UNIT 2 DIVIDE AND CONQUER APPROACH

Structure

2.0 INTRODUCTION

We have already mentioned in unit-1 of Block-1 that there are five fundamental techniques which are used to design the Algorithm efficiently. These are: Divide and Conquer, Greedy Method, Dynamic Programming, Backtracking and Branch and Bound. Out of these techniques Divide & Conquer is probably the most well-known.

Many useful algorithms are recursive in nature. To solve a given problem, they call themselves recursively one or more times. These algorithms typically follow a divide & Conquer approach. A divide & Conquer method works by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough (i.e. smaller in size w.r.t. original problem) to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The following Figure-2.1 show a typical Divide & Conquer Approach

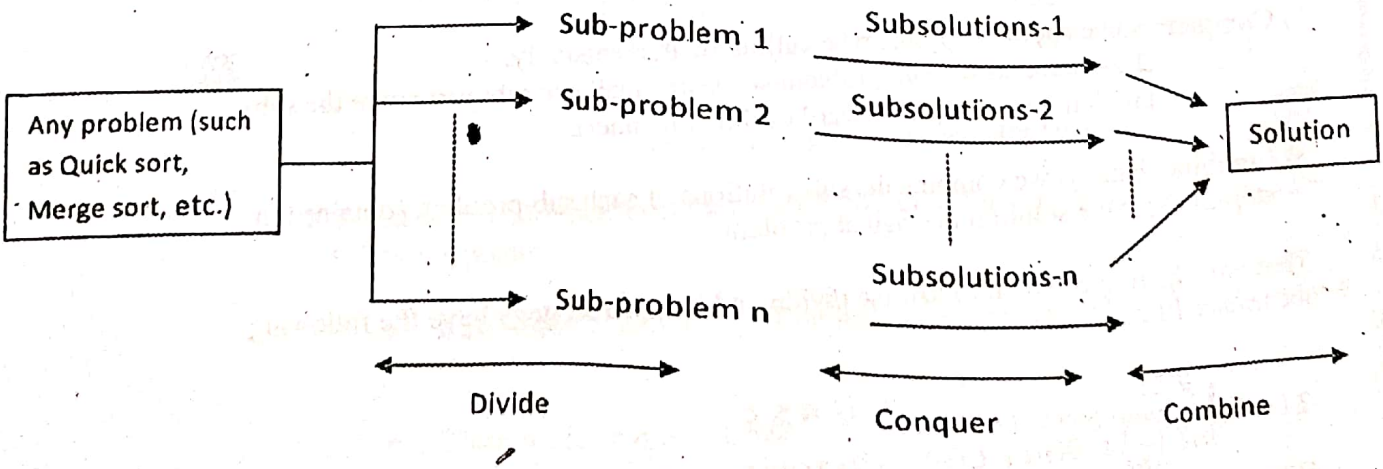


Figure 2.1: Steps in a divide and computer technique

Thus, in general, a divide and Conquer technique involves 3 Steps at each level of recursion:

Step 1: Divide the given big problem into a number of sub-problems that are similar to the original problem but smaller in size. A sub-problem may be further divided into its sub-problems. A Boundary stage arrives when either a direct solution of a sub-problem at some stage is available or it is not further sub-divided. When no further sub-division is possible, we have a direct solution for the sub-problem.

Step 2: Conquer (Solve) each solutions of each sub-problem (independently) by recursive calls; and then

Step 3: Combine the solutions of each sub-problems to generate the solutions of original problem.

In this unit we will solve the problems such as Binary Search, Searching - QuickSort, MergeSort, integer multiplication etc by using Divide and Conquer method;

2.1 OBJECTIVES

After going through this unit, you will be able to:

- understand the basic concept of Divide-and-Conquer;
- explain how Divide-and-Conquer method is applied to solve various problems such as Binary Search, Quick-Sort, Merge-Sort, Integer multiplication etc., and
- write a general recurrence for problems that is solved by Divide-and-Conquer.

2.2 GENERAL ISSUES IN DIVIDE AND CONQUER

Many useful algorithms are recursive in structure, they make a recursive call to itself until a base (or boundary) condition of a problem is not reached. These algorithms closely follow the **Divide and Conquer** approach.

To analyzing the running time of divide-and-conquer algorithms, we use a **recurrence equation** (more commonly, a **recurrence**). A recurrence for the running time of a divide-and-conquer algorithm is based on the 3 steps of the basic paradigm.

1) **Divide**: The given problem is divided into a number of sub-problems.

2) **Conquer**: Solve each sub-problem be calling them recursively.
(**Base case:** If the sub-problem sizes are small enough, just solve the sub-problem in a straight forward or direct manner).

3) **Combine**: Finally, we combine the sub-solutions of each sub-problem (obtained in step-2) to get the solution to original problem.

Thus any algorithms which follow the divide-and-conquer strategy have the following recurrence form:

$$T(n) = \begin{cases} \theta(1) & \text{if } n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{Otherwise} \end{cases}$$

Where

- $T(n) = \text{running time of a problem of size } n$
- If the problem size is small enough (say, $n \le c$ for some constant c), we have a base case. The brute-force (or direct) solution takes constant time: $\Theta(1)$
- Otherwise, suppose that we divide into a sub-problems, each $1/b$ of the size of the original problem of size n.

- Suppose each sub-problem of size **n/b** takes – time to solve and since there are *a* sub-problems so we spend – total time to solve sub-problems.
- is the cost(or time) of dividing the problem of size n.
- is the cost (or time) to combine the sub-solutions.

Thus in general, an algorithm which follows the divide and conquer strategy have the following recurrence:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where

- T(n) = running time of a problem of size n
- a means "In how many part the problem is divided"
- – means "Time required to solve a sub-problem each of size (n/b)"
- D(n) + C(n) = f(n) is the summation of the time requires to divide the problem and combine the sub-solutions.

  (Note: For some problem C(n)=0, such as Quick Sort)

Example: Merge Sort algorithm closely follows the (Figure 2.2) Divide-and-Conquer approach. The following procedure MERGE_SORT (A, p, r) sorts the elements in the subarray A [p ,..., r]. If $p \geq r$, the subarray has at most one element and is therefore already sorted. Otherwise, the Divide step is simply computer an index q that partitions A [p, ... , r] into two sub-arrays: A [p, ..., q] containing ceil (n/2) elements, and A[q+1, ..., r] containing floor (n/2) elements.

```
MERGE_SORT (A, p, r)

1. if (p < r)
2.   then q ← [(p + r)/2]          /* Divide
3.     MERGE_SORT (A, r, q)        /* Conquer
4.     MERGE_SORT (A, q + 1, r)    /* Conquer
5.     MERGE (A, p, q, r)          /* Combine
```

Figure 2.2: Steps in merge sort algorithms

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

- **Base Case:** MERGE SORT on just one element (n=1) takes constant time i.e.
- When we have n > 1 elements, we can find a running time as follows:

  (1) **Divide:** Just compute q as the middle of p and r, which takes constant time. Thus

(2) **Conquer**: We recursively solve two sub-problems, each of size n/2, which contributes

$$2T\left(\frac{n}{2}\right)$$

to the running time.

(3) **Combine**: Merging two sorted subarrays (for which we use MERGE (A, p, r) of an n-element array) takes time , so
Thus , which is a linear function of n.

Thus from all the above 3 steps, a recurrence relation for MERGE_SORT (A, 1, n) in the **worst case** can be written as:

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

Now after solving this recurrence by using any method such as Recursion-tree or Master Method (as given in UNIT-1), we have
This algorithms will be explained in detailed in section 2.4.2

2.3 BINARY SEARCH

Search is the process of finding the position (or location) of a given element (say x) in the linear array. The search is said to be successful if the given element is found in the array otherwise it is considered unsuccessful.

A Binary search algorithm (Figure 2.3) is a technique for finding a position of specified value (say x) within a **sorted array** A. the best example of binary search is "dictionary", which we are using in our daily life to find the meaning of any word. The Binary search algorithm proceeds as follow:

(1) Begin with the interval covering the whole array; binary search repeatedly divides the search interval by half.

(2) At each step, the algorithm compares the input key (or search) value x with the key value of the middle element of the array A.

(3) If it matches, then a searching element x has been found, so then its index, or position, is returned. Otherwise, if the value of the search element x is less than the item in the middle of the interval; then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search element x is greater than the middle element's key, then on the sub-array to the right.

(4) We repeatedly check until the searched element is found or the interval is empty, which indicates x is "not found".

```
BinarySearch_Iterative(A[1…n],n,x)

/* Input: A sorted (ascending) linear array of size n.

Output: This algorithm find the location of the search element x in linear array A. If
search ends in success, it returns the index of the searched element x, otherwise returns
-1 indicating x is "not found". Here variable low and high is used to keep track of the
first element and last element of the array to be searched, and variable mid is used as
index of the middle element of the array under consideration. */
{
    low=1
    high=n
    while(low<=high)
      {
         mid= (low+high)/2
         if(A[mid]==x]
            return mid;                    // x is found
        else if(x<A[mid])
            high=mid-1;
        else  low=mid+1;
      }
     return -1                             // x is not found
}
```

Figure 2.3: Binary search algorithms

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 k^{th} .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 $C.(\log n + 1) = O(\log n)$ comparisions. Thus $T(n) = O(\log n)$

Method 2:

Binary search closely follow the Divide-and-conquer technique.
We know that any problem, which is solved by using Divide-and-Conquer having a recurrence of the form: $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Since at each iteration, the array is divided into two sub-arrays but we are solving only one sub-array in the next iteration. So value of a=1 and b=2 and f(n)=k where k is a constant less than n.
Thus a recurrence for a binary search can be written as

$T(n) = T\left(\frac{n}{2}\right) + k$; by solving this recurrence using substitution method, we have:

$k + k + k + \cdots \ldots \ldots \ldots up\ to\ (logn) terms = k.logn = O(logn)$

Example1: consider the following sorted array DATA with 13 elements:

11	22	30	33	40	44	55	60	66	77	80	88	99

Illustrate the working of binary search technique, while searching an element (say ITEM)

(i) 40 (ii) 85

Solution

We apply the binary search to DATA[1,...13] for different values of ITEM.
(Figure 2.4)

(a) Suppose ITEM = 40. The search for ITEM in the array DATA is pictured in Fig.1, where the values of DATA[Low] and DATA[High] in each stage of the algorithm are indicated by circles and the value of DATA[MID] by a square. Specifically, Low, High and MID will have the following successive values:

1. Initially, Low = 1 and High = 13, Hence
   MID = so DATA
2. Since 40 < 55, High has its value changed by High = MID – 1 = 6.
   Hence MID = so DATA
3. Since 40 < 30, Low has its value changed by Low = MID – 1 = 4.
   Hence MID = so DATA

We have found ITEM in location LOC = MID = 5.

(1) (11,) 22, 30, 33, 40, 44, [55,] 60, 66, 77, 80, 88, (99,)

(2). (11,) 22, (30,) 33, 40, [44,] 55, 60, 66, 77, 80, 88, 99

(3) 11, 22, 30, (33,) [40,] [44,] 55, 60, 66, 77, 80, 88, 99 [Successful]

Figure 2.4: Binary search for ITEM = 40

(b) Suppose ITEM = 85. The binary search for ITEM is pictured in Figure 2.5. Here Low, High and MID will have the following successive values:

1. Again initially, Low = 1, High = 13, MID = 7 and DATA[MID] = 55.

2. Since 85 > 55, Low has its value changed by Low = MID + 1 = 8. Hence
   MID = ⌊(8 + 13)/2⌋ = 10 and so DATA[MID] = 77

3. Since 85 > 77, Low has its value changed by Low = MID + 1 = 11. Hence
   MID = ⌊(11 + 13)/2⌋ = 12 and so DATA[MID] = 88

4. Since 85 > 88, High has its value changed by High = MID − 1 = 11. Hence
   MID = ⌊(11 + 11)/2⌋ = 11 and so
   DATA[MID] = 80

   (Observe that now Low = High = MID = 11.)

Since 85 > 80, Low has its value changed by Low = MID + 1 = 12. But now Low > High, Hence ITEM does not belong to DATA.

(1) (11,) 22, 30, 33, 40, 44 [55,] 60, 66, 77, 80, 88 (99.)

(2) 11, 22, 30, 33, 40, 44, 55, (60,) 66, [77,] 80, 88, (99)

(3) 11, 22, 30, 33, 40, 44, 55, 60, 66, 77, (80) 88, 99 [unsuccessful]

Figure 2.5: Binary search for ITEM = 85

Example 2: Suppose an array DATA contains 1000000 elements. How many comparisons are required (in worst case) to search an element (say ITEM) using Binary search algorithm.

Solution: Observe that
$2^{10} = 1024 > 1000$ and hence $2^{20} > 1000^2 = 1000000$

Using the binary search algorithm, one requires only about **20 comparisons** to find the location of an ITEM in an array DATA with 1000000 elements, since $log_2 1000000 \approx log_2 2^{20} = 20$

☞ Check Your Progress 1

(Objective questions)

1. What are the three sequential steps of divide-and-conquer algorithms?
   (a) Combine-Conquer-Divide
   (b) Divide-Combine-Conquer
   (c) Divide-Conquer-Combine
   (d) Conquer-Divide-Conquer

2. Binary search executes in _____ time.
   (a) $O(n)$ (b) $O(\log n)$ (c) $O(n \log n)$ (d) $O(n^2)$

The following Algorithm merge the two sorted subarray A [p.. r] and A [q+1..r] into one sorted output subarray A [p..r].

```
Merge (A, p, q, r)
1. n1 ← q – p + 1                    // No. of elements in sorted subarray A [p..q]
2. n2 ← r – q                        // No. of elements in sorted subarray A[q+1 .. r]
   Create arrays L [1.. n1 +1] and R [1..n1 +1]
3. for i ← 1 to n1
4. do L[i] ← A [p + I -1]            // copy all the elements of A [p..r] into L [1 .. n1]
5. for j ← 1 to n1
6.       do R [j] ← A [q + j]        // copy all the elements of A [q + 1, .. r] into
                                        R[1..n2]
7. L [n1 +1] ← ∞

8. R [n2 + 1] ← ∞
9. i ← 1
10. j ← 1
11. for k ← p to r
12.       do if L [i] ≤ R [j]
13.                then A [k] ← L [i]
14.                     i ← i +1
15.                else A [k] ← R [j]
16.                     j ← j + 1
```

// The line 11-16 is used to merge the two sorted subarray L [1.. n_2] and R [1]..n_2 back to the Array A [p..r], using the idea linear time merging.

To understand both the algorithm Merge-Sort (A, p, r) and MERGE (A, p, q, r); consider a list of (7) elements (Figure 2.9):

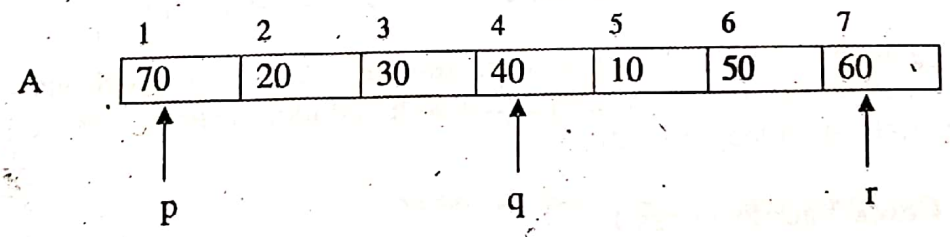


Figure 2.9: Merging algorithms

$$q \leftarrow \left\lfloor \frac{1+7}{2} \right\rfloor = 4$$

Then we will first make two sub-lists as:
MERGER-SORT (A, p, r) → MERGE-SORT (A, 1, 4) (Figure 2.10)
MERGE-SORT (A, q +1, r) → MERGE-SORT (A, 5, 7)

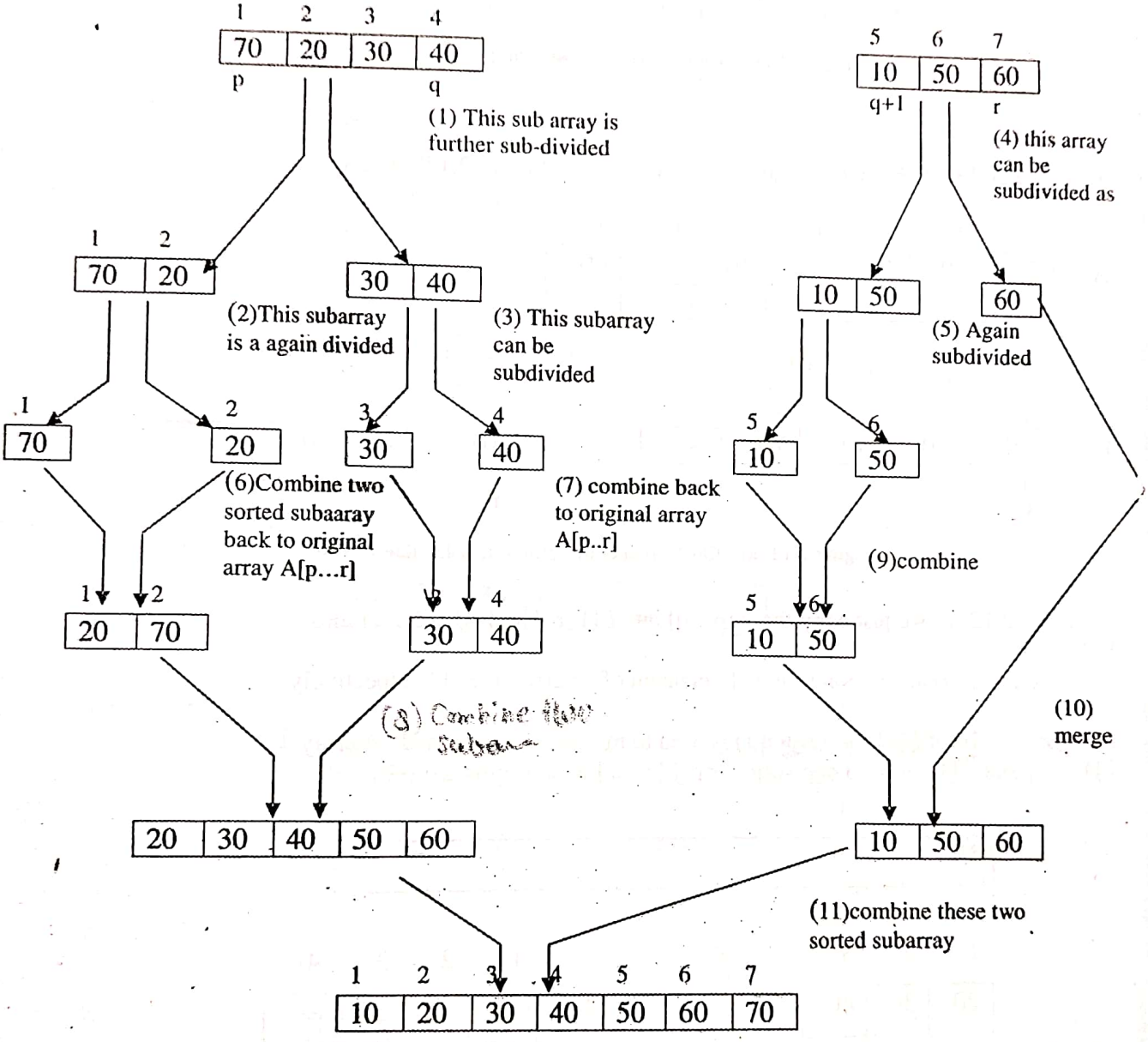


Figure 2.10: Illustration of merging process - 1

Lets us see the MERGE operation more closely with the help of some example. Consider that at some instance we have got two sorted subarray in A, which we have to merge in one sorted subarray (Figure 2.11).

Ex:

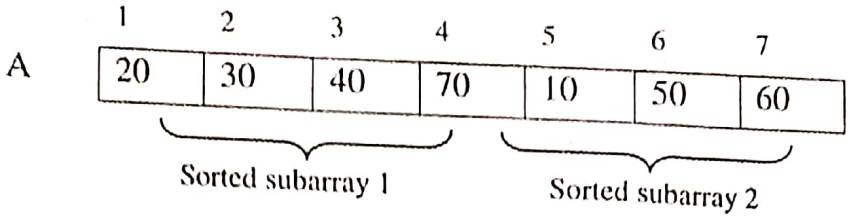
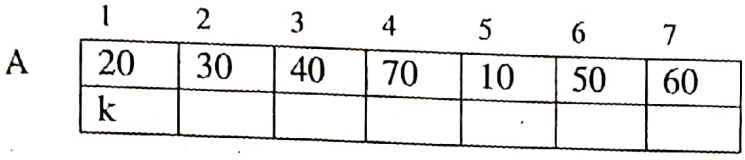


Figure 2.11: Example array for merging

Now we call MERGE (A,1,4,7), after line 1 to line 10 (Figure 2.12), we have



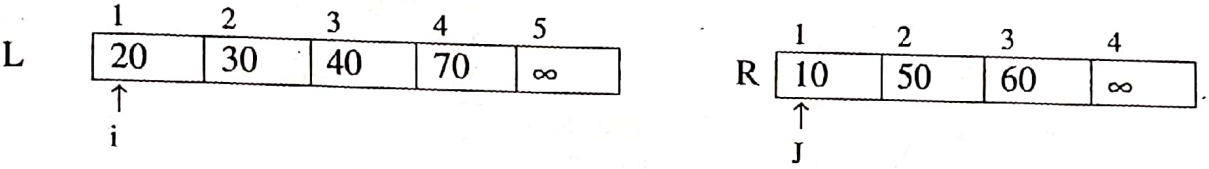
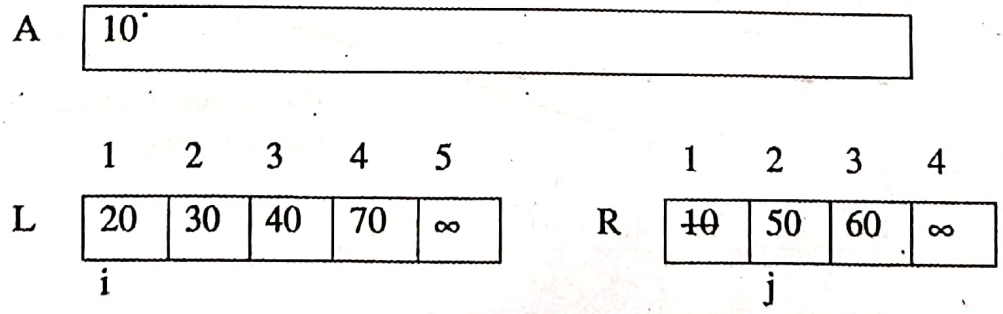


Figure 2.12 (a) : Illustration of merging – II using line 1 to 10

In Figure 2.12(a), we just copy the A[p…q] into L[1..n_1] and A[q+1,…r] into R[1…n_2]
Variable I and j both are pointing to 1st element of an array L & R, respectively.

Now line 11-16 of MERGE (A,p,q,r) is used to merge the two sorted subarray L[1,…4] and R[1…4] into one sorted array [1…7]; (see Figure 2.1 b-h)



(b)

l

10	20

	1	2	3	4	5
L	~~20~~	30	40	70	∞
		i			

	1	2	3	4
R	~~10~~	50	60	∞
		j		

(c)

	1	2	3
A	10	20	30

	1	2	3	4	5
L	~~20~~	~~30~~	40	70	∞
			i		

	1	2	3	4
R	~~10~~	50	60	∞
			j	

(d)

	1	2	3	4
	10	20	30	40

	1	2	3	4	5
L	~~20~~	~~30~~	~~40~~	70	∞
				i	

	1	2	3	4
R	~~10~~	50	60	∞
				j

(e)

	1	2	3	4	5	6	7
	10	20	30	40	50		

	1	2	3	4	5
L	~~20~~	~~30~~	40	70	∞
				i	

	1	2	3	4
R	~~10~~	~~50~~	60	∞
				j

(f)

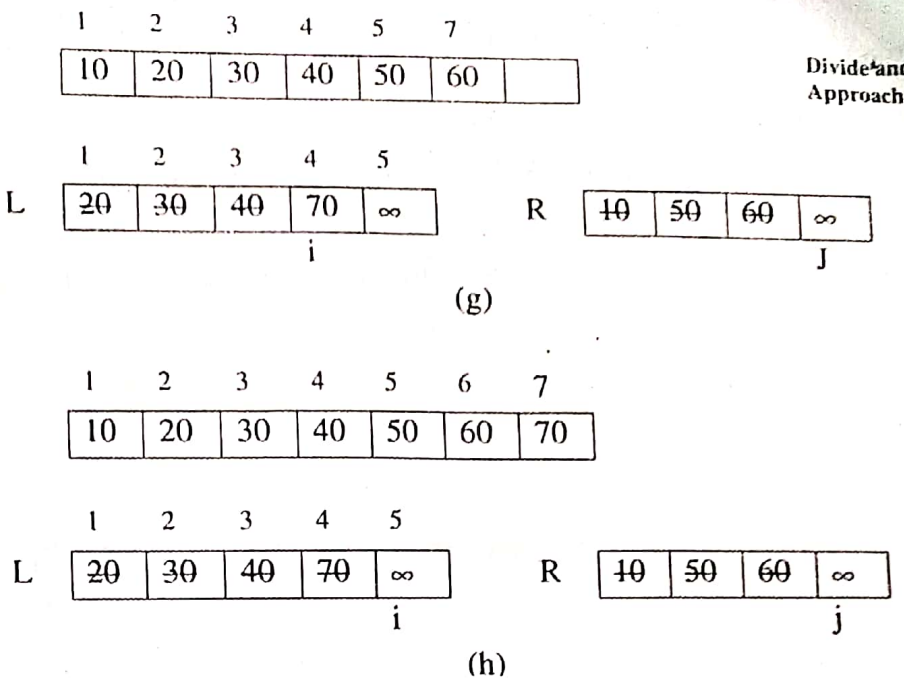


Figure 2.12 b - h: Illustration of merging – III using line 11 to 16

Analysis of MERGE-SORT Algorithm

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

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

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

  Combine: MERGE an n-element subarray takes O(n) time ⇒ 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:

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

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

(1) Master method or
(2) by Recursion tree method:

1) Master Method:

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

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

We have: a = 2

b = 2

f (n) = n

$n^{\log_b a} = n^{\log_2 2} = n$; Now compare f(n) with $n^{\log_2 2 + \epsilon}$ ($n^{\log_2 2} = n$)

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

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

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

2. Method 2: Recursion Tree Method

We rewrite the recurrence as:

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

Recursion tree:

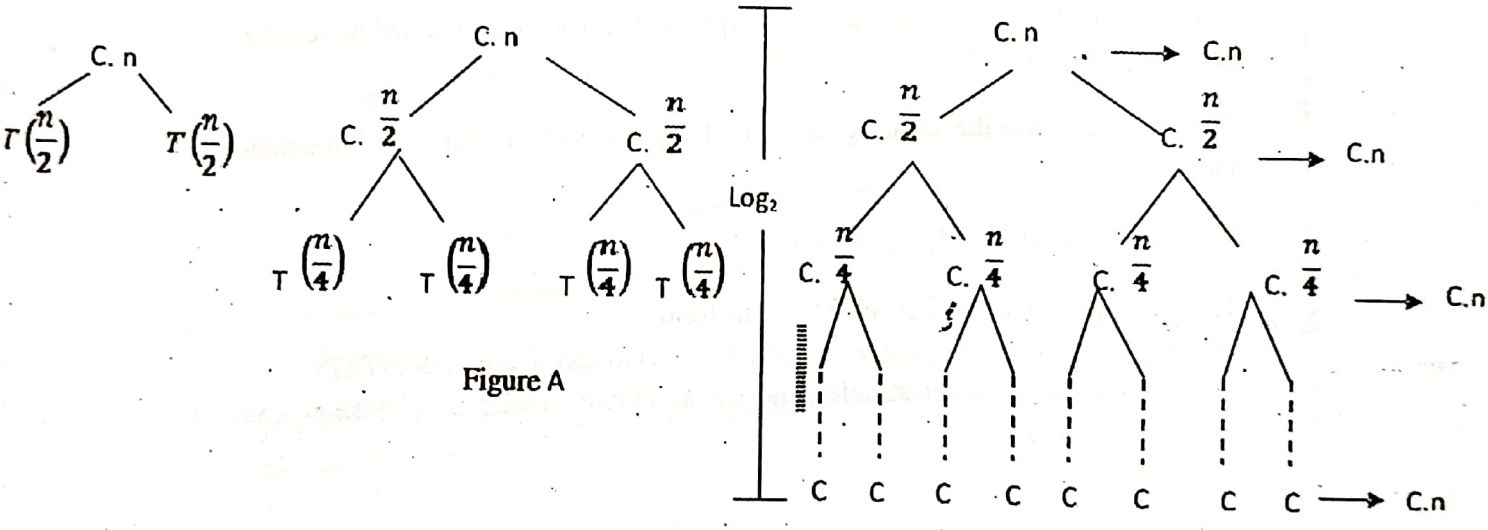


Figure A

Figure B

Total = C.n + C.n+ ..... ($\log_2 n + 1$) terms

= C.n (logn+1)

= $\theta(n \log n)$

2.4.2 QUICK-SORT

Quick-Sort, as its name implies, is the fastest known sorting algorithm in practice. The running time of Quick-Sort depends on the nature of its input data it receives for sorting. If the input data is already sorted, then this is the worst case for quick sort. In this case, its running time is $O(n^2)$. Inspite of this slow worst case running time,

Quick sort is often the best practical of this choice for sorting because it is remarkably efficient on the average; its expected running time in θ (nlogn).

1) Worst Case (when input array is already sorted):

2) Best Case (when input data is not sorted):

3) Average Case (when input data is not sorted & Partition of array is not unbalance as worst case) :

Avanta: - Quick Sort algorithm has the advantage of "Sorts in place"

Quick Sort

The Quick sort algorithm (like merge sort) closely follows the Divide-and Conquer strategy. Here Divide-and-Conquer Strategy involves 3 steps to sort a given subarray A[p..r].

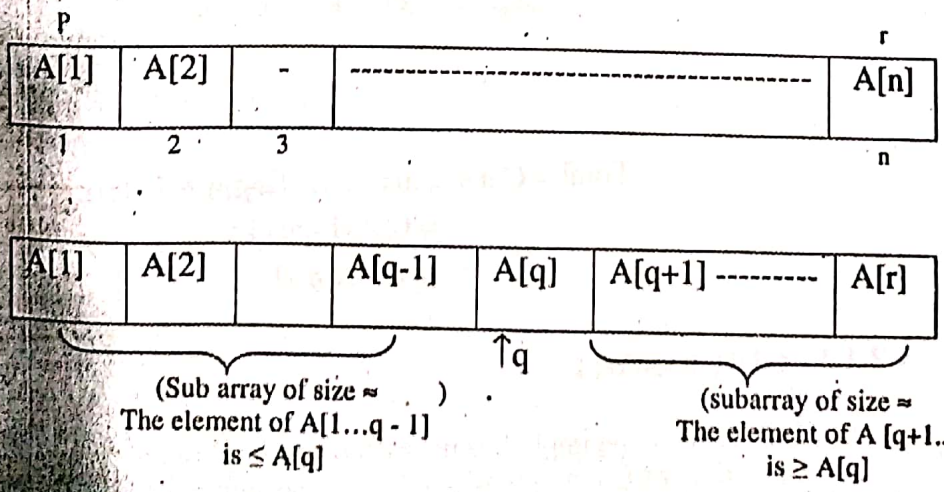
1) **Divide**: The array A [p. . r] is partitioned (rearranged) into two (possibility empty) sub-array A [p..q-1] and A [q+1,..r], such that each element in the left subarray A[p...q-1] is ≤ A [q] and A[q] is ≤ each element in the right subarray A[q+1...r]. to perform this Divide step, we use a PARTITION procedure; which returns the index q, where the array gets partitioned.

2) **Conquer**: These two subarray A [p...q-1] and A [q+1..r] are sorted by recursive calls to QUICKSORT.

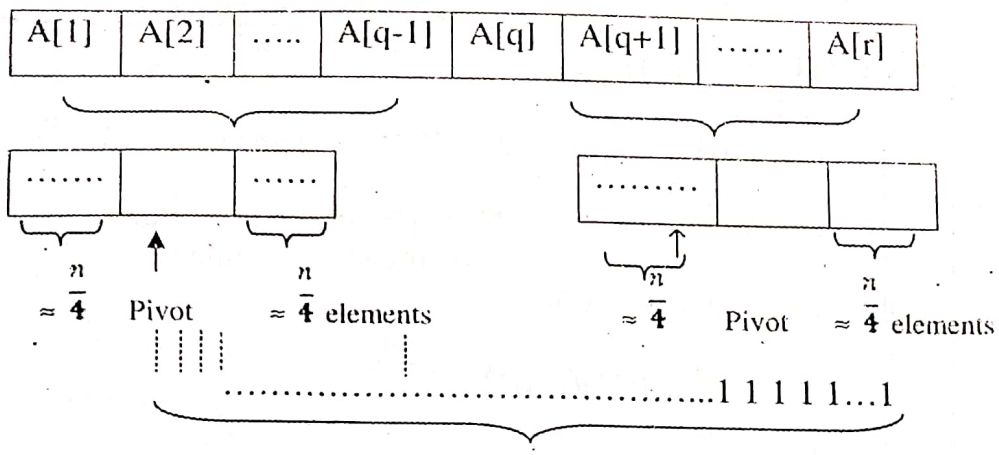
3) **Combine**: Since the subarrays are sorted in place, so there is no need to combine the subarrays.

Now, the entire array A{p..r} is sorted.

The basic concept behind Quick-Sort is as follows:
Suppose we have an unsorted input data A [p...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 follows:



These two subarray A [p...q-1] and A[q+1..r] is further divided by recursive call to QUICK-SORT and the process is repeated till we are left with only one-element in each sub-array (or no further division is possible).



(size of each subarray is now 1) Sorted Array A[p…r]

Pseudo-Code for QUICKSORT:

```
QUICK SORT (A, p, r)
{ If (p < r)                    /* Base Condition
  {
    q← PARTITION (A, p, r)      /* Divide Step*/
    QUICKSORT (A, p, q-1)       /* Conquer
    QUICKSORT (A, q+1, r)       /* Conquer
  }
}
```

- 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] ≤ A[q] and A[q] ≤ A[q+1…r].

Pseudo code for PARTITION:

```
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:          if (A[j] ≤ A [r])
                {
5:              i ← i + 1
6:              Exchange (A[i] ↔ A[j])
                }
            }/* end for
7:      Exchange (A [i + 1] and A[r])
8:      return ( i+ 1)
        }
```

The running time of PARTITION procedure is θ (n), since for an array A[a…n], the loop at line 3 is running 0(n) time and other lines at code take constant time i.e. 0(1) so overall time is 0(n).

To illustrate the operational PARTITION procedure, consider the 8-element array:

	1	2	3	4	5	6	7	8
A	2	8	7	1	3	5	6	4

Step 1: The input array with initial value of I, j, p and r.

x ← A[r] = 4
I ← p-1 = 0

	p						r
	2	8	7	1	3	5	6	4
i	1	2	3	4	5	6	7	8

Step 2: The array A after executing the line 3 – 6

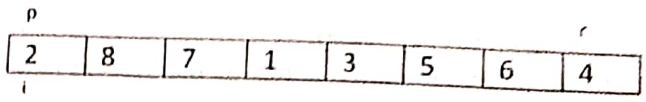
a)

J = 1 to 7
1) j = 1; if A[1] ≤ a i.e. 2 ≤ A[r] ⇒ YES
Therefore i ← i + 1 = 0 + 1 = 1
exchange (A[1] ↔ A[1])

	p							r
	2	8	7	1	3	5	6	4
	i							

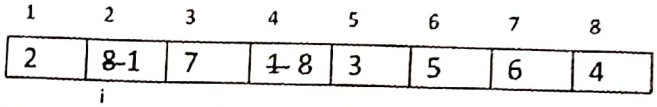
b)

2) j = 2; if A[2] ≤ 4 i.e 8 ≤ 4 ⇒ No
So line 5 – 6, will not be executed
Thus:



c)

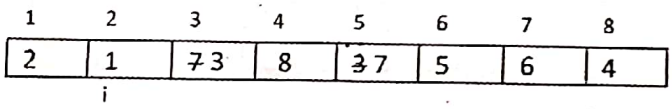
3) j = 3; if A[3] ≤ 4 i.e 7 ≤ 4 ⇒ No; so line 5-6 will not execute
4) j = 4; if A[4] ≤ 4 i.e 1 ≤ 4 ⇒ YES
So i ← i + 1 = 1 + 1 = 2
exchange (A[2] ↔ A[4])



d)

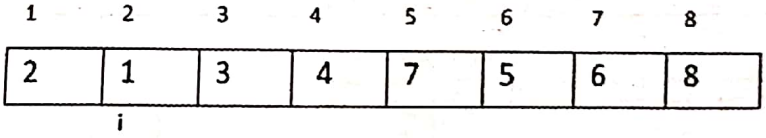
5) j = 5; A[5] ≤ 4 i.e 3 ≤ 4 ⇒ YES
- i ← i + 1 = 2 + 1 = 3

exchange (A [3] ↔ A [5])

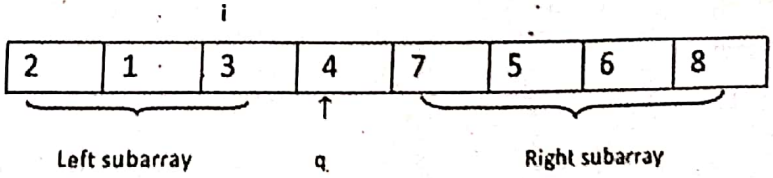


e)

6) j = 6; A[6] ≤ 4 i.e 5≤ 4 ⇒ NO
7) j = 7: A [7] ≤ 4 i.e 6 ≤ 4 ⇒ NO
Now for loop is now finished; so finally line 7 is execute i.e
exchange (A[4] ↔ A[8]), so finally we get:



Finally we return (i + 1) i.e (3 + 1) = 4; by this partition procedure;
Now we can easily see that all the elements of A[1, ... 3] ≤ A[4]; and all the elements of A[5, ..8] ≥ A[4]. Thus



To sort the entire Array A{1..8}; there is a Recursive calls to QuickSort on both the subarray A[..3] and A[5...8].

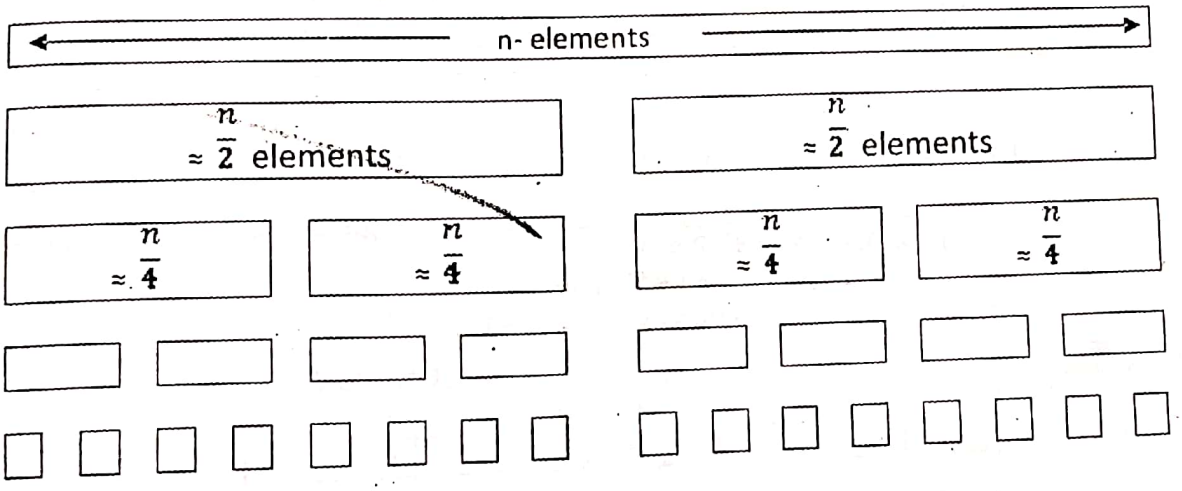
Performance of Quick Sort

The running time of QUICK SORT depends on whether the partitioning is balanced or unbalanced. Partitioning of the subarrays depends on the input data we receive for sorting.

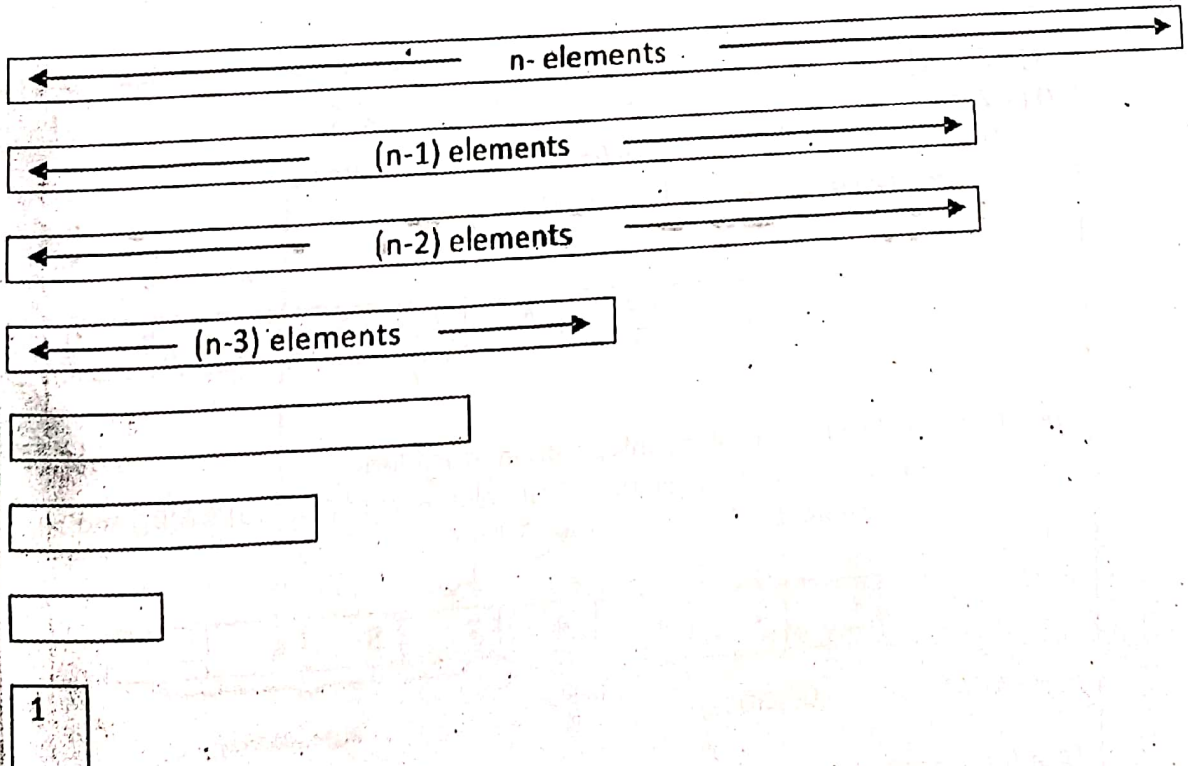
Best Case: If the input data is not sorted, then the partitioning of subarray is balanced; in this case the algorithm runs asymptotically as fast as merge-sort (i.e. 0(nlogn)).

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. θ (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:



(a) Best Case



(b)

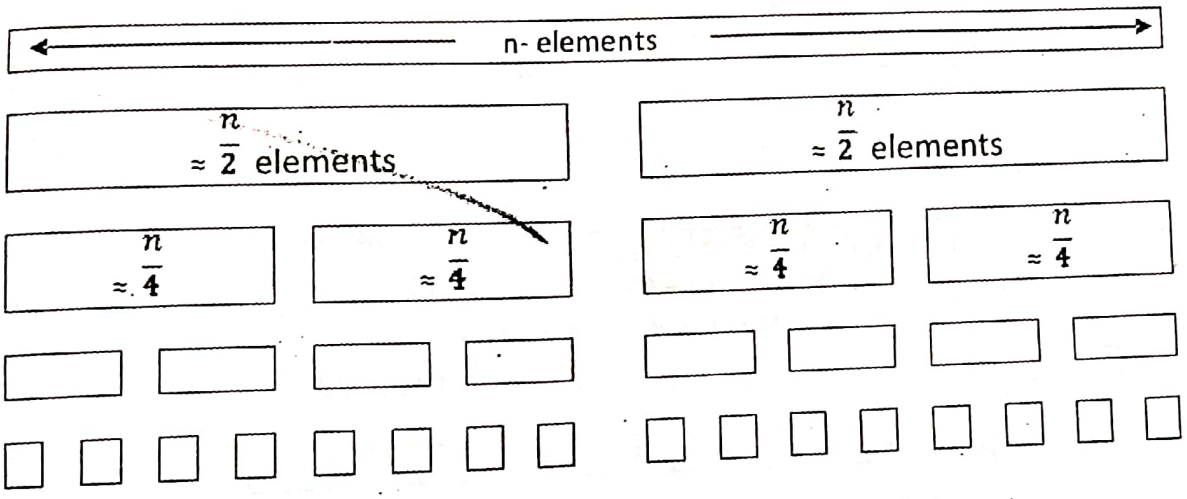
Performance of Quick Sort

The running time of QUICK SORT depends on whether the partitioning is balanced or unbalanced. Partitioning of the subarrays depends on the input data we receive for sorting.

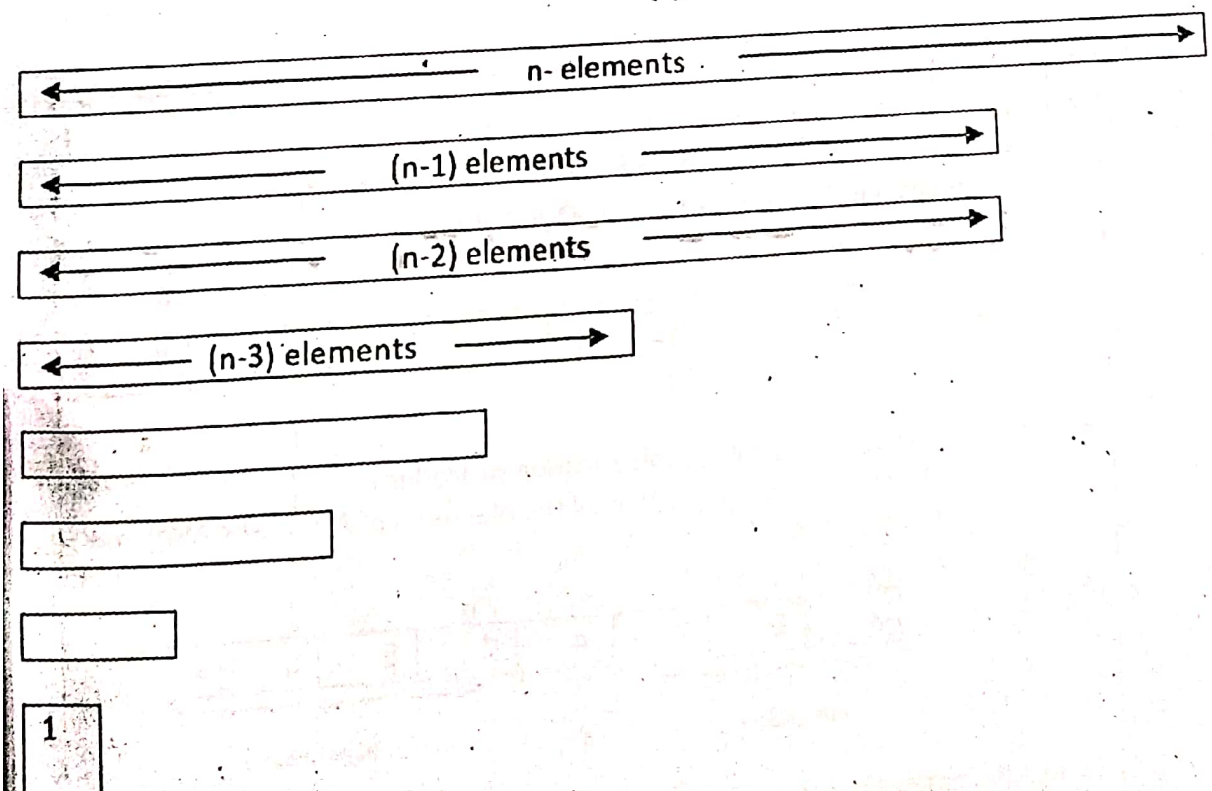
Best Case: If the input data is not sorted, then the partitioning of subarray is balanced; in this case the algorithm runs asymptotically as fast as merge-sort (i.e. 0(nlogn)).

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. θ (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:



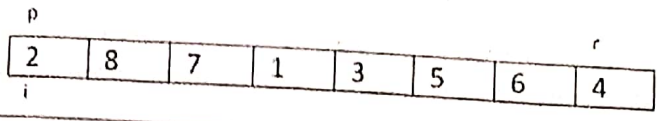
(a) Best Case



(b)

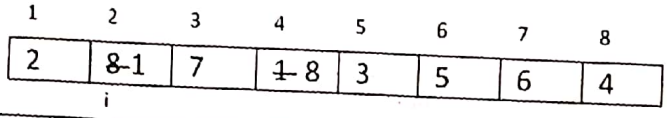
b)

2) j = 2; if A[2] ≤ 4 i.e 8 ≤ 4 ⇒ No
So line 5 – 6, will not be executed
Thus:



c)

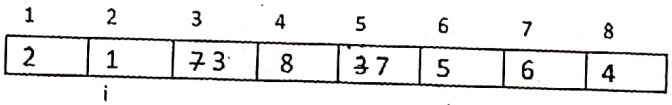
3) j = 3; if A[3] ≤ 4 i.e 7 ≤ 4 ⇒ No; so line 5-6 will not execute
4) j = 4; if A[4] ≤ 4 i.e 1 ≤ 4 ⇒ YES
So i ← i + 1 = 1 + 1 = 2
exchange (A[2] ↔ A[4])



d)

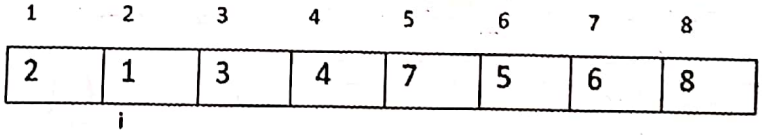
5) j = 5; A[5] ≤ 4 i.e 3 ≤ 4 ⇒ YES
- i ← i + 1 = 2 + 1 = 3

exchange (A [3] ↔ A [5])

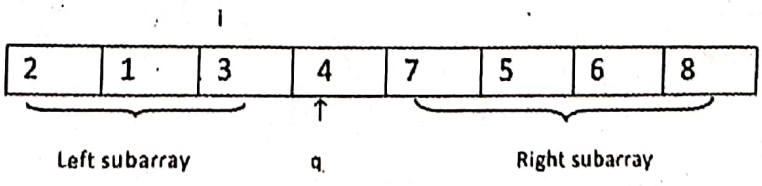


e)

6) j = 6; A[6] ≤ 4 i.e 5≤ 4 ⇒ NO
7) j = 7: A [7] ≤ 4 i.e 6 ≤ 4 ⇒ NO
Now for loop is now finished; so finally line 7 is execute i.e
exchange (A[4] ↔ A[8]), so finally we get:

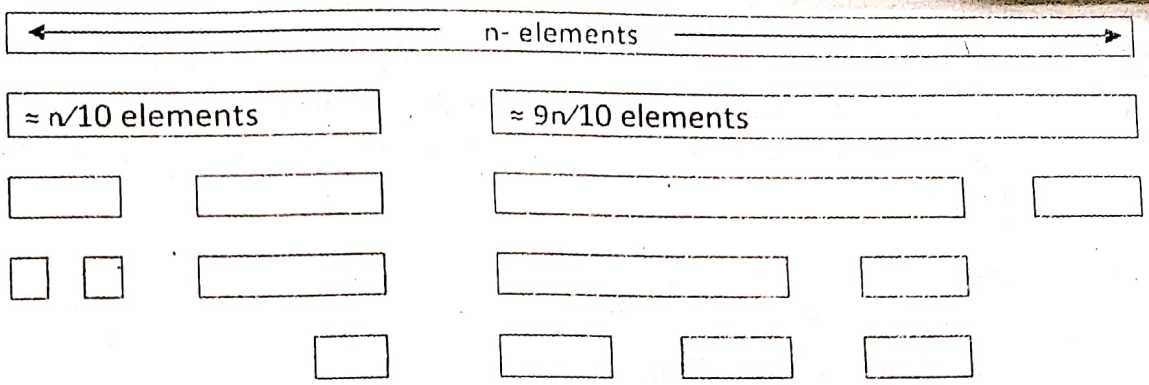


Finally we return (i + 1) i.e (3 + 1) = 4; by this partition procedure;
Now we can easily see that all the elements of A[1, ... 3] ≤ A[4]; and all the elements of A[5, ..8] ≥ A[4]. Thus



To sort the entire Array A{1..8}; there is a Recursive calls to QuickSort on both the subarray A[..3] and A[5...8].

(b) Worst Case



(c) Average Case

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:

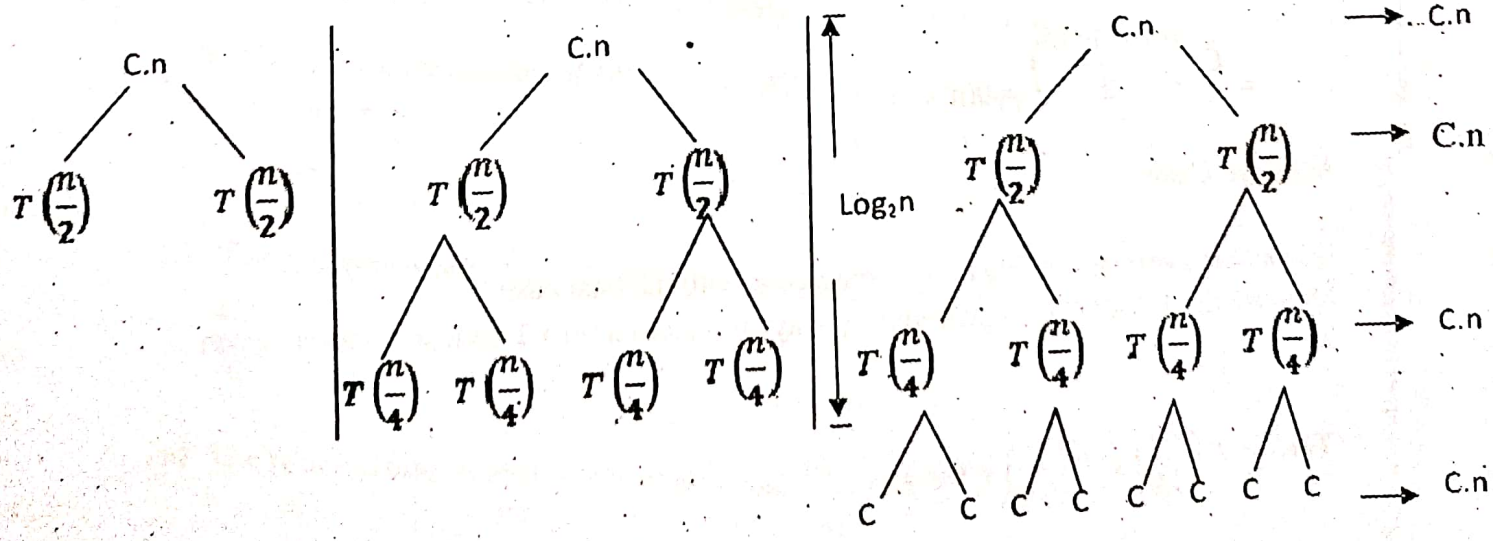
$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

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= 0 ($n^{\log_2 2}$) →Case 2 of master method

Method 2: Using Recursion Tree:

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



Total = C.n + C.n+ - - - - - -+ $\log_n n$ terms (c)

= C.n logn

θ (nlogn)

Worst Case:

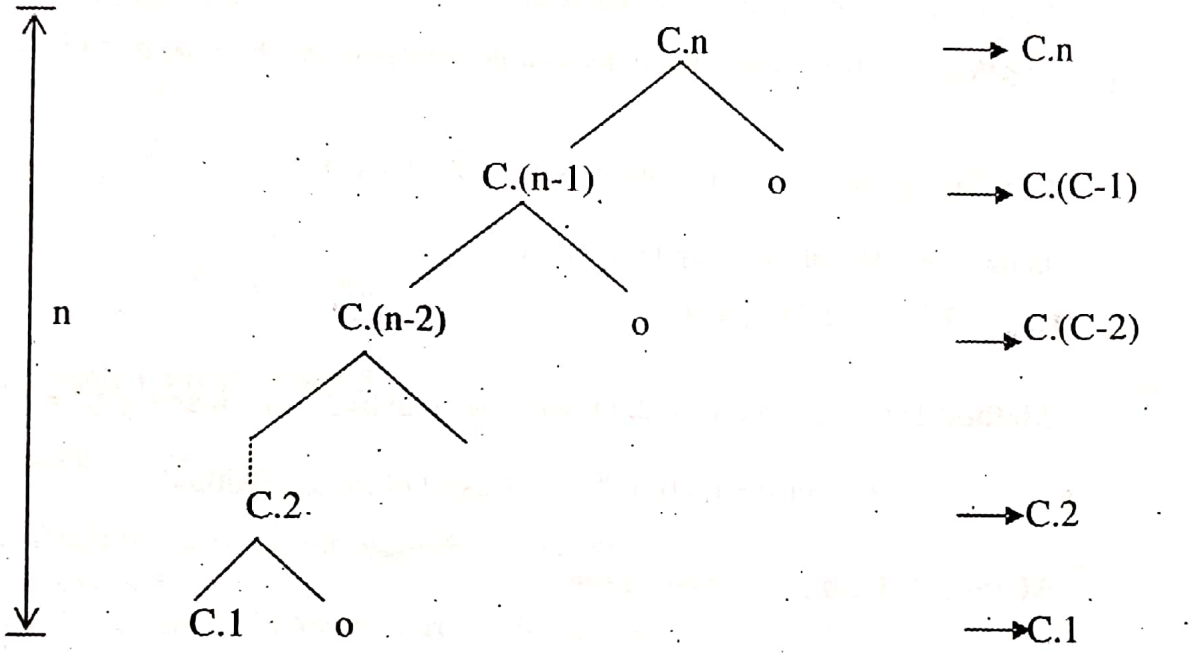
[When input array is already sorted]

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

In this case:

$$T(n) = T(n-1) + T(0) + \theta(n)$$
$$= T(n-1) + 0 + C.n$$

Recursion Tree:



Total = C (n + (n-1) + (n-2) + -------+2+1)

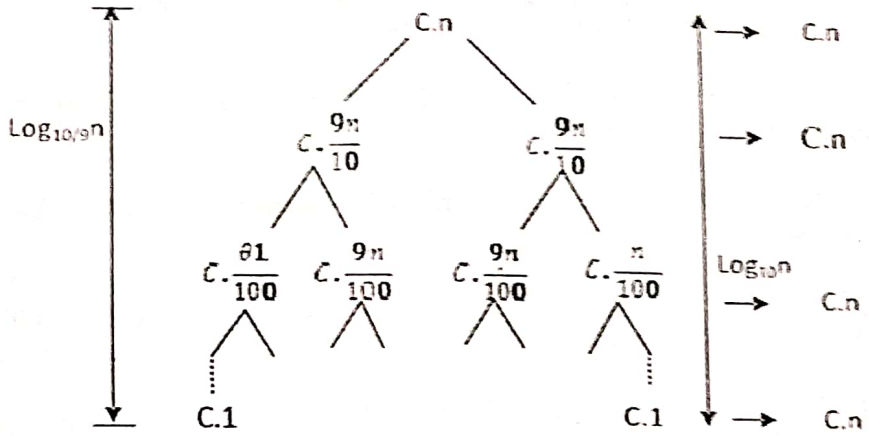
$$= C.\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

Average Case

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:.

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

Recursion Tree:



For Smaller Height:

Total = C.n+C.n+---$\log_{10}$n times

= C.n$\log_{10}$n

= Ω (nlogn)

For Bigger height

Total = C.n.+C.n +---$\log_{10/9}$n

= C.n$\log_{10/9}$n

= 0(nlogn)

$$\left.\begin{array}{l} T(n) = \Omega\ (n\log n) \text{ — (1)} \\ \&\ T(n) = 0(n\log n) \quad \text{— (2)} \end{array}\right\} \Rightarrow T(n) = \theta\ (n\log n)$$

☞ **Check Your Progress 2**

(Objective questions)

1) Which of the following algorithm have same time complexity in Best, average and worst case:

a) Quick sort b) Merge sort c) Binary search d) all of these

2) The recurrence relation of MERGESORT algorithm in worst case is:

a) – b) –

c) – d) –

3) The recurrence relation of QUICKSORT algorithm in worst case is:

a) – b)

c) – d)

4) The running time of PARTITION procedure of QUICKSORT algorithm is

a) b) c) d)

5) Suppose the input array A[1…n] is already in sorted order (increasing or decreasing) then it is _____ case situation for QUICKSORT algorithm

a) Best b) worst c) average d) may be best or worst

6) Illustrate the operation of MERGESORT algorithm to sorts the array:

70	35	5	85	45	88	50	10	60

7) Show that the running time of MERGESORT algorithm is $\Theta(nlogn)$.

8) Illustrate the operation of PARTITION Procedure on the array
$A =< 35,10,40,5,60,25,55,30,50,25 >$

9) Show that the running time of PARTITION procedure of QUICKSORT algorithm on a Sub-array on size n is $\Theta(n)$.

10) Show that the running time of QUICKSORT algorithm in the best case is $\Theta(nlogn)$

11) Show that the running time of QUICKSORT algorithm is $\Theta(n^2)$ when all elements of array A have the same value.

12) Find the running time of QUICKSORT algorithm when the array A is sorted in non increasing order.

2.5 INTEGER MULTIPLICATION

Input: Two n-bit decimal numbers x and y are represented as:
$X = < x_{n-1}\ x_{n-2}$ --------$x_1 x_0 >$ and
$Y = < y_{n-1}\ y_{n-2}$ --------$y_1 y_0 >$, where each x_i and y_i E{0, 1 ….9}.

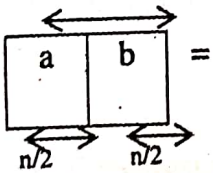
Output: The 2n-digit decimal is representative of the product x.y;
$x.\ y = z = z_{2n-2}\ z_{2n-3}$ .....$z_1\ z_0$

Note: The algorithm, which we are discussing here, works for any number base, e.g., binary, decimal, hexadecimal etc. For simplicity matter, we use decimal number.

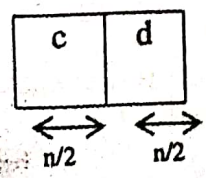
The straight forward method (Broute force method) requires $0(n^2)$ time to multiply two n-bit numbers. But by using divide and conquer, it requires only $0(n^{\log_2 3})$ i.e. $0(n^{1.59})$ time.

In 1962, A.A. Karatsuba discovered an asymptotically faster algorithm $0(n^{1.59})$ for multiplying two n-digit numbers using divide & conquer approach.

A Divide & Conquer based algorithm splits the number X and Y into ⌉ equal parts as:

X= 

= | x_{n-1} x_{n-2} ..... X ⌈n/2⌉ | X ⌈n/2⌉ ---------- x_1 x_0 | $= a \times 10^{n/2} + b$

Y= 

= | y_{n-1} y_{n-2} ..... X ⌈n/2⌉ | X ⌈n/2⌉ – 1 ---------- y_1 y_0 | $= c \times 10^{n/2} + d$

Note: Both number X and Y should have same number of digits; if any number has less number of digits then add zero's at most-significant bit position. So that we can easily get a, b, c and d of $\frac{n}{2}$ - digits. Now X and Y can be written as:

$X = a * 10^{n/2} + b$ -----┐

$Y = c * 10^{n/2} + d$ ------┐

For example : $\lfloor n/2 \rfloor$ = largest integer less than or equal to $n/2$

If X = 1026732

Y = 743914

Then X = 1026732 = 1026 x 10^3 + 732

Y = 0743914 = 0743 x 10^3 + 914

Now we can compute the product as:

$$Z = X.Y \left(a.10^{\lfloor n/2 \rfloor} + b\right).\left(c.10^{\lfloor n/2 \rfloor} + d\right)$$

$$X.Y = a.c.10^{2.\lfloor n/2 \rfloor} + (bc + ad)10^{\lfloor n/2 \rfloor} + b.d \quad \text{--------- (1)}$$

Where a, b, c, d is $\frac{n}{2}$ - digits. This equation ⊛ requires 4 multiplication of size $\frac{n}{2}$ – digits and 0(n) additions; Hence

$$T(n) = 4T\left(\frac{n}{2}\right) + 0(n)$$

After solving this recurrence using master method, we have: $T(n) = \theta(n^2)$; So direct (or Brute force) method requires O (n^2) time.

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

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

$$T(n) = \begin{cases} 0(1) & \text{if } n=1 \\ 3T(n/2) + 0(n) & \text{Otherwise} \end{cases}$$

Where 0(n) is the cost of addition, subtraction and digit shift (multiplications by power of 10's), all these takes time proportional to 'n'.

Method-1 - (Master Method)

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

a = 3
b = 2
f(n) = n
$n^{\log_b a} = n^{\log_2 3}$
$f(n)=n=\theta(n^{\log_2 3 - 1})$ ⇒ case 1 of Master Method

$\Rightarrow T(n) = \theta(n^{\log_2 3})$
$\Rightarrow \theta(n^{1.59})$

Method 2 (Substitution Method)

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

Now

$$T(n) = c.n + 3T\left(\frac{n}{2}\right)$$
$$= c.n + 3\left\{c.\frac{n}{2} + 3T\left(\frac{n}{4}\right)\right\}$$
$$= c.n + \frac{3}{2}cn + 3^2.T\left(\frac{n}{4}\right)$$
$$= c.n + \frac{3}{2}cn + 3^2\left\{c\frac{n}{4} + 3T\left(\frac{n}{8}\right)\right\}$$
$$= c.n + \frac{3}{2}cn + \left(\frac{3}{2}\right)^2 cn + 3^3 T\left(\frac{n}{8}\right)$$
$$= c.n + \frac{3}{2}cn + \left(\frac{3}{2}\right)^2 cn + 3^3\left\{c.\frac{n}{8} + 3T\left(\frac{n}{16}\right)\right\}$$
$$= c.n + \frac{3}{2}cn + \left(\frac{3}{2}\right)^2 cn + \left(\frac{3}{2}\right)^3 cn + \cdots + \left(\frac{3}{2}\right)^k .T\left(\frac{n}{2k}\right)$$
$$= c.n + \frac{3}{2}cn + \left(\frac{3}{2}\right)^2 cn + \left(\frac{3}{2}\right)^3 cn + \cdots + \left(\frac{3}{2}\right)^{\log n} .c$$
$$= c.n\left(1 + \frac{3}{2}cn + \left(\frac{3}{2}\right)^2 cn + \left(\frac{3}{2}\right)^3 cn + \cdots + \left(\frac{3}{2}\right)^{\log n}\right)$$
$$= c.n\left[\frac{1\left[\left(\frac{3}{2}\right)^{\log n+1} - 1\right]}{\frac{3}{2} - 1}\right]$$
$$= 2\,c.n\left[\left(\frac{3}{2}\right)^{\log n+1} - 1\right]$$
$$= 2\,c.n\left[\left(\frac{3}{2}\right)^1\left(\frac{3}{2}\right)^{\log n} - 1\right]$$
$$= 2\,c.n\left[\frac{3}{2}n^{\log_2 3/2} - 1\right]$$
$$= 2\,c.n\left[\frac{3}{2}n^{\log_2 2 - \log_2 2} - 1\right]$$
$$= 2\,c.n\left[\frac{3}{2}n^{\log_2 2^{-1}} - 1\right]$$

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

2.6 MATRIX MULTIPLICATION

Let A and B be two $(n \times n)$ – matrices.

$A = (a_{ij})\ i,j = 1 \ldots n$ $\quad [a_{ij}]$ where $i,j = 1 \ldots\ldots n$
$B = (b_{ij})\ i,j = 1 \ldots n$ $\quad c_{ij}\ B = [b_{ij}]$ where $i, j = n$

The product matrix $C = A.B. = (Cij)_{i,j=1\ldots n}$ is also an $(n \times n)$ matrix, where $(i,j)^{th}$ elements is defined as:

$$Cij = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Straight forward method:

To compute Cij using this formula, we need multiplications.
As the matrix C has (n^2) elements, the time for the resulting matrix multiplication is

1) Divide & Conquer Approach:

The divide and conquer strategy is yet another way to compute the product of two $(n \times n)$ matrices. Assuming that n is an exact power of 2 (i.e. $n=2^k$). We divide each of A, B and C into four

$\left(\frac{n}{2} \times \frac{n}{2}\right)$ matrices. i.e.

$$A = \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \text{ and } B = \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right)$$ where each A_{ij} and B_{ij} are sub matrices of C_{ij}

$\left(\frac{n}{2} \times \frac{n}{2}\right)$.

$$A = \left(\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{21}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right)$$

i.e. $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
$C_{12} = A_{11}B_{12} + A_{12}B_{22}$
$C_{21} = A_{21}B_{11} + A_{22}B_{21}$
$C_{22} = A_{21}B_{12} + A_{22}B_{22}$

(2) Here all Aij, Bij are sub matrices of size $\left(\frac{n}{2} \times \frac{n}{2}\right)$.

Algorithm Divide and Conquer Multiplication (A,B)

1. n ← no. of rows of A
2. if n = 1 then return $(a_{11} b_{11})$
3. else

4. Let Aij, Bij (for i,j = 1,2, be $\left(\frac{n}{2} \times \frac{n}{2}\right)$ submatrices)

S.t $A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$ and $B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$

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

6. Compute $C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$
$C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$
$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}$
$C_{22} = A_{21}.B_{21} + A_{22}.B_{22}$

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 $8T\left(\frac{n}{2}\right)$ arithmetic operations.
(i.e. in order to compute AB using e.g. (2), we need 8- multiplications of $\left(\frac{n}{2} \left(\frac{n}{2}\right)\right.$ matrices).
- Line 6 requires $4\left(\frac{n}{2}\right) = \theta(n^2)$
(i.e 4 additions of $\left(\frac{n}{2} \left(\frac{n}{2}\right)\right.$ 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

$f(n) = n^2 = O(n^{\log_2 - })$ => case 1of master method

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

Now we can see by using V. Stressen's Method, we improve the time complexity of matrix multiplication from $O(n^3)$ to $O(n^{2.81})$.

3) Strassen's Method

Volker Strassen had discovered a way to compute the Cij of Eg. (2) using only (7) multiplication and (18) additions / subtractions.

This method involves (2) steps:

1) Let

$$P_1 = (A_{11}+A_{22})\,(B_{11}+B_{22})$$
$$P_2 = (A_{21}+A_{22}).B_{11}$$
$$P_3 = A_{11}\,(B_{12}-B_{22})$$
$$P_4 = A_{22}\,(B_{21}-B_{11})$$
$$P_5 = (A_{11}+A_{12}).B_{22}$$
$$P_6 = (A_{21}-A_{11})\,(B_{11}+B_{12})$$
$$P7 = (A_{12}-A_{22})\,(B_{21}+B_{22}) \qquad \text{——— (I)}$$

Recursively compute the (7) $\left(\frac{n}{2}\ (\frac{n}{2}\right)$ matrices $P_1, P_2 \ldots\ldots P_7$ as in eg. (I)

2) Then, the Cij are computed using the formulas in eg. (II).

$$C_1 = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6 \qquad \text{——— (II)}$$

Here the overall computing time

$$T(n) = \begin{cases} \theta(1) & n < 2 \\ 7T(n/2) + \theta(n^2) & \text{Otherwise} \end{cases} \Rightarrow$$

Using master method: a = 7, b = 2 and $n^{\log_b a} = n^{\log_2 7} = n^{2.81}$ $f(n) = n^2$;

$f(n) = n^2 = O(n^{\log 2-})$ => case 1of 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:

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

$$A_{21} = \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} \quad A_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \quad B_{21} = \begin{bmatrix} 2 & 0 \\ 1 & 4 \end{bmatrix} \quad B_{22} = \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}$$

g only (7)

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):

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$P_2 = (A_{21} + A_{22})B_{11}$$
$$P_3 = A_{11}(B_{12} - B_{22})$$
$$P_4 = A_{22}(B_{21} - B_{11})$$
$$P_5 = (A_{11} + A_{12})B_{22}$$
$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$
$$C_{11} = P_1 + P_4 - P_5 + P_7$$
$$C_{12} = P_3 + P_5$$
$$C_{21} = P_2 + P_4$$
$$C_{22} = P_1 + P_3 - P_2 + P_6$$

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

$$P_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) = \begin{bmatrix}1 & 2\\0 & 6\end{bmatrix} + \begin{bmatrix}1 & 2\\5 & 0\end{bmatrix} \times \begin{bmatrix}1 & 4\\3 & 1\end{bmatrix} + \begin{bmatrix}1 & 3\\5 & 1\end{bmatrix}$$

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

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

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

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

$$P_5 = (A_{11} \times A_{12}) \times B_{22} = \begin{bmatrix}34 & 18\\45 & 9\end{bmatrix}$$

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

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

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

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

$$C_{21} = P_2 + P_4 = \begin{bmatrix}11 & 25\\19 & 3\end{bmatrix}$$

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

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

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

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

Otherwise

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

$n^{\log_b a} = n^{\log_2 7} = n^{2.81}$

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

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

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

The solution of this recurrence is $T(n) = 0(n^{\log_2 7}) = 0(n^{2.81})$

☞ Check Your Progress 3

(Objective questions)

1) The recurrence relation of INTEGER Multiplication algorithm using Divide & conquer is:

a) $T(n) = 3T\left(\frac{n}{2}\right) + O(n^2)$ b) $T(n) = 4T\left(\frac{n}{2}\right) + O(n^2)$

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

2) Which one of the following algorithm design techniques is used in Strassen's matrix multiplication algorithm?

(a) Dynamic programming
(b) Backtracking approach
(c) Divide and conquer strategy
(d) Greedy method

3) Strassen's algorithm is able to perform matrix multiplication in time_____.

(a)$O(n^{2.61})$ (b) $O(n^{2.71})$ (c) $O(n^{2.81})$ (d) $O(n^3)$

4) Strassen's matrix multiplication algorithm ($C = AB$),if the matrices A and B are not of type $2^n \times 2^n$, the missing rows and columns are filled with _____.

(a) 0's (b) 1's (c) -1's (d) 2's

5) 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?

(a) 9 and 9 (b) 6 and 12 (c)12 and 6 (d) none of theses

6) Multiply 1026732× 0732912 using divide and conquer technique (use karatsuba method).

7) Use Strassen's matrix multiplication algorithm to multiply the following two matrices:

$A = \begin{bmatrix} 5 & 3 \\ 6 & 7 \end{bmatrix}$ $B = \begin{bmatrix} 1 & 4 \\ 5 & 9 \end{bmatrix}$

2.7 SUMMARY

- Many useful algorithms are recursive in structure as they make a recursive call to itself until a base (or boundary) condition of a problem is not reached. These algorithms closely follow the **Divide and Conquer** approach.
- **Divide and Conquer** is a top-down approach, which directly attack the complete instance of a given problem and break down into smaller parts.
- Any divide-and-conquer algorithms consists of 3 steps:
  1) **Divide**: The given problem is divided into a number of sub-problems.
  2) **Conquer**: Solve each sub-problem be calling them recursively.
     (**Base case:** If the sub-problem sizes are small enough, just solve the sub-problem in a straight forward or direct manner).
  3) **Combine**: Finally, we combine the sub-solutions of each sub-problem (obtained in step-2) to get the solution to original problem.
- To analyzing the running time of divide-and-conquer algorithms, we use a **recurrence equation** (more commonly, a **recurrence**). Any algorithms which follow the divide-and-conquer strategy have the following recurrence form:

$$T(n) = \begin{cases} \theta(1) & if\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & Otherwise \end{cases}$$

Where

T(n) = running time of a problem of size n

a means "In how many part the problem is divided"

– means "Time required to solve a sub-problem each of size (n/b)"

D(n) + C(n) = f(n) is the summation of the time requires to divide the problem and combine the sub-solutions.

- Applications of divide-and conquer strategy are Binary search, Quick sort, Merge sort, multiplication of two n-bit numbers and V. Strassen's matrix multiplications.
- The following table summarizes the recurrence relations and time complexity of the various problems solved using Divide-and-conquer.

Problems that follows	Recurrence relation	Time complexity		
		Best	Worst	Average
Binary search	Worst case: –	O(1)		
Quick Sort	Best case: –			
	Worst Case:			
	Average: — —			
Merge sort	Best case or worst: –			
Multiplication of two n-bits numbers	Worst case: –			
Strassen's matrix multiplication	Worst case: –			