ n) /* scanning S1 and S2*/
if S1[i] < S2[j]
{
S[k] = S1[i];
i++;
}
else
{
S[k]= S2[j];
j++;
}
k++
}
if(i> m)
copy S2[j]through S2[n]to S[k] through S[m+n]
else
copy S1[i] through S[m] to S[k] through S[m+n]
}

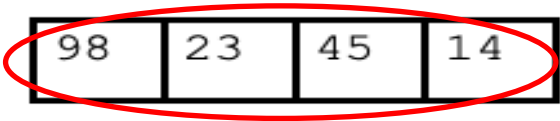
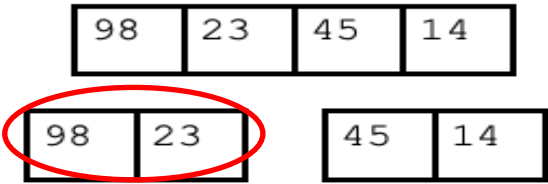
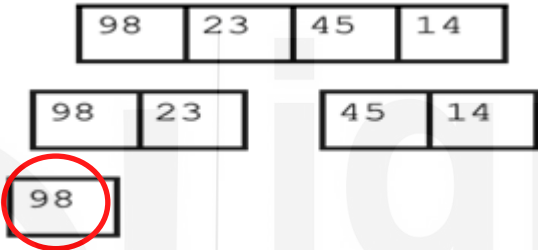
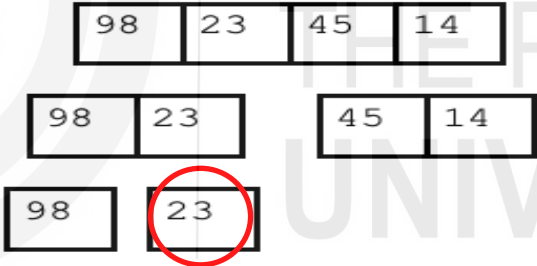
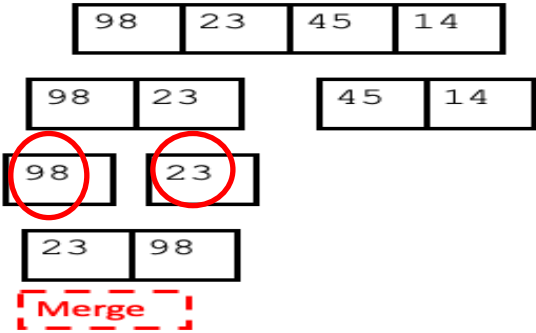
```

Figure 8: Merge Algorithm

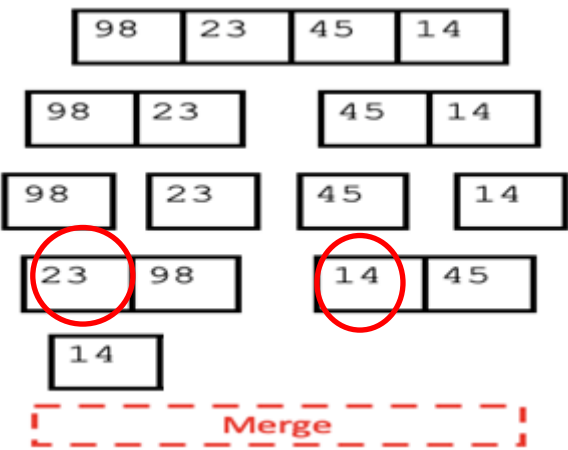
Note that the merge algorithm which requires an amount of *additional storage* equal to the amount of storage for all of the original data .

Example 3: Consider an array ‘A’ of size ‘4’ as [98, 23, 45, 14]. Perform Merge-Sort on ‘A’.

Solution: Merge-Sort follows partitioning of an array into two sub-arrays from the middle index. This partition is performed recursively on each sub-array till individual elements are obtained. Then, merge operation is performed on individual elements in a sorted order recursively. Following are the steps of Merge-Sort to sort array ‘A’. Here, red circle denotes the considered elements at a particular step.

	<p>m = 0; n = 3 As m < n, if condition is TRUE. So, calculate 'mid' as: mid = $\lfloor [(0 + 3)/2] \rfloor = 1$; Now, recursive call to Merge-Sort(A, 0, 1);</p>
	<p>m = 0; n = 1; As m < n, if condition is TRUE. So, calculate 'mid' as: mid = $\lfloor [(0 + 1)/2] \rfloor = 0$ Now, recursive call to Merge-Sort(A, 0, 0);</p>
	<p>m = 0; n = 0; As 'm' is not less than 'n', if condition is False. So, else condition is executed. This completes recursive call of Merge-Sort(A, 0, 0). Now, recursive call to Merge-Sort(A, 1, 1);</p>
	<p>m = 1; n = 1; As 'm' is not less than 'n', if condition is False. So, else condition is executed. This completes recursive call of Merge-Sort(A, 1, 1). Now, call Merge(A, 0, 0, 1);</p>
	<p>In Merge, each value in the two arrays are compared individually. Then, they are stored in sorted order in a new array. As there are single values in the two arrays, both values are compared and stored accordingly in a new array termed as 'X'. i.e., 98 > 23, 23 is placed first in 'X' while 98 is placed after 23 in 'X'. This completes call of Merge(A, 0, 0, 1). Now, recursive call to Merge-Sort(A, 2, 3);</p>

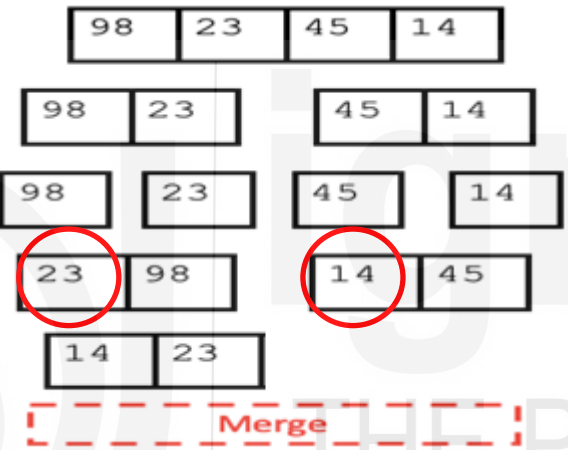
	<p> $m = 2;$ $n = 3;$ As $m < n$, if condition is TRUE. So, calculate 'mid' as: $\text{mid} = \text{floor}[(2+3)/2] = 2$ Now, recursive call to <i>Merge-Sort</i>(A, 2, 2); </p>
	<p> $m = 2;$ $n = 2;$ As 'm' is not less than 'n', if condition is False. So, <i>else</i> condition is executed. This completes recursive call of <i>Merge-Sort</i>(A, 2, 2). Now, recursive call to <i>Merge-Sort</i>(A, 3, 3); </p>
	<p> $m = 3;$ $n = 3;$ As 'm' is not less than 'n', if condition is False. So, <i>else</i> condition is executed. This completes recursive call of <i>Merge-Sort</i>(A, 3, 3). Now, call <i>Merge</i>(A, 2, 2, 3); </p>
	<p> In <i>Merge</i>, each value in the two arrays are compared individually. Then, they are stored in sorted order in a new array. As there are single values in the two arrays, both values are compared and stored accordingly in a new array termed as 'X'. i.e., $45 > 14$, 14 is placed first in 'X' while 45 is placed after 14 in 'X'. This completes call of <i>Merge</i>(A, 2, 2, 3). Now, call to <i>Merge</i> (A, 0, 1, 3); </p>



In *Merge*, each value in the two arrays are compared individually. Then, they are stored in sorted order in a new array.

In case of multiple values in any of the two arrays, each value of two arrays are considered one by one and compared to store in a new array termed as 'X' accordingly.

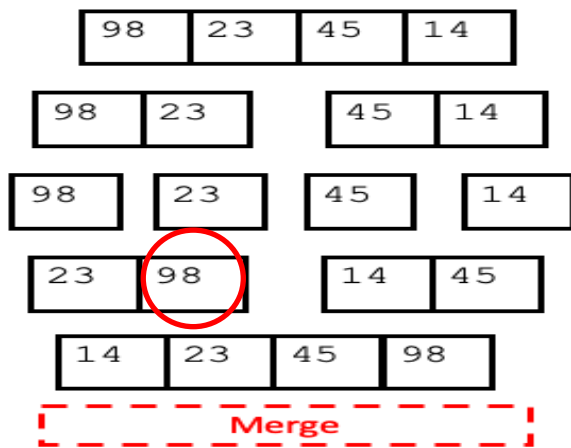
As 23 is greater than 14, 14 is placed first in 'X'. The value 23 will not be placed after 14 in 'X' but will be compared with next value in the other array.



As 23 is less than 45, 23 is placed after 14 in 'X'. The value 45 will not be placed after 23 in 'X' but will be compared with next value in the other array.



As 45 is less than 98, 45 is placed after 23 in 'X'. The value 98 will not be placed after 45 in 'X' but will be compared with next value in the other array.



To compare 98, there is no element in the other array. So, the value 98 will be placed after 45 in 'X'.

Finally, the sorted array is: [14, 23, 45, 98]

Figure 9: Example 3: Merge Sort

Analysis of MERGE-SORT Algorithm

- For simplicity, assume that n is a power of $\Rightarrow 2$ each divide step yields two sub-problem, both of size exactly $n/2$.
- The base case occurs when $n = 1$.
- When $n \geq 2$, then

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = O(1)$

Conquer: Recursively solve sub-problems, each of size $\frac{n}{2} \Rightarrow 2T\left(\frac{n}{2}\right)$

Combine: MERGE an n -element subarray takes $O(n)$ time $\Rightarrow C(n) = O(n)$

❖ $D(n) = O(1)$ and $C(n) = O(n)$

❖ $F(n) = D(n) + C(n) = O(n)$, which is a linear function in n .

Hence Recurrence for Merge Sort algorithm can be written as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n) & \text{if } n \geq 2 \end{cases} \quad (1)$$

This Recurrence 1 can be solved by any of two methods:

(1) Master method or

(2) by Recursion tree method:

1) Master Method:

$$T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n \quad \text{-- (1)}$$

By comparing this recurrence with $T(n) = 2T\left(\frac{n}{2}\right) + f(n)$

We have: $a = 2$

$$b = 2$$

$$f(n) = n$$

$$n^{\log_b a} = n^{\log_2 2} = n; \text{ Now compare } f(n) \text{ with } n^{\log_2 2} \text{ i.e. } (n^{\log_2 2} = n)$$

Since $f(n) = n = O(n^{\log_2 2}) \Rightarrow$ Case 2 of Master Method

$$\Rightarrow T(n) = \theta(n^{\log_b a} \cdot \log n) \\ = \theta(n \cdot \log n)$$

$$\text{Total} = C \cdot n + C \cdot n + \dots + (\log_2 n + 1) \text{ terms}$$

$$= C \cdot n (\log n + 1)$$

$$= \theta(n \log n)$$

Recursion tree:

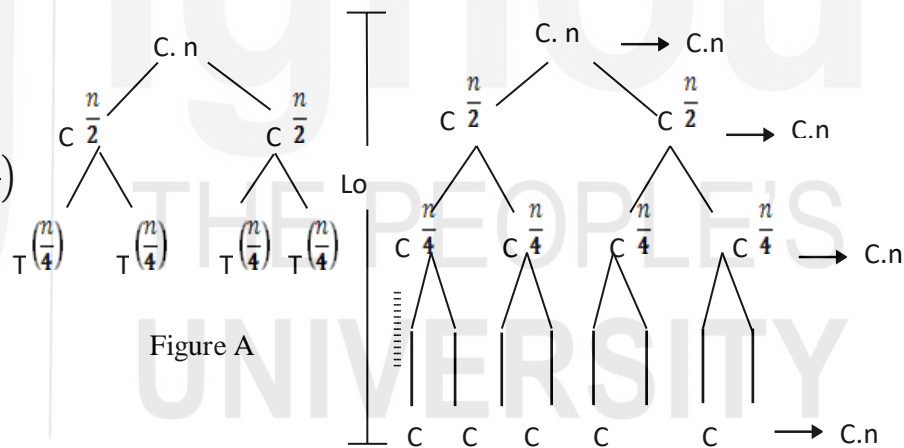
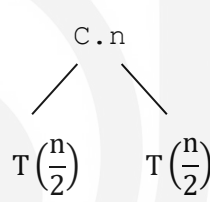


Figure B

Figure 10 : Recursion Tree of Merge Sort

On solving these recurrence relations, $T(n)$ will be $O(n \log n)$.

Note:

- Merge-Sort is not an in-place algorithm. However, it is an out-place algorithm as it needs $O(n)$ extra space to perform merge operation.
- It is not suitable for array of small size.
- It is a stable algorithm which means that the order of repeated elements in the given is not changed in the sorted array.

2.4.2 Quick-Sort

Quick-Sort is another sorting algorithm that works on the principle of divide-and-conquer approach. It works by arranging the elements in an array by identifying their correct place (or index). Moreover, it sorts elements within the list without requiring any additional space as compared to Merge-Sort which requires additional space to perform sorting. Due to which, Quick-Sort provides advantage of performing in-place sorting. The arrangement of elements in the input array (which is to be sorted) effects the running time of Quick-Sort.

Generally, it is the fastest among all sorting algorithms in practice and that's why, it is named as Quick-Sort. The running time of quick-sort is $O(n \log n)$ in average scenario. However, if the elements are sorted already, then it is the worst-case scenario for quick-sort and it's running time is $O(n^2)$. Quick sort algorithm is different from Merge Sort algorithm is that in Quick Sort, the array is partitioned around the pivot value. Smaller elements than the pivot value are placed at the left side and larger elements at the right side (best case). This is the central idea of partitioning algorithm. Quick-sort algorithm performs following steps to sort the elements of a given array 'A' of size $p \times r$:

- 1) **Divide:** In this, an element of the given array is considered as pivot element whose correct index 'q' in the array is determined by rearranging the array elements. Then, the given array $A[m \dots n]$ is partitioned into two sub-arrays, $A[p..q]$ and $A[q+1..r]$, in such a way that all the elements in $A[p..q]$ are smaller than $A[q]$ and all the elements in $A[q+1..r]$ are greater than $A[q]$. Generally, the procedure which is used to perform this step is termed as Partition. The output of this procedure is the index 'q' which divides the given array 'A'.
- 2) **Conquer:** The partitioned sub-arrays, $A[p..q]$ and $A[q+1..r]$, recursively call the step (1) to perform sorting of the elements.
- 3) **Combine:** As all the elements are sorted in their respective places, there is no need to perform combine step and the array is sorted.

The basic concept behind Quick-Sort is as follows: Suppose we have an unsorted input data $A[p \dots r]$ to sort. Here **Partition** procedures always select a last element $A[r]$ as a Pivot element and set the position of this $A[r]$ as given below:

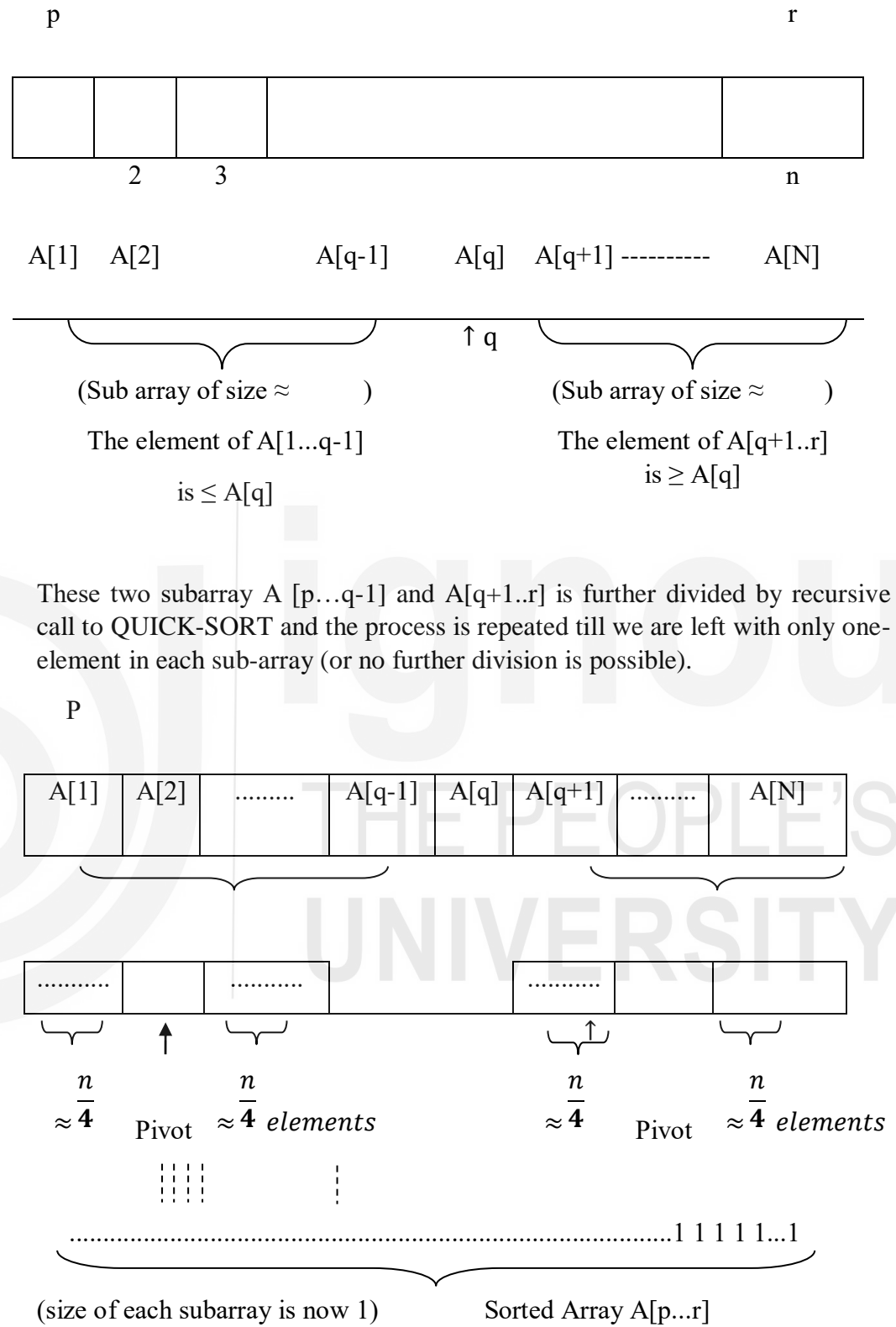


Figure 11: Graphic Representation of partitioning procedure in Quick Sort
Quick sort algorithm. is presented below:

Algorithm 4: QUICK-SORT(A, p, r)

Greedy Technique

```

if (p < r)
{
    q = PARTITION(A, p, r); // Divide
    QUICKSORT(A, p, q);      // Conquer
    QUICKSORT(A, q+1, r);    // Conquer
}

```

Figure 11: Quick Sort Algorithm

A brief description of the algorithm.

- To sort an array A with n -elements, a initial call to QuickSort in QUICKSORT ($A, 1, n$)
- QUICKSORT (A, p, r) uses a procedure Partition (), which always select a last element $A[r]$, and set this $A[r]$ as a Pivot element at some index (say q) in the array $A[p..r]$.

The PARTITION () always return some index (or value), say q , where the array $A[p..r]$ partitioned into two subarray $A[p..q-1]$ and $A[q+1..r]$ such that $A[p..q-1] \leq A[q]$ and $A[q] \leq A[q+1..r]$. The partition algorithm is presented in figure 12.

Algorithm 5: PARTITION(A, m, n)

```

PARTITION (A, p, r)
{
1:  x ← A[r]          /* select last element
2:  i ← p - 1          /* i is pointing one position
                        before than p, initially
3:  for j ← p to r - 1 do
4:      {
5:          if A[j] ≤ x
6:              {
5:                  i ← i + 1
6:                  Exchange (A[i] ↔ A[j])
              }
            } /* end for
7:  Exchange (A [i + 1] and A[r])
8:  return ( i+ 1)
}

```

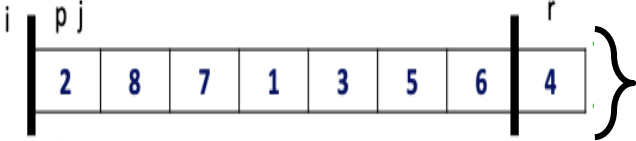
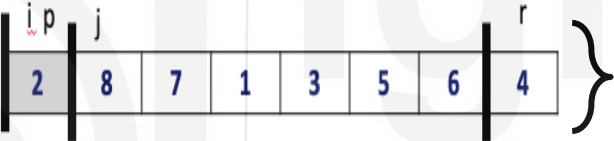
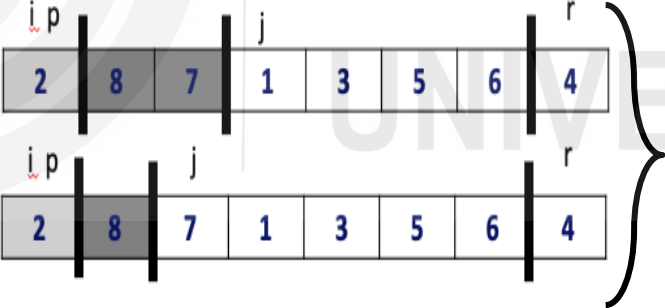
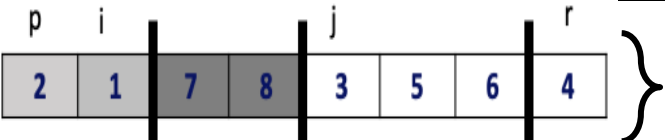
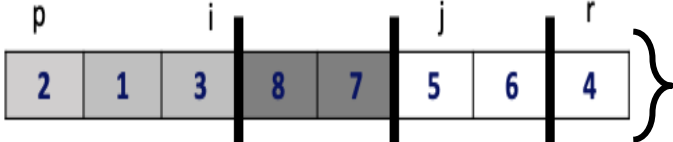
Figure 12: Partition Algorithm

The running time of PARTITION procedure is $\theta(n)$, since for an array $A[p..r]$, the loop at line 3 is running $O(n)$ time and other lines of code take constant time i.e. $O(1)$ so overall time is $O(n)$.

Let us consider one Quick Sort example based on the algorithm presented above.

Example 4: Consider an array ‘A’ of size ‘8’ as [2, 8, 7, 1, 3, 5, 6, 4]. Perform Quick-Sort on ‘A’.

Solution: The sorting of the given array ‘A’ is illustrated below. We have considered four temporary variables i.e., i, p, j, and r, to perform Quick-Sort. Here, ‘r’ represents the chosen pivot element. As Quick-Sort partitions array into two sub-blocks, the elements of the first block are highlighted with light shade while elements of the second block are highlighted with dark shade.

	The initial array of 8 elements and variable settings. None of the elements have been placed in either of the first two partitions.
	The value 2 is “swapped with itself” and put in the partition of smaller values.
	The values 8 and 7 are added to the partition of larger values
	The values 1 and 8 are swapped, and the smaller partition grows.
	The values 3 and 7 are swapped, and the smaller partition grows.

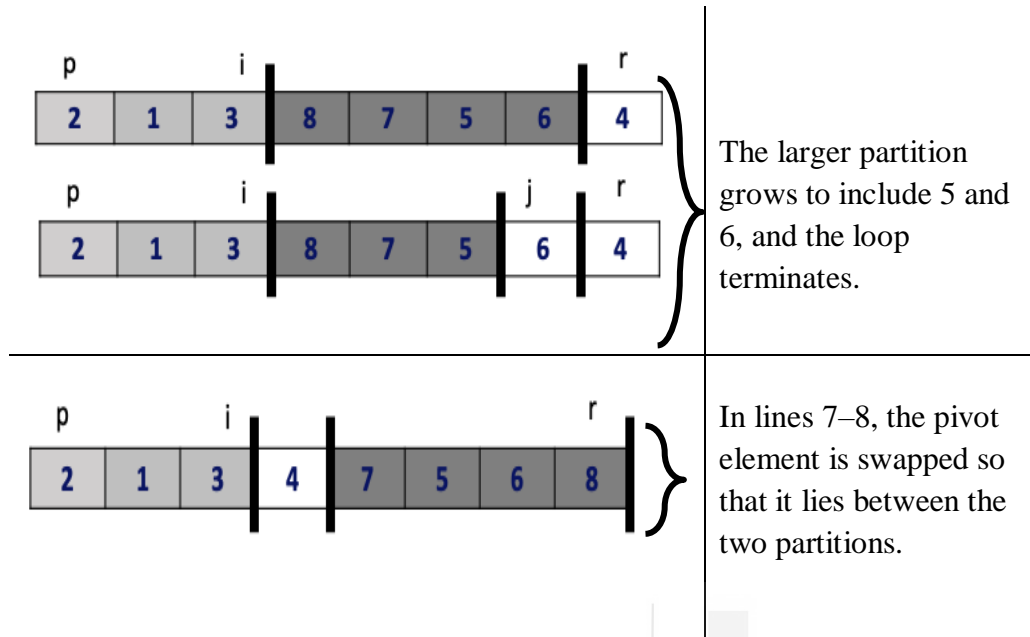


Figure 12: An example of Partition procedure

Analysis of Quick-Sort:

The computation complexity of Quick-Sort is based on the arrangement of array elements. Arrangement of elements affects the partitioning of an array. If partitioning is unbalanced, the partitioning procedure will perform more number of times in comparison to balanced partition. So, more number of calls to partitioning procedure means high computational complexity. Therefore, the Quick-Sort has different complexity in different scenarios:

Partitioning of the subarrays depends on the input data we receive for sorting.

Best Case: If the input data is not sorted, then the partitioning of subarray is balanced; in this case the algorithm runs asymptotically as fast as merge-sort (i.e. $O(n \log n)$).

Worst Case: If the given input array is already sorted or almost sorted, then the partitioning of the subarray is unbalancing in this case the algorithm runs asymptotically as slow as Insertion sort (i.e. $\theta(n^2)$).

Average Case: Except best case or worst case. The figure (a to c) shows the recursion depth of Quick-sort for Best, worst and average cases:

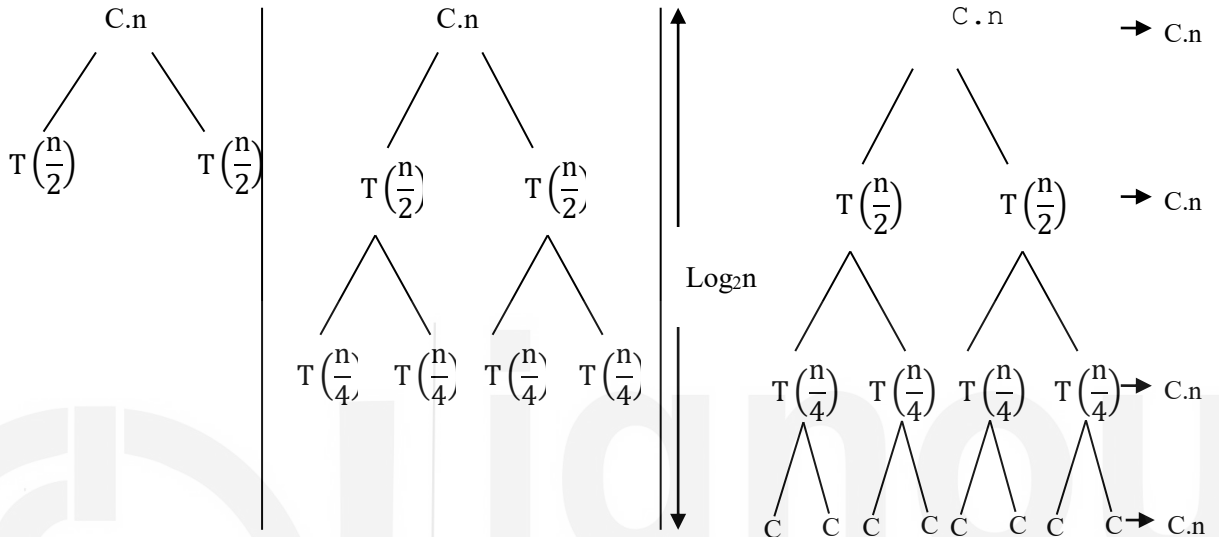
Best Case (Input array is not sorted)

The best case behaviour of Quicksort algorithm occurs when the partitioning

procedure produces two regions of size $\approx \frac{n}{2}$ elements.

In this case, Recurrence can be written as: $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$

Method 1: Using Master Method; we have $a=2$; $b=2$, $f(n)=n$ and $n^{\log_b a} = n^{\log_2 2} = n$



❖ $F(n) = n = O(n^{\log_2 2}) \rightarrow$ Case 2 of master method
 $T(n) = \theta(n \log n)$

Figure 12: Method 2: Recursion Tree:

$$T(n) = 2T\left(\frac{n}{2}\right) + C \cdot n$$

$$\begin{aligned} \text{Total} &= C \cdot n + C \cdot n + \dots + \log_2 n \text{ terms } (c) \\ &= C \cdot n \log_2 n \\ &= \theta(n \log n) \end{aligned}$$

Worst Case:

[When input array is already sorted]

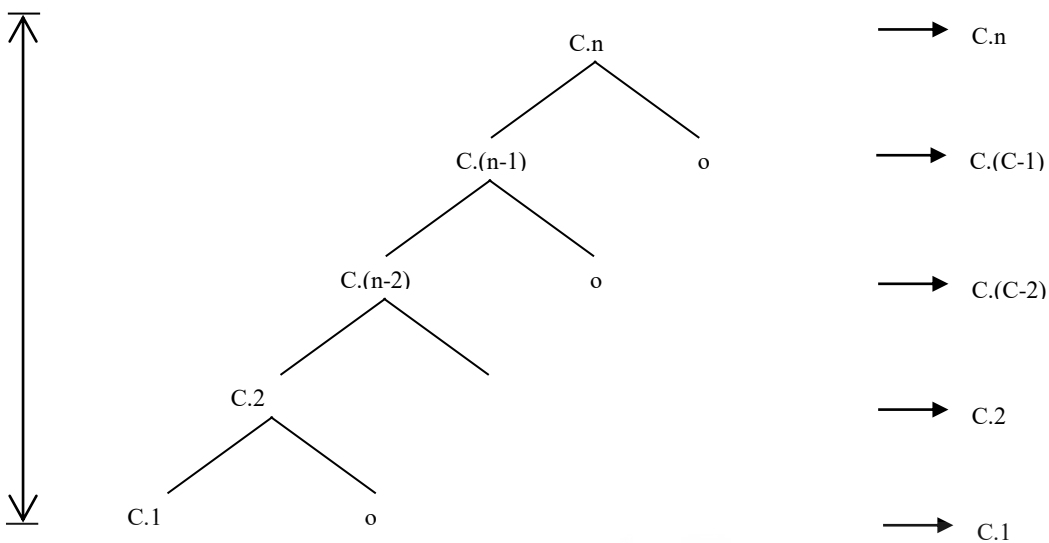
The worst case behaviour for QuickSort occurs, when the partitioning procedures one region with $(n-1)$ elements and one with 0-elements \rightarrow completely unbalanced partition.

In this case:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \theta(n) \\ &= T(n-1) + 0 + C \cdot n \end{aligned}$$

Recursion Tree:

Greedy Technique



$$\text{Total} = C (n+(n-1)+(n-2)+\dots+2+1)$$

$$= C \cdot \left(\frac{n(n+1)}{2} \right) = O(n^2)$$

Figure 13: Unbalanced Recursion tree

The worst case scenario in Quick Sort can be avoided with the following consideration:

- Pick up any element as a pivot randomly instead of the last element of an array.

The following code fragment illustrates the concept:

```
Randomized_Partition ( A, p, r)
```

```
{
```

```
    PivotIndex = random( p, r)
```

```
    exchange( A[PivotIndex], A[r])
```

```
    partition( A,p,r)
```

```
}
```

Now let us modify the QuickSort()

```
QuickSort(A,p,r)
```

```
if (p < r)
```

```
{
```

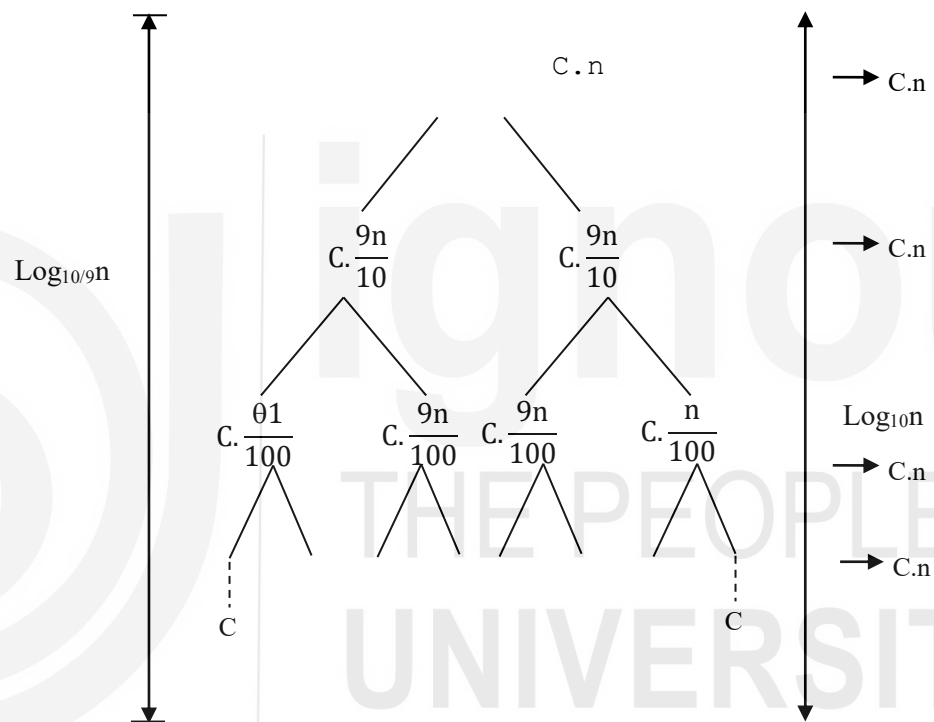
```
    PivotIndex = Randomized_Partition( A, p,r)
```

```
    QuickSort( A, p, PivotIndex-1)
```

$$\}$$
$$\}$$

Quick sort average running time is much closer to the best case. Suppose the PARTITION procedure always produces a 9-to-1 split so recurrence can be:

$$\mathbf{T}(\mathbf{n}) = \mathbf{T}\left(\frac{\mathbf{9n}}{\mathbf{10}}\right) + \mathbf{T}\left(\frac{\mathbf{n}}{\mathbf{10}}\right) + \theta(\mathbf{n})$$



For Bigger height

$$\begin{aligned} \text{Total} &= C.n + C.n + \dots \\ &= C.n \log_{10/9} n \\ &= O(n \log n) \end{aligned}$$

$$\left. \begin{array}{l} T(n) = O(n \log n) \quad - (1) \\ \& T(n) = O(n \log n) \quad - (2) \end{array} \right\} \Rightarrow T(n) = \Theta(n \log n)$$

24

Check Your Progress 2

Greedy Technique

Question 1: Consider the following elements and sort in ascending order using merge sort: 6, 2, 11, 7, 5, 4.

Question 2: What are properties of a Merge-Sort.

Question 3: Consider first element of the array 'A' = [7, 11, 14, 6, 9, 4, 3, 12] as the pivot element. What will be the sequence of elements on applying Quick-Sort on this, after the correct position of the considered pivot element.

Question 4: A machine needs a minimum of 200 sec. to sort 1000 elements by Quick-Sort. Approximately what will be the minimum time needed to sort 200 elements ?

2.5 Integer Multiplication

The brute force algorithm for multiplying two large integer numbers which everyone of us uses by hand, takes quadratic time i.e., $O(n^2)$. Because each digit of one number is multiplied by each digit in another number. Let us explore the better algorithm, which applies divide and conquer technique for integer multiplication for large numbers.

Assume X and Y are two n digits number. Divide X and Y into two halves of approximately $n/2$ digits each. The following examples illustrate the division process:

- (i) $657,138 = 657 * 10^3 + 138$
- (ii) $6578,381 = 6578 * 10^3 + 381$

Let us generalize the number representation. If Z is an n-digit number, it would be divided, into two halves, the first half with Ceiling with $\lceil n/2 \rceil$ and the second half with Floor $\lfloor n/2 \rfloor$ as shown below:

$$Z(\text{ n digit number}) = X_L * 10^m + X_R$$

Suppose are given two n- digit numbers:

$$Z1 = X_L * 10^m + X_R$$

$$Z2 = Y_L * 10^m + Y_R$$

$$\begin{aligned} Z1 * Z2 &= (X_L * 10^m + X_R)(Y_L * 10^m + Y_R) \\ &= X_L * Y_L * 10^{2m} + (X_L * Y_R + Y_L * X_R) 10^m + X_R Y_R \quad (i) \end{aligned}$$

It is visible from the operations that the product of Z1 and Z2 require four operations, each of which is half in size. There are some additional plus operation which increases to $O(n)$ extra work

If we do these four multiplications recursively, applying the algorithm and terminating at the base conditions, the recurrence relation of integer multiplication can be formulated as:

$$T(n) = 4T(n/2) + O(n)$$

Applying the master method, $T(n) = O(n^2)$.

There is no improvement in time complexity. In 1962, A.A. Karatsuba discovered an asymptotically faster algorithm $O(n^{1.59})$ for multiplying two n -digit numbers using divide & conquer approach.

Karatsuba method (using Divide and Conquer)

In 1962, A.A. Karatsuba discovered a method to compute $X.Y$ (as in Equation(1)) in only 3 multiplications, at the cost of few extra additions; as follows:

Let $U=(a+b).(c+d)$

$V=a.c$

$W=b.d$

Now

$$X.Y = V.10^{2^{[n/2]}} + (U - V - W).10^{[n/2]} + W \dots \dots \dots (2)$$

Now, here, $X.Y$ (as computed in equation (2)) requires only 3 multiplications of size $n/2$, which satisfy the following recurrence:

If $n=1$

$$T(n) = \begin{cases} O(1) \\ 3T\left(\frac{n}{2}\right) + O(n) \end{cases}$$

Otherwise

Where $O(n)$ is the cost of addition, subtraction and digit shift (multiplications by power of 10's), all these take time proportional to 'n'.

Method 1 (Master method)

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

$a=3$

$b=2$

$f(n)=n$

$$n^{\log_b a} = n^{\log_2 3}$$

$$f(n) = n = O(n^{\log_2 3 - \epsilon}) \Rightarrow \text{case 1 of Master Method}$$

$$\Rightarrow T(n) = \Theta(n^{\log_2 3});$$

$$\Rightarrow \Theta(n^{1.59})$$

Method 2 (Substitution Method)

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

$$= 3T\left(\frac{n}{2}\right) + c.n$$

Now

$$T(n) = c.n + 3T\left(\frac{n}{2}\right)$$

$$= c.n + 3\left\{c.\frac{n}{2} + 3T\left(\frac{n}{4}\right)\right\}$$

$$= c.n + \frac{3}{2}c.n + 3^2.T\left(\frac{n}{4}\right)$$

$$= c.n + \frac{3}{2}c.n + 3^2\left\{c.\frac{n}{4} + 3T\left(\frac{n}{8}\right)\right\}$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + 3^3T\left(\frac{n}{8}\right)$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + 3^3\left\{c.\frac{n}{8} + 3T\left(\frac{n}{16}\right)\right\}$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + \left(\frac{3}{2}\right)^3 c.n + \dots + \left(\frac{3}{2}\right)^k .T\left(\frac{n}{2^k}\right)$$

$$= c.n + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + \left(\frac{3}{2}\right)^3 c.n + \dots + \left(\frac{3}{2}\right)^{\log n} .c$$

$$= c.n \left(1 + \frac{3}{2}c.n + \left(\frac{3}{2}\right)^2 c.n + \left(\frac{3}{2}\right)^3 c.n + \dots + \left(\frac{3}{2}\right)^{\log n}\right)$$

$$= c.n \left[\frac{1 \left[\left(\frac{3}{2}\right)^{\log n + 1} - 1 \right]}{\frac{3}{2} - 1} \right]$$

$$= 2c.n \left[\left(\frac{3}{2}\right)^{\log n + 1} - 1 \right]$$

$$= 2c.n \left[\left(\frac{3}{2}\right)^1 \left(\frac{3}{2}\right)^{\log n} - 1 \right]$$

$$= 2c.n \left[\frac{3}{2} n^{\log \frac{3}{2}} - 1 \right]$$

$$= 2c.n \left[\frac{3}{2} n^{\log_2 3 - \log_2 2} - 1 \right]$$

$$\begin{aligned}
&= 2 c . n \left[\frac{3}{2} n^{\log_2 3 - 1} - 1 \right] \\
&= 2 c . n \left[\frac{3 n^{\log_2 3}}{2 n} - 1 \right] \\
&= 3 c . n^{\log_2 3} - 2 c . n \\
&= O(n^{\log_2 3})
\end{aligned}$$

2.6 Matrix Multiplication

- Matrix multiplication is a binary operation of multiplying two or more matrices one by one that are conformable for multiplication. For example two matrices A, B having the dimensions of $p \times q$ and $s \times t$ respectively; would be conformable for $A \times B$ multiplication only if $q=s$ and for $B \times A$ multiplication only if $t=p$.
- Matrix multiplication is associative in the sense that if A, B, and C are three matrices of order $m \times n$, $n \times p$ and $p \times q$ then the matrices $(AB)C$ and $A(BC)$ are defined as $(AB)C = A(BC)$ and the product is an $m \times q$ matrix.
- Matrix multiplication is not commutative. For example two matrices A and B having dimensions $m \times n$ and $n \times p$ then the matrix $AB = BA$ can't be defined. Because BA is not conformable for multiplication, even if AB are conformable for matrix multiplication.
- For three or more matrices, matrix multiplication is associative, yet the number of scalar multiplications may vary significantly depending upon how we pair the matrices and their product matrices to get the final product.

2.6.1 Straight forward method

Let's suppose, we have taken two matrices A and B of size $m \times n = 2 \times 2$ and want to multiply $A \times B$ and store the multiplication in C.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{ij} = \sum_{k=1}^n A_{ik} \times B_{kj}$$

$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$, where, i and j represents the number of rows and columns.

For multiplying both the matrices A and B simple algorithm will be:

```
for (int i = 0; i < N; i++)
```

```

{
    for (int j = 0; j < N; j++)
    {
        C[i][j] = 0;
        for (int k = 0; k < N; k++)
        {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}

```

In the above algorithm accessing the element from 2D matrix definitely required two for loops. And then finding the multiplication for each element of C matrix required one more **for loop**. Thus, in simple approach multiplying any square matrix needed three for loops and complexity will be $O(N^3)$.

2.6.2 Divide & Conquer Strategy for multiplication

In divide and conquer strategy we say that if the problem is large, we break the problem into small problems called sub problems and solve those sub problems and combined solution of sub problems to get the solution of main problem. Now what will be the size of sub- problem? If we have 2×2 matrices we can multiply that without applying divide and conquer but if we have larger size matrices then we need to divide those matrices and solve the smaller matrices to get the final solution.

Now look at how to find the multiplication in C matrix after multiplying A and B.

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$$

$$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$$

$$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$$

$$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$$

To multiply the 2×2 matrices A and B, we don't need to use the three for loops. We can perform multiplication using above formula that takes 8 multiplication and 4 additions. That is a constant time computation and this is defined as a small problem. Now what if matrix size is greater than 2 then we need to divide the matrix and solve the small matrices to get the final multiplication. We assume that matrices having dimension of power of 2 only like 4×4 , 8×8 , 16×16 and so on. Powers of 2 matrices are easy to divide into smaller sizes of 2×2 and if any matrix is not of power of 2 then fill it with zero's to make it power of 2. If we want to multiply the product of two $(n \times n)$ matrices and if n is an exact power of 2 (i.e. $n=2^k$), we divide each of A, B and C Where the result will be stored into $\left[\frac{n}{2} \times \frac{n}{2}\right]$ matrices.

Divide and Conquer Method:

Consider two matrices A and B with 4×4 dimension each as shown below,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

The matrix multiplication of the above two matrices A and B is Matrix C,

$$C = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

where,

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21} + a_{13} * b_{31} + a_{14} * b_{41}$$

$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22} + a_{13} * b_{32} + a_{14} * b_{42}$$

$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21} + a_{23} * b_{31} + a_{24} * b_{41}$$

$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22} + a_{23} * b_{32} + a_{24} * b_{42}$$

Now, let's look at the Divide and Conquer approach to multiply two matrices. Take two sub-matrices from the above two matrices A and B each as (A11 & A12) and (B11 & B21) as shown below,

$$\begin{array}{c}
 \mathbf{A} \qquad \qquad \mathbf{B} \\
 \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array} \right] \quad \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array} \right]
 \end{array}$$

And the matrix multiplication of the two 2x2 matrices A11 and B11 is,

$$\begin{array}{c}
 \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] * \left[\begin{array}{cc} b_{11} & b_{12} \\ b_{21} & b_{22} \end{array} \right] = \left[\begin{array}{cc} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{array} \right] \\
 \mathbf{A11} \qquad \mathbf{B11}
 \end{array}$$

Also, the matrix multiplication of two 2x2 matrices A12 and B21 is as follows,

$$\begin{array}{c}
 \left[\begin{array}{cc} a_{13} & a_{14} \\ a_{23} & a_{24} \end{array} \right] * \left[\begin{array}{cc} b_{31} & b_{32} \\ b_{41} & b_{42} \end{array} \right] = \left[\begin{array}{cc} a_{13} * b_{31} + a_{14} * b_{41} & a_{13} * b_{32} + a_{14} * b_{42} \\ a_{23} * b_{31} + a_{24} * b_{41} & a_{23} * b_{32} + a_{24} * b_{42} \end{array} \right] \\
 \mathbf{A12} \qquad \mathbf{B21}
 \end{array}$$

So if you observe, I can conclude the following,

$$\mathbf{A11 * B11 + A12 * B21} = \begin{bmatrix} c_{11} & c_{12} & \cdot & \cdot \\ c_{21} & c_{22} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Where, '+' is Matrix Addition, and c_{11} , c_{12} , c_{21} and c_{22} are equal to equations 1, 2, 3 and 4 respectively.

So the idea is to recursively divide $n \times n$ matrices into $n/2 \times n/2$ matrices until they are small enough to be multiplied in the naive way, more specifically into 8 multiplications and 4 matrix additions. Now let us complete the pseudocode of the algorithm.

Pseudocode of the Matrix Multiplication Algorithm

1. $n \leftarrow$ no. Of rows of A
2. If $n=1$ then return (a11 b11)
3. Else
4. Let A_{ij} , B_{ij} (for $i,j = 1,2$ be $\left(\frac{n}{2} * \frac{n}{2}\right)$ submatrices)

$$\text{S.t } \mathbf{A} = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \text{ and } \mathbf{B} = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right)$$

5. Recursively compute $A_{11}B_{11}$, $A_{12}B_{21}$, $A_{11}B_{12}$,..... $A_{22}B_{22}$

6. Compute

$$C_{11} = A_{11}B_{11} + A_{12} + B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12} + B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22} + B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22} + B_{22}$$

7. Return $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$

Analysis of Divide and Conquer based Matrix Multiplication

Let $T(n)$ be the no. Of arithmetic operations performed by D&C-MATMUL.

- Line 1,2,3,4,7 require $\Theta(1)$ arithmetic operations.
- Line 5, requires $8T(n/2)$ arithmetic operations.

(i.e. In order to compute AB using e.g. (2), we need 8- multiplications of $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices).

- Line 6 requires $4(n/2) = \theta(n^2)$
(i.e. 4 additions of $\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices)

So the overall computing time, $T(n)$, for the resulting Divide and conquer Matrix Multiplication

$$T(n) = 8T\left(\frac{n}{2}\right) + \theta(n^2)$$

Using Master method, $a=8$, $b=2$ and $f(n)=n^2$; since

$$f(n) = n^2 = O(n^{\log_2 8}) \Rightarrow \text{case 1 of master method}$$

$$\Rightarrow T(n) = Q\left(n^{\log_2 8}\right) = Q(n^3)$$

For multiplying two matrices of size $n \times n$, we make 8 recursive calls above, each on a matrix/sub-problem with size $n/2 \times n/2$. Each of these recursive calls multiplies two $n/2 \times n/2$ matrices, which are then added together. For the addition, we add two matrices of size $\frac{n^2}{4}$, so each addition takes $\Theta\left(\frac{n^2}{4}\right)$ time.

Thus, recurrence relation will be:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

From the Case 1 of Master's Theorem, the time complexity of the above approach is $O(n \log 28)$ or $O(n^3)$ which is the same as the naive method of matrix multiplication.

Strassen's Matrix Multiplication Algorithm:

Greedy Technique

Strassen's algorithm makes use of the same divide and conquer approach as above, but instead uses only 7 recursive calls rather than 8 as shown in the equations below. Here we save one recursive call, but have several new additions and subtractions of $n/2 \times n/2$ matrices.

1. Divide the input matrices A and B into $n/2 \times n/2$ sub-matrices, which takes $\Theta(1)$ time.
2. Now calculate the 7 sub-matrices M1-M7 by using below formulas:

$$M1 = (A11 + A22) (B11 + B22)$$

$$M2 = (A21 + A22) B11$$

$$M3 = A11 (B12 - B22)$$

$$M4 = A22 (B21 - B11)$$

$$M5 = (A11 + A12) B22$$

$$M6 = (A21 - A11) (B11 + B12)$$

$$M7 = (A12 - A22) (B21 + B22)$$

3. To get the desired sub-matrices C11, C12, C21, and C22 of the result matrix C by adding and subtracting various combinations of the M_i sub-matrices. These four sub-matrices can be computed in $\Theta(n^2)$ time.

$$C11 = M1 + M4 - M5 + M7$$

$$C12 = M3 + M5$$

$$C21 = M2 + M4$$

$$C22 = M1 - M2 + M3 + M6$$

Using the above steps, we get the recurrence of the following format:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1. \end{cases}$$

Using master method: $a=7$, $b=2$ and $n^{\log b^a} = n^{\log 2^7} = n^{2.81}$ $f(n) = n^2$

$f(n) = n^2 = O(n^{\log 2 - \epsilon}) \Rightarrow$ case 1 of master method

$\Rightarrow T(n) = Q(n^{\log b^a}) = Q(n^{\log 2^7}) = Q(n^{2.81})$.

Ex:- To perform the multiplication of A and B

$$AB = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 3 \\ 4 & 1 & 1 & 2 \\ 0 & 3 & 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & 4 & 2 & 7 \\ 3 & 1 & 3 & 5 \\ 2 & 0 & 1 & 3 \\ 1 & 4 & 5 & 1 \end{bmatrix}$$

We define the following eight $n/2$ by $n/2$ matrices:

$$\begin{aligned} A_{11} &= \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} & A_{12} &= \begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix} & B_{11} &= \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} & B_{12} &= \begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix} \\ A_{21} &= \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} & A_{22} &= \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} & B_{21} &= \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix} & B_{22} &= \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix} \end{aligned}$$

Strassen showed how the matrix C can be computed using only 7 block multiplications and 18 block additions or subtractions (12 additions and 6 subtractions):

$$p_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$p_2 = (A_{21} + A_{22})B_{11}$$

$$p_3 = A_{11}(B_{12} - B_{22})$$

$$p_4 = A_{22}(B_{21} - B_{11})$$

$$p_5 = (A_{11} + A_{12}) \cdot B_{22}$$

$$p_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$p_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

The correctness of the above equations is easily verified by substitution.

$$p_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$$

$$= \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} * \begin{bmatrix} 2 & 7 \\ 8 & 2 \end{bmatrix} = \begin{bmatrix} 36 & 22 \\ 58 & 47 \end{bmatrix}$$

$$P_2 = (A_{21} + A_{22}) * B_{11} = \begin{bmatrix} 14 & 23 \\ 14 & 23 \end{bmatrix}$$

$$P_3 = A_{11} * (B_{12} - B_{22}) = \begin{bmatrix} -3 & 12 \\ -12 & 24 \end{bmatrix}$$

$$P_4 = A_{22} * (B_{21} - B_{11}) = \begin{bmatrix} -3 & 2 \\ 5 & -20 \end{bmatrix}$$

$$P_5 = (A_{11} * A_{12}) * (B_{11} + B_{12}) = \begin{bmatrix} 34 & 18 \\ 45 & 9 \end{bmatrix}$$

$$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12}) = \begin{bmatrix} 3 & 27 \\ -18 & -18 \end{bmatrix}$$

$$P_7 = (A_{12} - A_{11}) * (B_{21} + B_{22}) = \begin{bmatrix} 18 & 16 \\ 3 & 0 \end{bmatrix}$$

$$C_{11} = P_{21} + P_4 - P_5 + P_7 = \begin{bmatrix} 17 & 22 \\ 21 & 18 \end{bmatrix}$$

$$C_{12} = P_3 + P_5 = \begin{bmatrix} 31 & 30 \\ 33 & 33 \end{bmatrix}$$

$$C_{21} = P_2 + P_4 = \begin{bmatrix} 11 & 25 \\ 14 & 3 \end{bmatrix}$$

$$C_{22} = P_1 + P_3 - P_2 + P_6 = \begin{bmatrix} 22 & 38 \\ 14 & 30 \end{bmatrix}$$

$$C = \begin{bmatrix} 17 & 22 & 31 & 30 \\ 21 & 18 & 33 & 33 \\ 11 & 25 & 22 & 38 \\ 19 & 3 & 14 & 30 \end{bmatrix}$$

The overall time complexity of Strassen's Method can be written as:

$$T(n) = \begin{cases} 0(1) & \text{if } n = 1 \\ 7T\left(\frac{n}{2}\right) + 0(n^2) \end{cases}$$

Otherwise

$$a = 7; b = 2; f(n) = n^2$$

$$n^{\log_a b} = n^{\log_2 7} = n^{2.81}$$

$$f(n)n^2 = 0\left(n^{\log_2 7 - \epsilon}\right) \Rightarrow \text{case 1 of master method}$$

$$\Rightarrow T(n) = Q\left(n^{\log_2 7}\right)$$

$$= Q(n^{2.81})$$

$$\text{The solution of this recurrence is } T(n) = O\left(n^{\log_2 7}\right) = O(n^{2.81})$$

Generally Strassen's Method is not preferred for practical applications for following reasons:

- 1) The constants used in Strassen's method are high and for a typical application Naive method works better.
- 2) For Sparse matrices, there are better methods especially designed for them.

- 3) The sub matrices in recursion take extra space.
- 4) Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen's algorithm than in Naive Method.

Check your Progress 3:

Question 1: Which designing approach is used in Strassen's matrix multiplication algorithm?

Question 2: What is the time taken by Strassen's algorithm to perform matrix multiplication?

Question 3: In Strassen's matrix multiplication algorithm ($C = AB$), the matrix C can be computed using only 7 block multiplications and 18 block additions or subtractions. How many additions and how many subtractions are there out of 18?

Question 4: Use Strassen's matrix multiplication algorithm to multiply the following two matrices:

$$P = \begin{bmatrix} 5 & 3 \\ 6 & 7 \end{bmatrix}$$

$$Q = \begin{bmatrix} 1 & 4 \\ 5 & 9 \end{bmatrix}$$

2.7 Summary

- **Divide and Conquer** approach follows a recursive approach which making a recursive call to itself until a base (or boundary) condition of a problem is not reached but with reduced problem size.
- **Divide and Conquer** is a top-down approach, which consists of three steps:

Divide: the given problem is break down into smaller parts.

Conquer: Solve each sub-problem by recursively calling them.

Combine: each sub-solution is combined to generate solution to the original problem.

- Divide-and-conquer approach is widely applicable on problems like Binary search, Quick-Sort, Merge-Sort, Matrix multiplication.
- Binary search is the procedure of finding the location of an element in a sorted array by dividing the array into two halves recursively.
- Merge-Sort algorithm is a divide-and-conquer based sorting algorithm which recursively partitions an array into several sub-arrays until each sub-array consists of single element. Then, each sub-array is merged

- ## 2.8 Solution to Check Your Progress

Solution 1: 147.1 to 148.1

Check Your Progress 2

Solution 1: 2, 4, 5, 6, 7

Solution 2:

- Solution 3:** Considering pivot element as 7, following sequence of steps will be followed.

- Therefore, the output is: 6 3 4 7 9 14 11 12

Solution 4: 31.11 sec Approximately. The Quick sort requires $O(n \log n)$ comparisons in best case, where 'n' is size of input array. So, $1000 * \log 1000 \approx 9000$ comparisons are required to sort 1000 elements, which takes 200 sec. To sort 200 elements minimum of $200 * \log 200 \approx 1400$ comparisons are required. This will take $200 * 1400 / 9000 \approx 31.11$ sec.

Check Your Progress 3

Solution 1: Divide and Conquer

Solution 2: $O(n^{2.81})$

Solution 3: 12 and 6

Solution 4: $R = \begin{bmatrix} 23 & 52 \\ 48 & 93 \end{bmatrix}$

2.9 Further Readings

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, Introduction to Algorithms, MIT Press, 3rd Edition, 2009.
2. Steven Skiena, The Algorithm Design Manual, Springer; 2nd edition, 2008.
3. Knuth, The art of Computer Programming Volume 1, Fundamental Algorithms, Addison-Wesley Professional; 3 edition, 1997.
4. Horowitz and Sahni, Fundamentals of Computer Algorithms, Computer Science Press, 2008.
5. Sedgewick, Algorithms in C, 3rd edition. Addison Wesley, 2002.