Unit - 2 :- Brute Force Design Technique

The solution is only based on the answer to the problem instead of taking efficiency into consideration.

→ Advantages:-

(i) The solution / algorithm are simple

(ii) Wide range of Problems can be solved

→ Disadvantages :-

(i) Some-times the algorithm is too slow

(ii) Efficiency is not always present.

⇒ Selection Sort :- (An application of Brute Force Technique)

Inplace but not Stable

i									
6	8	2	4	1	9	5	3	7	0
0	i 8	2	4	1	9	5	3	7	6
0	1	i 2	4	8	9	5	3	7	6
0	1	2	i 4	8	9	5	3	7	6
0	1	2	3	i 8	9	5	4	7	6
0	1	2	3	4	i 9	5	8	7	6
0	1	2	3	4	5	i 9	8	7	6

0 1 2 3 4 5 6 8 7 9 (i)

0 1 2 3 4 5 6 7 8 9 (i)

0 1 2 3 4 5 6 7 8 9 ⇒ Sorted array.

Algorithm :- SelectionSort (A[0...n-1])

// Input :- A given array A[0..n-1] of 'n' elements

// Output :- The sorted Array

```
for i ← 0 to n-2 do
    Min ← i
    for j ← i+1 to n-1 do
        if A[j] < A[Min]
            Min ← j
    swap (A[i], A[Min])
```

⇒ Mathematical Analysis :-

(i) Input Size ⇒ n

(ii) Basic operation ⇒ Comparison

(iii) The basic operations number only depends on the size of the array.

$\therefore C_{Best}(n) = C_{avg}(n) = C_{worst}(n) = C(n)$

l_{worst} :

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} n-1-i-x+x$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} (1) - \frac{(n-2)(n-1)}{2}$$

$$= \frac{(n-1)(n-1)}{} - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)\left[(n-1) - \frac{(n-2)}{2}\right]$$

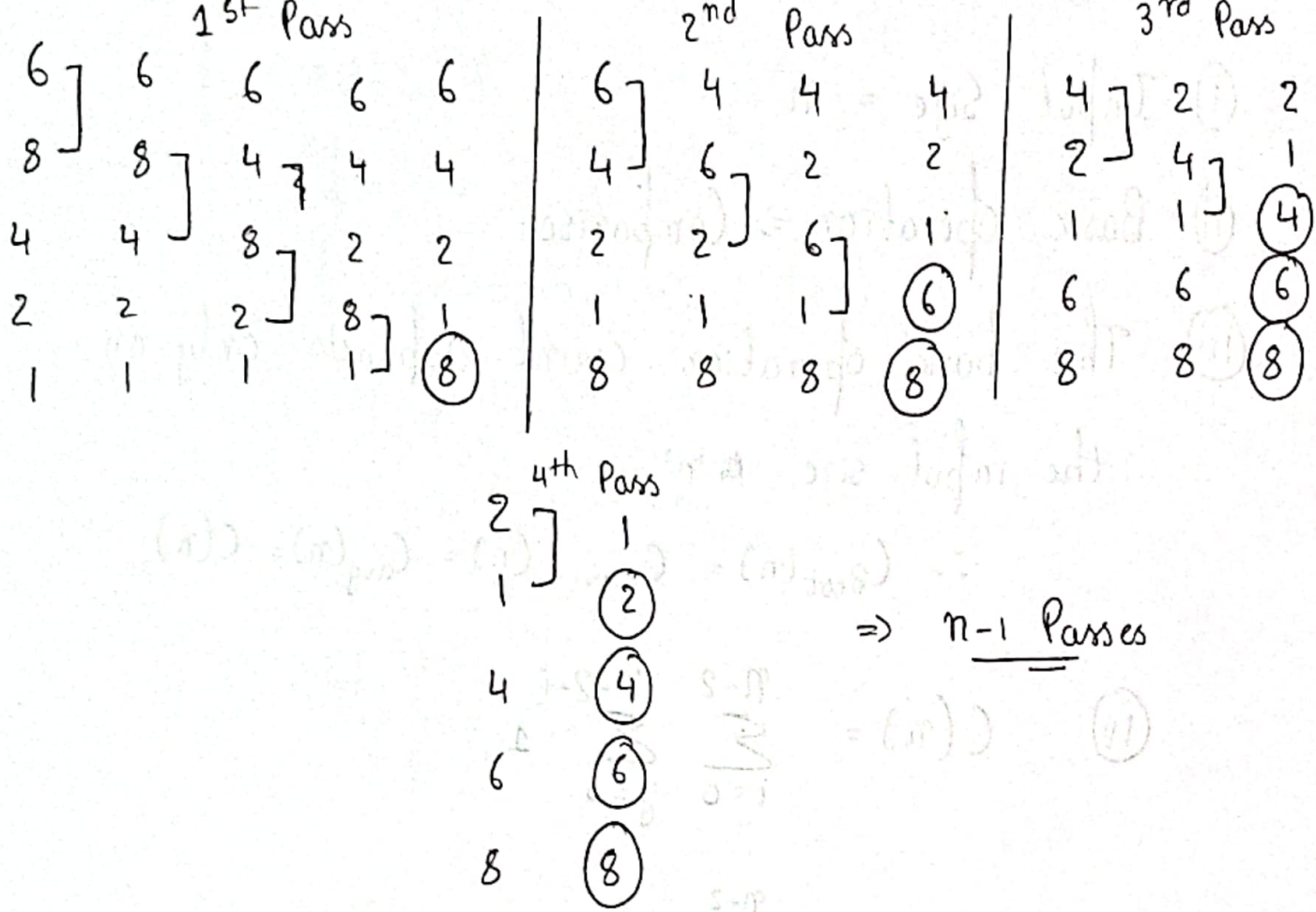
$$= \frac{(n-1)(2n-2-n+2)}{2}$$

$$= \frac{n(n-1)}{2}$$

$$C(n) \approx \frac{n^2}{2}$$

$$\therefore \quad C(n) \in \Theta(n^2)$$

Bubble Sort :- Inplace and Stable



⇒ n-1 Passes

Algorithm :-

```
BubbleSort (A[0...n-1])
// Input :- Array A[0...n-1] of n elements
// Output :- Array sorted in ascending order
    for i ← 0 to n-2 do
        for j ← 0 to n-i-2 do
            if A[j] > A[j+1]
                Swap (A[j], A[j+1])
```

Analysis :-

(i) Input Size = n

(ii) Basic operation ⇒ Comparison

(iii) The basic operation count depends only on the input size 'n'

$$\therefore C_{best}(n) = C_{worst}(n) = C_{avg}(n) = C(n)$$

(iv) $$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

$$= \sum_{i=0}^{n-2} (n-2-i-0+1)$$

$$= \sum_{i=0}^{n-2} (n-1)-i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$= (n-1)(n-1) - \frac{(n-2)(n-1)}{2}$$

$$= \frac{(n-1)(n)}{2}$$

$$\approx \frac{n^2}{2}$$

$$\therefore C(n) \in \theta(n^2)$$

=> Better Bubble Sort :-

```
for i ← 0 to n-2 do
    flag ← 0
    for j ← 0 to n-2-i do
        if a[j] > a[j+1]
            swap(a[j], a[j+1])
            flag ← 1
    if flag = 0
        break
```

Here, in this algorithm

$$C_{Best}(n) = \sum_{j=0}^{n-2-0} 1$$

$$= n-2-0+1$$

$$= n-1$$

$$\therefore C_{Best}(n) \in \theta(n)$$

③ Design an algorithm (Brute force) for solving a Polynomial

$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x_1 + a_0$ and determine

Worst case efficiency

ⓐ If the algorithm is designed for $\theta(n^2)$, if possible convert the algorithm into linear algorithm

ⓒ Is it possible to design an algorithm with better than linear efficiency for this problem

→ Algorithm :-

```
Polynomial (A[0... n], x0)

Sum ← 0
for i ← n to 0 do
    power ← 1
    for j ← 1 to i do
        power ← power * x0
    Sum ← Sum + power * A[i]
return Sum
```

$$M(n) = \sum_{i=0}^{n} \sum_{j=1}^{i} 1$$

$$= \sum_{i=0}^{n} i$$

$$M(n) = \frac{n(n+1)}{2}$$

$$\therefore M(n) \in \theta(n^2)$$

(b) Linear Algorithm :-

```
Sum ← A[0]
Power ← 1
for i ← 1 to n do
    Power ← Power * x.
    Sum ← Sum + A[i] * Power
end for
return Sum
```

Analysis :-

Input Size ⇒ n

Basic Operation ⇒ Multiplication.

$$M(n) = \sum_{i=1}^{n} 2$$

$$= 2 \sum_{i=1}^{n} 1$$

$$M(n) = 2n$$

$$\therefore M(n) \in \theta(n)$$

© The algorithm cannot be reduced to order of growth less the linear.

Note :- Linear Search is also a Brute Force Technique Algorithm.

⇒ Brute Force String Matching / Naive String Matching Algorithm :-

Inputs :- A String of 'n' characters (Text)
A String of 'm' characters (Pattern)

Output :- Returns index of leftmost position in Text if the substring matches the pattern.

Ex:- A A B A A C A A D A A B A A A B A A n = 18

```
A A F A                          3   m = 4
  A A F A                        2
    A A F A                      1
      A A F A                    3
        A A F A                  2
          A A F A                1
            A A F A              3
              A A F A            2
                A A F A          1
                  A A F A        3
                    A A F A      2
                      A A F A    1
                        A A F A  3
                          A A F A  3
                            A A F A  2
```

15 Realignments

Total 32 Comparisons

Algorithm :-

BruteForceStringMatch (T[0...n-1], P[0...m-1])

// Input :- A String T[0...n-1] of size 'n' and characters (Text)

An array P[0...m-1] of 'm' characters (Pattern)

// Output :- Index of the first character in the text that matches the pattern ~~Else -1~~

Else -1

```
for i ← 0 to n-m
    j ← 0
    while j < m and P[j] = T[i+j]
        j ← j+1
    end while
    if j = m
        return i
end for
return -1
```

Mathematical Analysis :-

Worst case scenario when m comparisons are done in each alignment

① Basic Operation ⇒ Comparison

$$C_{worst}(n) = \sum_{i=0}^{n-m} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m} m-1+1$$

$$= \sum_{i=0}^{n-m} m$$

$$= m \sum_{i=0}^{n-m} 1$$

$$= m\,(n-m-0+1)$$

$$C_{worst}(n) = m\,(n-m+1)$$

→ Best Case Scenario :-

When the Pattern matches the Text in the first alignment itself.

$$\therefore C_{Best}(n) = m$$

Average case :-

$$\therefore C_{avg}(n) = n$$

(1) Find the number of character comparison required by Brute Force String Matching Algorithm:-

(i) THERE_IS_MORE_TO_LIFE_THAN_INCREASING_ITS_SPEED

Pattern :- GANDHI

Number of Comparison = 43

(2) How many successful and unsuccessful comparisons are made in searching each of following pattern in a binary text of 1000 zeroes.

(i) 00001

Here, $n = 1000$ $m = 5$

No of alignment = $n - m + 1 = 996$

No of comparison for each alignment = 5

∴ Total no of comparison = 5×996

$= 4980$

(ii) 10000

Here, $n = 1000$ $m = 5$

No of comparison = $1 \times (1000 - 5 + 1)$

$= 996$

(iii) 01010

No of comparison = $2 \times (1000 - 5 + 1)$

$= 2 \times 996$

$= 1992$

⇒ Exhaustive Search :-

Brute force approach to solve combinatorial problems.

⇒ Nab Sack :-

We are given with 'n' items with known weights and each of these are associated with values, you are given a bag of capacity 'w' which has to be filled with a subset such that the value of ~~ba~~ subset is maximum.

Ex :-

Number of items, $n = 3$

Capacity, $W = 105$

$[w_1, w_2, w_3] = [100, 10, 10]$

$[v_1, v_2, v_3] = [20, 15, 15]$

Subset	Weight	Value	
2^n Subsets: {0}	0	0	
{1}	100	20	
{2}	10	35	
{3}	10	15	
{1,2}	100+10	20+15 (35)	(Not feasible)
{1,3}	100+10	20+15 (35)	(Not feasible)
{2,3}	10+10	15+15 (30)	⇒ Optimal Soln
{1,2,3}	100+10+10	20+15+15 (50)	(Not feasible)

⇒ Travelling Salesman Problem :- (Minimization Problem)

The problem can be visualized as a weighted graph where vertices represent cities and edges represent cost to travel from one city to another

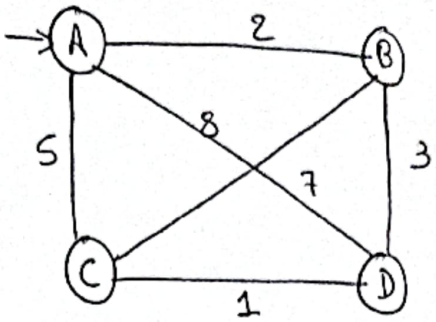
Conditions :-

(i) The Salesman must visit all the cities

(ii) The cost of travel must be minimum

(iii) Each city must be visited only once

Ex:-



Minimum:-

ABDCA

= 2+3+1+5

= 11

From each city, there are $(n-1)!$ ways to travel which are feasible

⇒ Job Assignment Problem

There are 'n' people to be assigned to 'n' jobs, which job has to be assigned to which person so that minimum cost is required when cost required to complete the same job by different person is different

	J_1	J_2	J_3	J_4
P_1	9	(2)	7	8
P_2	(6)	4	3	7
P_3	5	8	(1)	8
P_4	7	6	9	(4)

C [i, j] ⇒ Cost for i^{th} person to complete j^{th} job

Minimum Cost = 13

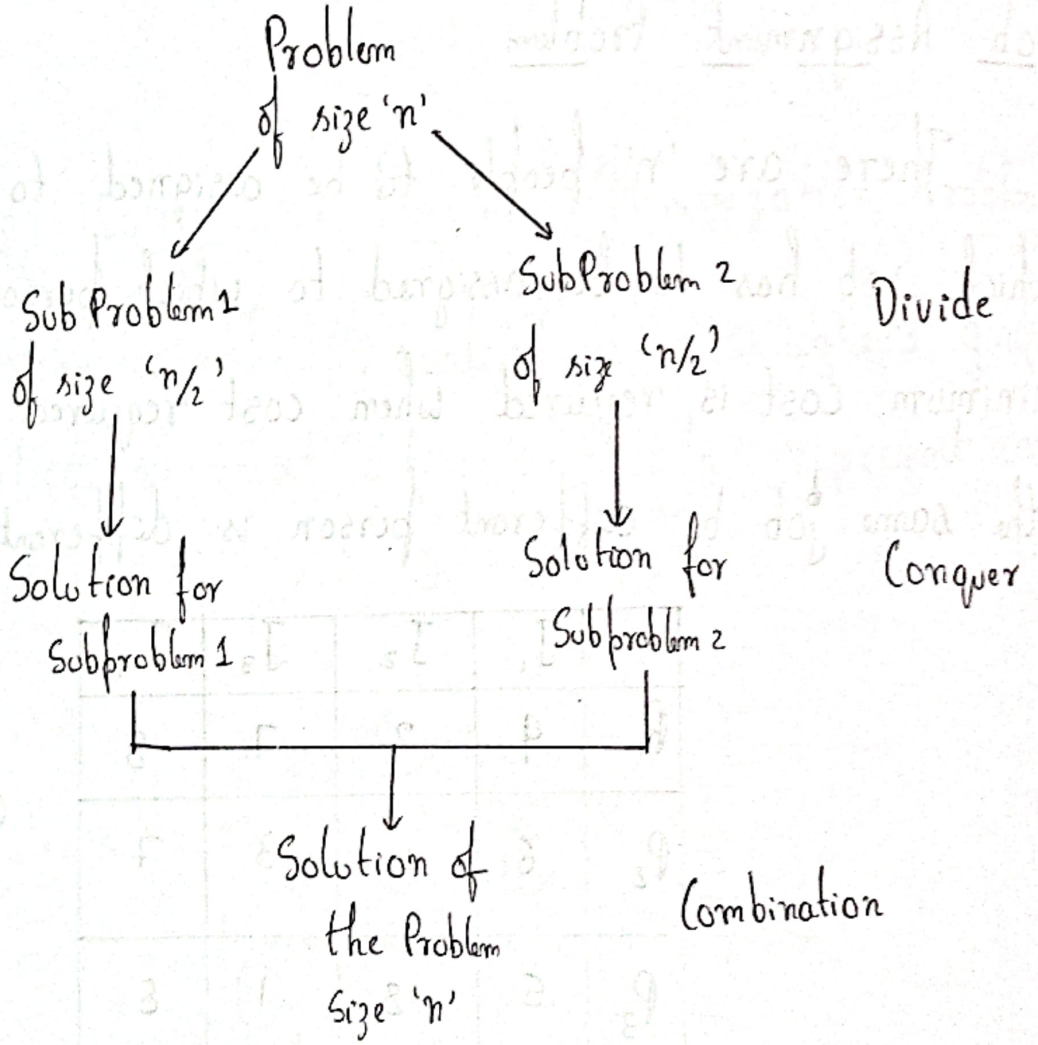
⇒ Divide and Conquer :-

→ Steps involved :-

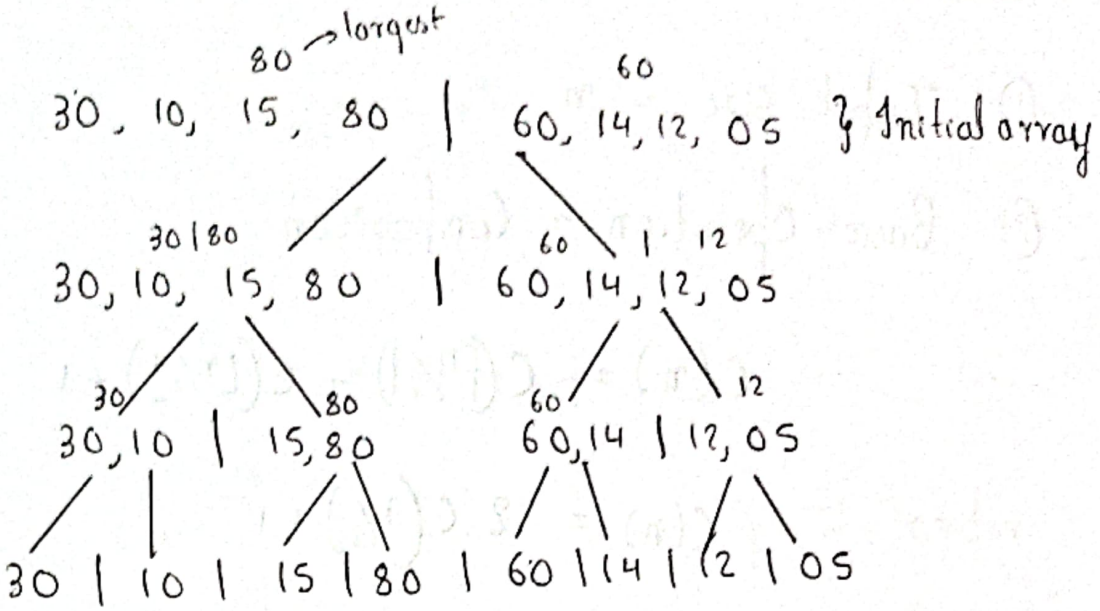
(i) The problem instance is divided into smaller instances of the almost same size.

(ii) The smaller instances are solved

(iii) If necessary, the smaller instances are combined to obtain solution of the original instances



⇒ Finding largest element using divide and conquer.



Algorithm :-

Largest (A[0....n-1])

// Input :- Array A[0... n-1] of size 'n'

// Output :- Largest element in array.

```
Largest (A[l... r])
    if l = r
    return l
    Else
      t1 ← largest (A[l.... (l+r)/2])
      t2 ← largest (A[[(l+r)/2+1].... r])
      if A[t1] >= A[t2]
        return t1
    Else
        return t2
```

Mathematical analysis :-

① Input size = n

② Basic operation = Comparison

$$C(n) = C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1$$

Initial Condition
$C(1) = 0$

$$C(n) = 2\,C(n/2) + 1$$

By Smoothness theorem

$$n = 2^k$$

$$C(2^k) = 2\,C(2^{k-1}) + 1$$

$$C(2^k) = 2\left[2C(2^{k-2}) + 1\right] + 1$$

$$= 4\,C(2^{k-2}) + 2 + 1$$

$$= 4 \times \left[2\,C(2^{k-3}) + 1\right] + 2 + 1$$

$$= 2^3\,C(2^{k-3}) + 4 + 2 + 1$$

$$= 2^3\,C(2^{k-3}) + 2^2 + 2^1 + 2^0$$

$$= 2^i\,C(2^{k-i}) + 2^{i-1} + 2^{i-2} \cdots + 2^0$$

Put $i = k$

$$= 2^k\,C(2^{k-k}) + \frac{1\,(2^{k-1+1} - 1)}{2-1}$$

$$= 2^k\,C(1) + 2^k - 1$$

$$\therefore \quad C(2^k) = 2^k(0)^0 + 2^k - 1$$

$$C(2^k) = 2^k - 1$$

$$\therefore \quad C(n) = n - 1$$

$$C(n) \in \Theta(n)$$

* For finding the largest element in array, the order of growth using Divide and conquer and Brute force technique is same.

* But, some drawbacks in Divide and Conquer:-

  * It requires more time and space due to the recursive calls.

⇒ General Expression of a Divide and Conquer Technique:-

Let the problem be divided into 'b' parts and among which 'a' parts are solved.

$$T(n) = a \cdot T(n/b) + f(n) \qquad \begin{matrix} b > 1 \\ a \geq 1 \end{matrix}$$

$f(n)$ → Time taken for combining the result of 'a' parts

=> Master Theorem :-

For any recurrence relation of the form

$$T(n) = a\,T(n/b) + f(n) \quad a \geq 1,\ b > 1$$

where, $f(n) \in \theta(n^d)$ with $d \geq 0$

Then,

$$T(n) \in \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_2 n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Ex :-

Let, recurrence relation

$$C(n) = 2\,(C(n/2)) + 1$$

$a = 2,\ b = 2,\ f(n) = 1 \Rightarrow d = 0$

$\Rightarrow a > b^d \Rightarrow 2 > 2^0 \Rightarrow 2 > 1$

$\Rightarrow T(n) \in \theta(n^{\log_2 2})$

$\Rightarrow C(n) \in \theta(n)$ //

① Find order of growth of following reccurence relation using Master's Theorem :-

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

Here, $a = 4$, $b = 2$, $f(n) = n$

$\Rightarrow n^d = n \Rightarrow d = 1$

$\therefore 4 > 2^1 \Rightarrow a > b^d$

$\therefore T(n) \in \Theta(n^{\log_2 4})$

$T(n) \in \Theta(n^2)$ //

(ii) $T(n) = 4T(n/2) + n^2$

Here, $a = 4$, $b = 2$, $f(n) = n^2$

$\Rightarrow n^d = n^2 \Rightarrow d = 2$

$\therefore 4 = 2^2 \Rightarrow a = b^d$

$T(n) \in \Theta(n^d \log_2 n)$

$T(n) \in \Theta(n^2 \log_2 n)$ //

(iii) $T(n) = 4T(n/2) + n^3$

$a = 4$, $b = 2$, $f(n) = n^3$

$\Rightarrow n^d = n^3 \Rightarrow d = 3$

$a = 4 < b^d = 2^3$.

$\therefore T(n) \in \Theta(n^d)$

$T(n) \in \Theta(n^3)$ //

⇒ Write a Pseudocode for computing a^n where $a > 0$ and 'n' is some positive integer using divide and conquer

```
Power (int a, n)
// Input :- Positive integers a and n
// Output :-  a^n
   if n = 1
      return a
   Else
      return  Power(a, ⌊n/2⌋) * Power(a, ⌈n/2⌉)
```

Analysis :-

① Input Size ⇒ n

② Basic Operation ⇒ Multiplication

$$M(n) = M\left(\lfloor n/2 \rfloor\right) + M\left(\lceil n/2 \rceil\right) + 1$$

Initial Condition
$M(1) = 0$

$$M(n) = 2M\left(n/2\right) + 1$$

By Smoothness Theorem,

$$n = 2^k$$

$$M(2^k) = 2M(2^{k-1}) + 1$$

$$= 2\left[2M(2^{k-2}) + 1\right] + 1$$

$$= 2^2 M(2^{k-2}) + 2 + 1$$

$$= 2^2\left(2M(2^{k-3}) + 1\right) + 2 + 1$$

$$= 2^3 M(2^{k-3}) + 2^2 + 2 + 1$$

$$\vdots$$

$$= 2^i M(2^{k-i}) + 2^{i-1} + 2^{i-2} + \ldots 2^0$$

Put $k = i$

$$= 2^k \cancel{M(2^0)}^{0} + 1\,\frac{(2^k - 1)}{2 - 1}$$

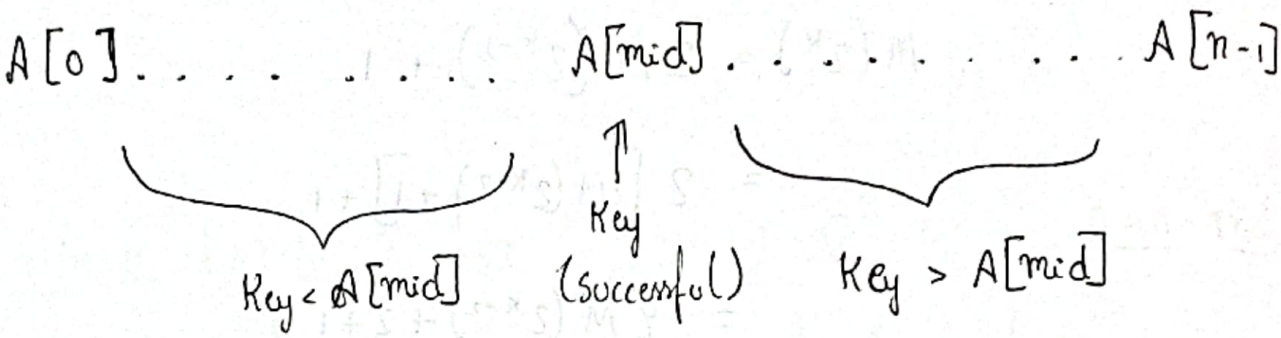
$$= 2^k - 1$$

$$\therefore M(n) = n - 1 \;//$$

$$\therefore M(n) \in \Theta(n)$$

⇒ Binary Search

* It is an example for divide and conquer technique
* Works only on sorted array
* Compares the key to be searched with the middle element



⇒ Algorithm :-

```
BinarySearch (A[0....n-1], Key, l, r)
{ // Input :- A sorted array A[0....n-1], the key to be searched
  and left, right indices of the array
// Output :- Return index of the Key element if found else
              return -1

if l > r
   return -1

Else
   mid ← (l+r)/2
   if Key = A[mid]
      return mid
   Else
      if Key < A[mid] return BinarySearch (A, Key, l, mid-1)
```

Else

return BinarySearch (A, Key, mid+1, r)

}

Analysis :-

① Input size => n

② Basic ~~Compo~~ Operation => Comparision

Worst Case Analysis :-

$$C_{worst}(n) = C(n/2) + 1 \qquad C(1) = 1$$

By Smoothness Theorem

$\because n = 2^k$

$$C_{worst}(2^k) = C(2^{k-1}) + 1$$

$$= C(2^{k-2}) + 1 + 1$$

$$= C(2^{k-3}) + 1 + 1 + 1$$

$$\vdots$$

$$= C(2^{k-i}) + i$$

Put $i = k$

$$\therefore C_{worst}(2^k) = C(1) + k$$

$$C_{worst}(2^k) = k + 1$$

$$C_{worst}(n) = \log_2 n + 1$$

$$\Rightarrow C_{worst}(n) \in \Theta(\log_2 n)$$

→ Best Case Scenario :- When the first middle element of the array itself is the key to be searched.

$$\therefore C_{Best}(n) \in \Theta(1)$$

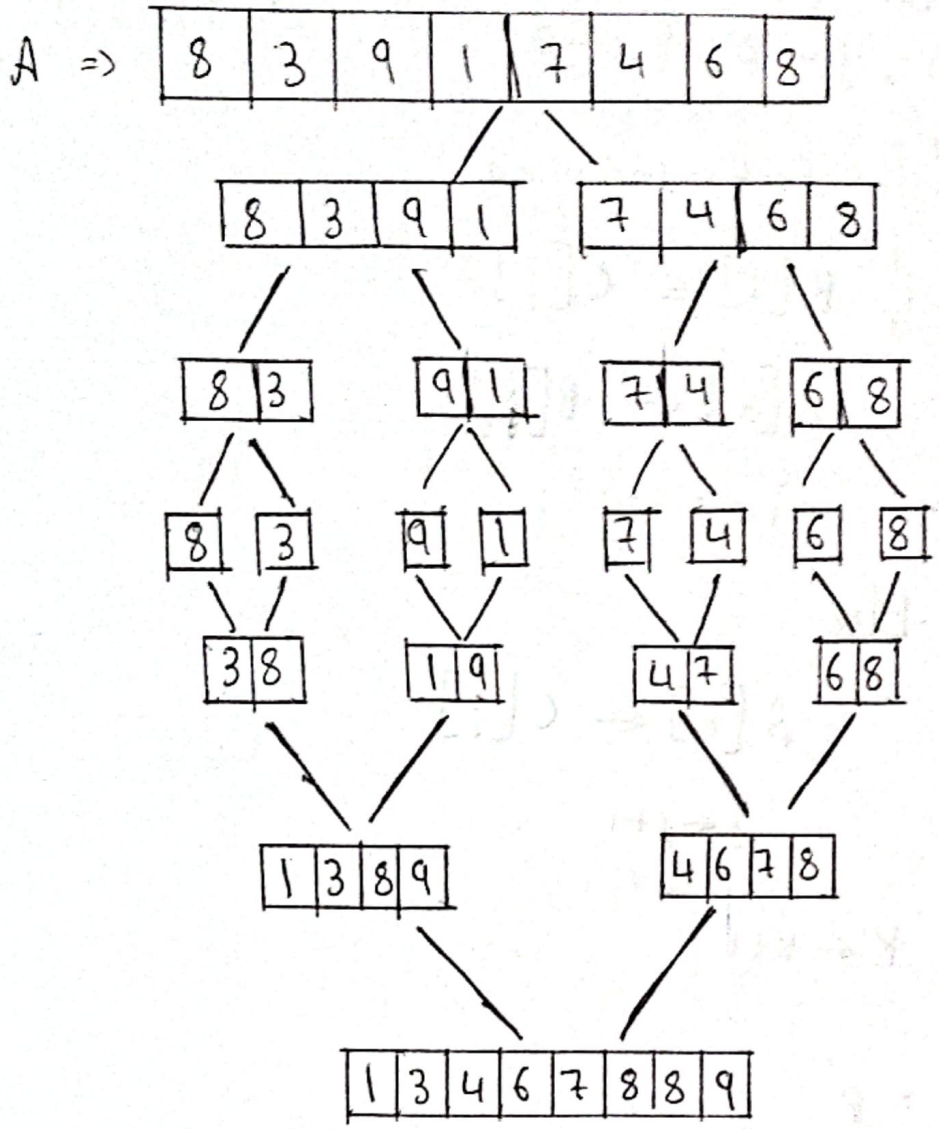
$$C_{Best}(n) = 1$$

→ Average Case :-

$$C_{avg}(n) \approx \log n$$

⇒ Merge Sort :-

Sorts given array by dividing a single array into 2 parts and then reccursively sorts the array and combines the 2 sorted arrays to form a single array.



Algorithm :-

```
MergeSort( A[0.... n-1])
{
    if n>1
    copy (A[0.... ⌊n/2⌋ - 1] to B[0.... ⌊n/2⌋ -1])
    copy (A[⌊n/2⌋.... n-1] to c[0.... ⌈n/2⌉ -1])
    MergeSort (B[0.... ⌈n/2⌉ -1])
    MergeSort (C[0.... ⌊n/2⌋ -1])
    Merge (B, C, A)
}
```

```
Merge(B[0...p-1], C[0...q-1], A[0..p+q-1])
{  i ← 0 , j ← 0, K ← 0

    while i < p and j < q
      if B[i] ≤ C[j]
         A[K] ← B[i]
          i ← i+1
      Else
          A[K] ← C[j]
           j ← j+1
      K ← K+1

    if i = p
      copy(C[j... q-1] to A[K.... p+q-1])
    Else
      copy(B[i.... p-1] to A[K.... p+q-1])

}
```

Merge Sort is not an Inplace Algorithm but it is a Stable Algorithm.

Input Size ⇒ n

Basic Operation ⇒ Comparison.

Recurrence Relation,

$C(1) = 0$

$$C(n) = 2C(n/2) + C_{merge}$$

$$C_{mergeworst}(n) = n-1$$

$$C(n) = 2C(n/2) + n - 1$$

$$n = 2^k$$

$$C(2^k) = 2C(2^{k-1}) + 2^k - 1$$

$$C(2^k) = 2\left(2C(2^{k-2}) + 2^{k-1}\right) + 2^k - 1$$

$$= 2^2 C(2^{k-2}) + 2^k - 2 + 2^k - 1$$

$$= 2^2\left[2C(2^{k-3}) + 2^{k-2} - 1\right] + 2\cdot 2^k - 2 - 1$$

$$= 2^3 C(2^{k-3}) + 2^k - 2^2 + 3\cdot 2^k - 2 - 1$$

$$= \vdots$$

$$= \vdots$$

$$= 2^i C(2^{k-i}) + i\cdot 2^k - 2^{i-1} - 2^{i-2} - \dots$$

$$= 2^i C(2^{k-i}) + i\cdot 2^k - (2^i - 1)$$

Put $i = k$

$$= 2^k \overset{0}{\cancel{C(1)}} + 2^k \times k - (2^k - 1)$$

$$= k2^k - 2^k + 1$$

$$= 2^k(k-1) + 1$$

$$2^K = n$$

$$\log_2 n = K$$

$$\therefore \quad C(n) = n(\log_2 n - 1) + 1$$

$$\Rightarrow \quad C(n) \in \theta(n \log_2 n)$$

By Master's Theorem,

$$a = 2, \; b = 2 \qquad d = 1$$

$$a = b^d \Rightarrow 2 = 2$$

$$\therefore \quad C(n) \in \theta(n \log_2 n) //$$

For Best Case :-

$$C(n) = 2C(n/2) + C_{merge}$$

$$C_{merge\,Best} = \frac{n}{2}$$

$$\therefore \quad C(n) \in \theta(n \log_2 n) // \quad \text{(By Master Theorem)}$$

$$C(n) = 2C(n/2) + \frac{n}{2} \qquad C(1) = 0$$

$n = 2^K$ (By Smoothness Theorem)

$$C(2^K) = 2C(2^{K-1}) + 2^{K-1}$$

$$C(2^K) = 2\left[2\,C(2^{K-2}) + 2^{K-1}\right] + 2^{K-1}$$

$$= 2^2 C(2^{K-2}) + 2^{K-1} + 2^{K-1}$$

$$\vdots$$

$$= 2^i C(2^{K-i}) + i \times 2^{K-1}$$

Put $i = K$

$$= 2^K \cancel{C(2^{K-K})}^{0} + K \times 2^{K-1}$$

$$= K\,2^{K-1}$$

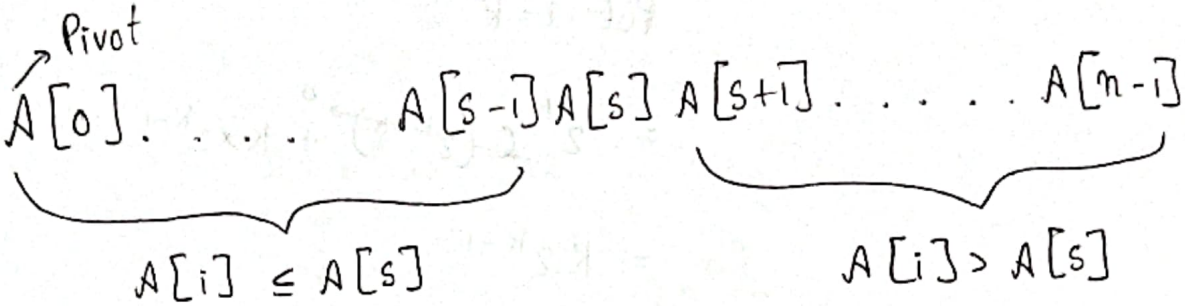
$$\therefore \quad C(n) = \frac{\log_2 n\,(n)}{2}$$

$$\Rightarrow \quad C(n) = \frac{n \log_2 n}{2}$$

$$\Rightarrow \quad C(n) \in \Theta(n \log_2 n)$$

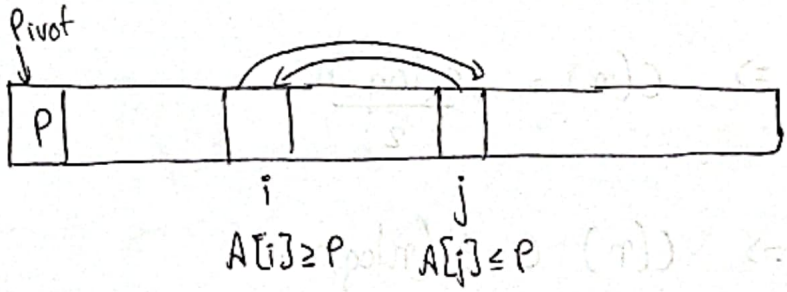
⇒ Quick Sort :- (Inplace but not Stable)

It rearranges the elements of a given array to achieves a partition / position 's' such that all values before A[s] are less than A[s] and all elements to right are greater than A[s].

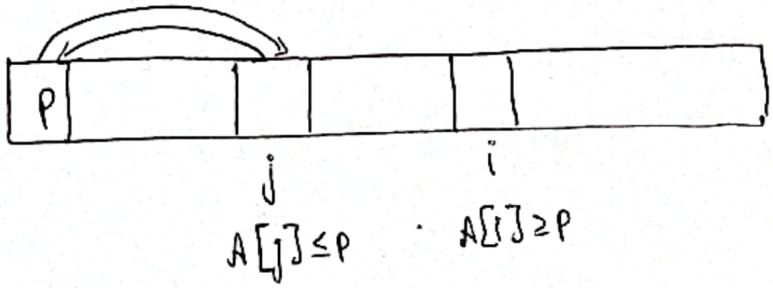


→ Three cases while scanning :- (2 Scans L→R & R→L)

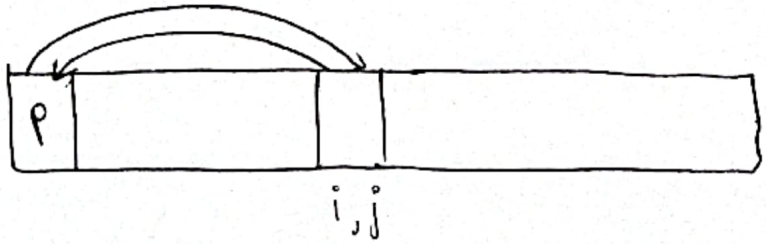
Case (i) :-

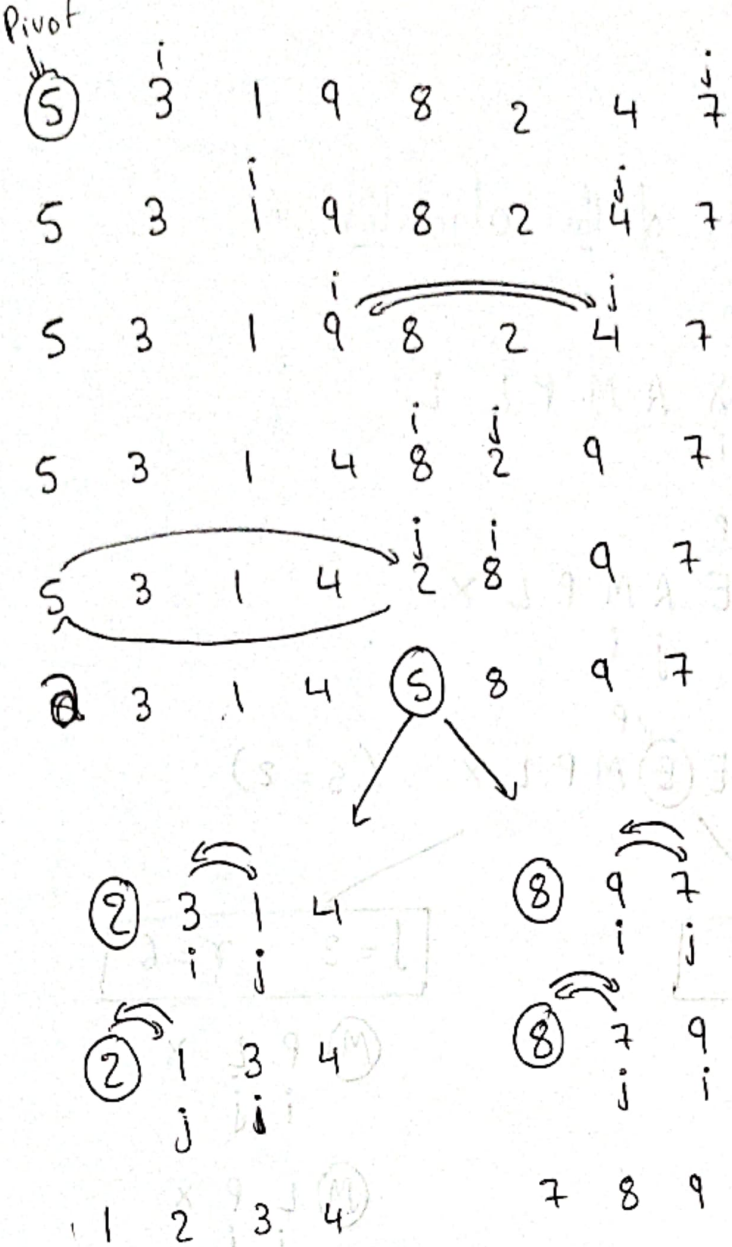


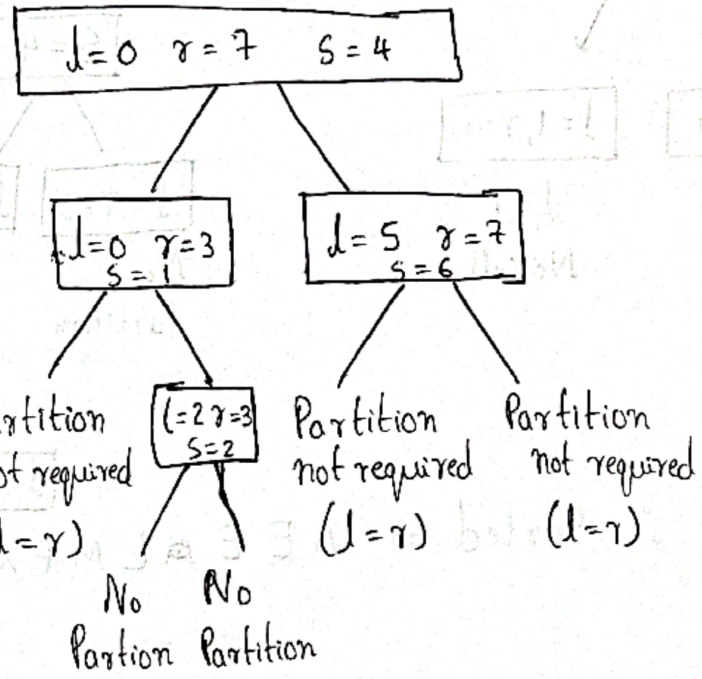
Case (ii) :-



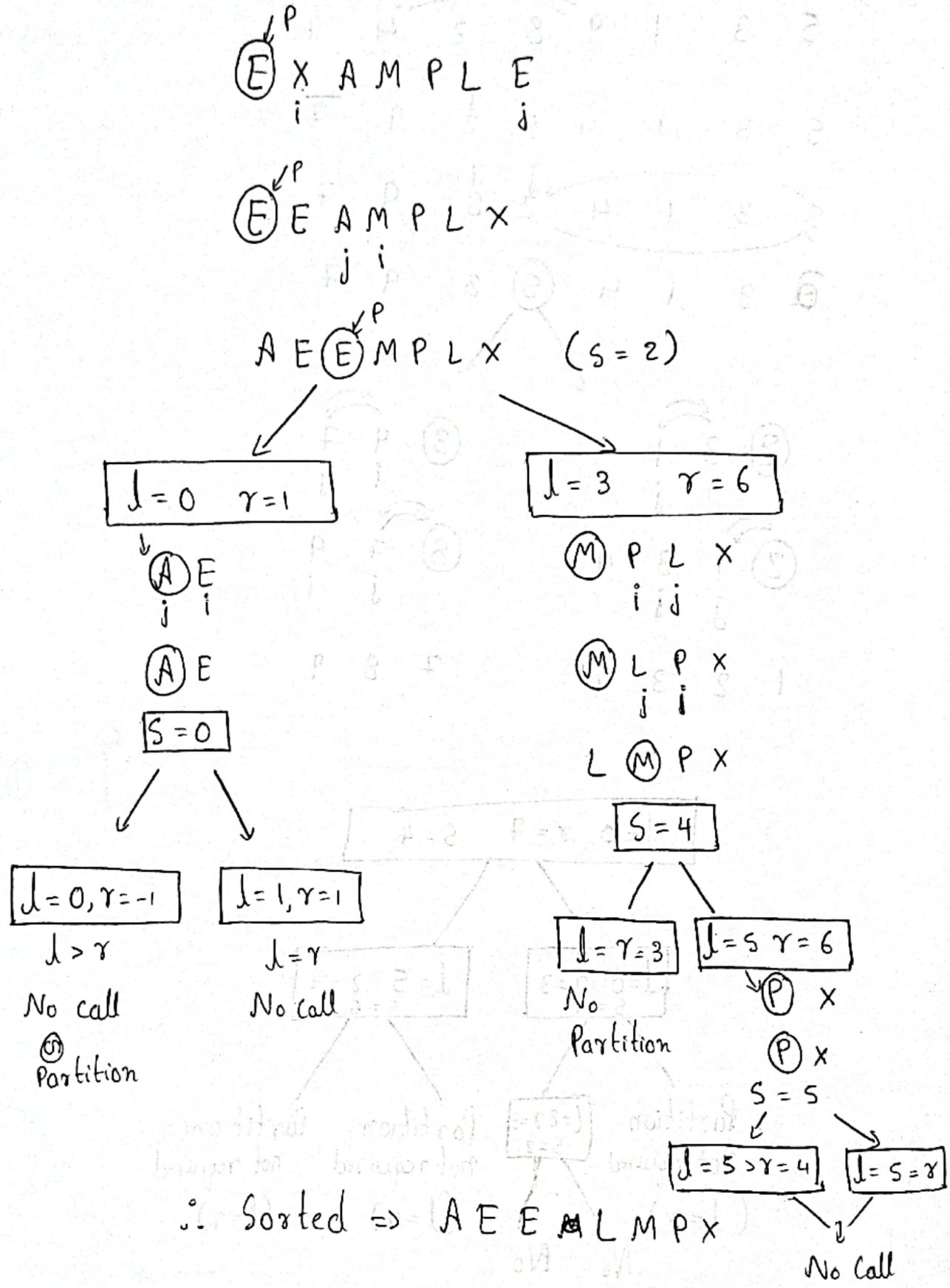
Case (iii) :-




Pivot
5 3 1 9 8 2 4 7
5 3 1 4 8 2 9 7
5 3 1 4 2 8 9 7
2 3 1 4 5 8 9 7
2 3 1 4
2 1 3 4
8 9 7
8 7 9
7 8 9
1 2 3 4


l=0 r=7 s=4
l=0 r=3 s=1
l=5 r=7 s=6
Partition not required (l=r)
l=2 r=3 s=2
Partition not required (l=r)
Partition not required (l=r)
No Partion
No Partition

⇒ To verify stability of the algorithm :-



Algorithm :-

Quicksort (A[l .... r])

// I/P :- A subarray A[l ... r] of A[0 ... n-1] defined by its left and right indices l & r

// O/P :- Sorted subarray A[l ... r]

```
if l < r
    s ← partition (A[l ... r])
    Quicksort (A[l .... s-1])
    Quicksort (A[s+1 .... r])
```

Partition (A[l .... r])

// I/P :- A subarray A[l ... r] of A[0 ... n-1] defined by its left and right indices l and r

// O/P :- A partition of A[l .... r] with the split position returned as its return value

```
p ← A[l] , i ← l , j ← r+1
repeat
    repeat i ← i+1
    until A[i] ≥ p
    repeat j ← j-1
    until A[j] ≤ p
```

```
swap (A[i], A[j])
    until (i ≥ j)
swap (A[i], A[j])
swap (A[l], A[j])
return j
```

Analysis :-

(i) Input Size => n

(ii) Basic Operation => Comparison

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

Best Case Input :-

(2) 2 2 2 2 2 2
i j

(2) 2 2 2 2 2 2
i j

(2) 2 2 2 2 2 2
i j

(2) 2 2 2 2 2 2
j i

2 2 2 (2) 2 2 2
↑
Pivot

Here, $f(n) = n$

$\therefore \quad C(n) = 2C\left(\frac{n}{2}\right) + n$

$n = 2^k$

$C(2^k) = 2\left(C(2^{k-1}) + 2^k\right.$

$C(2^k) = 2\left(2C(2^{k-2}) + 2^{k-1}\right) + 2^k$

$C(2^k) = 2^2 C(2^{k-2}) + 2 \times 2^{k-1} + 2^k$

$C(2^k) = 2^2 \left[2C(2^{k-3}) + 2^{k-2}\right] + 2^k + 2^k$

$C(2^k) = 2^3 C(2^{k-3}) + 2^k \times 3$

$\vdots$

$C(2^k) = 2^i C(2^{k-i}) + 2^k \times i$

$C(2^k) = 2^k \cancel{C(1)}^{0} + k \times 2^k$

$C(2^k) = k2^k$

$\Rightarrow C(n) = n \log n$

Worst Case Input:- Array is either sorted in Ascending (or) Descending order.

1 2 3 4 5 6 7 8 $\Rightarrow 9 \quad (n+1)$

j i

1 2 3 4 5 6 7 8 $\Rightarrow 8 \quad (n)$

j i

1 2 3 4 5 6 7 8 $\Rightarrow 7 \quad (n-1)$

j i

1 2 3 4 5 6 7 8 $\Rightarrow 6$

j i

1 2 3 4 5 6 7 8 $\Rightarrow 5$

j i

1 2 3 4 5 6 7 8 $\Rightarrow 4$

j i

1 2 3 4 5 6 7 8 $\Rightarrow 3$

j i

$$\therefore \quad C_{worst}(n) = n+1+n+n-1+\ldots+3$$

$$= \frac{(n+1)(n+2)}{2} - 3$$

$$\therefore C_{worst}(n) \in \Theta(n^2)$$

Ex:- Sort MERGESORT using Quicksort

M E R G E S O R T
  i i   j

M E E G R S O R T
    j i

G E E M R S O R T

G E E
  j

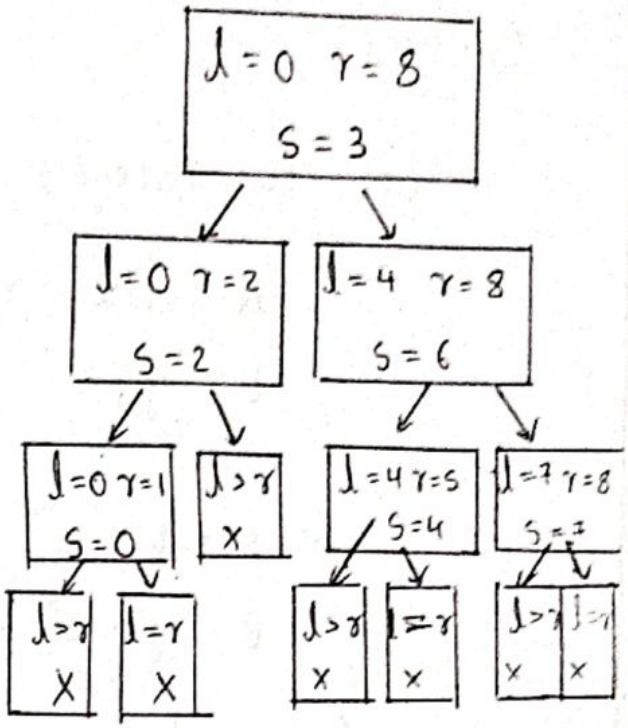
E E G

R S O R T
  i   j

R O S T
  j i

R R S T

O R S T
j i



⇒ Sorted Array :- EEGMORRST

⇒ Multiplication of Two Large Integers using Divide and Conquer :-

Let two integers

$$A : \overset{n/2}{w}\overset{n/2}{x} = n$$

$$B : yz$$

$$\Rightarrow A = w \times 10^{n/2} + x$$

$$B = y \times 10^{n/2} + z$$

$$A \times B = \left[(w \times 10^{n/2}) + x\right] \times \left[(y \times 10^{n/2}) + z\right]$$

$$= wy \times 10^{n} + (wz + xy) \times 10^{n/2} + xz \quad \text{(4 Multiplications)}$$

Basic operation :- Multiplication

$$M(n) = 4\,M(n/2) \qquad M(1) = 1$$

$$n = 2^k$$

$$M(2^k) = 4\,M(2^{k-1})$$

$$= 4\left(4\,M(2^{k-2})\right)$$

$$= 2^4\,M(2^{k-2})$$

$$= 2^{2i}\,M(2^{k-i})$$

$$M(2^k) = 4^i\,M(2^{k-i})$$

Put $i = k$

$$M(2^k) = 4^k\,M(2^{k-k})$$

$$M(2^k) = (2^k)^2$$

$$\boxed{M(n) = n^2}$$

The middle term $(wz + xy)$ can also be expressed as

$$E = (wz + xy) = (w+x) * (z+y) - wy - xz$$

$$E = (w+x) * (z+y) - C - D \quad (C = wy,\ D = xz)$$

$$\therefore A \times B = C \times 10^n + E \times 10^{n/2} + D \quad (3 \text{ Multiplication})$$

Now,

$$M(n) = 3\left(M(n/2)\right) \quad M(1) = 1$$

$$n = 2^k$$

$$M(2^k) = 3\left(M(2^{k-1})\right)$$

$$= 3\left(3M(2^{k-2})\right)$$

$$\vdots$$

$$= 3^i M(2^{k-i})$$

Put $i = k$

$$\therefore M(2^k) = 3^k$$

$$M(n) = 3^{\log_2 n}$$

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

$$\boxed{M(n) = n^{1.585}}$$

Ex:- $A = 23,\ B = 14$

$A = 2 \times 10 + 3 \Rightarrow w = 2 \quad x = 3$

$B = 1 \times 10 + 4 \Rightarrow y = 1 \quad z = 4$

$C = wy = 2 \times 1 = 2$

$D = xz = 3 \times 4 = 12$

$E = (w+x) \times (y+z) - C - D$

$= (2+3) \times (1+4) - 2 - 12$

$= 25 - 14$

$E = 11$

$n = 2$

$\therefore A \times B = C \times 10^{n} + E \times 10^{n/2} + D$

$= 2 \times 10^{2} + 11 \times 10 + 12$

$= 200 + 110 + 12$

$= 322$

$= \underline{322}$

⇒ Strassen's Matrix Multiplication :-

* The number of basic operations required to multiply 2 2x2 matrices using Brute Force Technique is 8

* But, in divide and conquer method, it can be reduced to 7, this was proposed by V. Strassen (1969)

The formula used for this :

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Where,

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$

$$m_2 = (a_{10} + a_{11}) * b_{00}$$

$$m_3 = a_{00} * (b_{01} - b_{11})$$

$$m_4 = a_{11} * (b_{10} - b_{00})$$

$$m_5 = (a_{00} + a_{01}) * b_{11}$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

⇒ Analysis :-

$$M(n) = 7M(n/2) \quad \text{for } n>1 \quad M(1)=1$$

$$n = 2^k$$

$$M(2^k) = 7M(2^{k-1})$$

$$= 7(7M(2^{k-2}))$$

$$= 7^2 M(2^{k-2})$$

$$= 7^3 M(2^{k-3})$$

$$\vdots$$

$$= 7^i M(2^{k-i})$$

Put $i = k$

$$= 7^k M(2^{k-k})$$

$$= 7^k$$

$$\therefore M(n) = 7^{\log_2 n}$$

$$= n^{\log_2 7}$$

$$M(n) \approx n^{2.807}$$

This is smaller than n^3 obtained in Brute force Techniq

$M(n) \in \theta(n^{\log_2 7})$ [Strassen's Method]

$M(n) \in \theta(n^3)$ [Brute Force Technique]