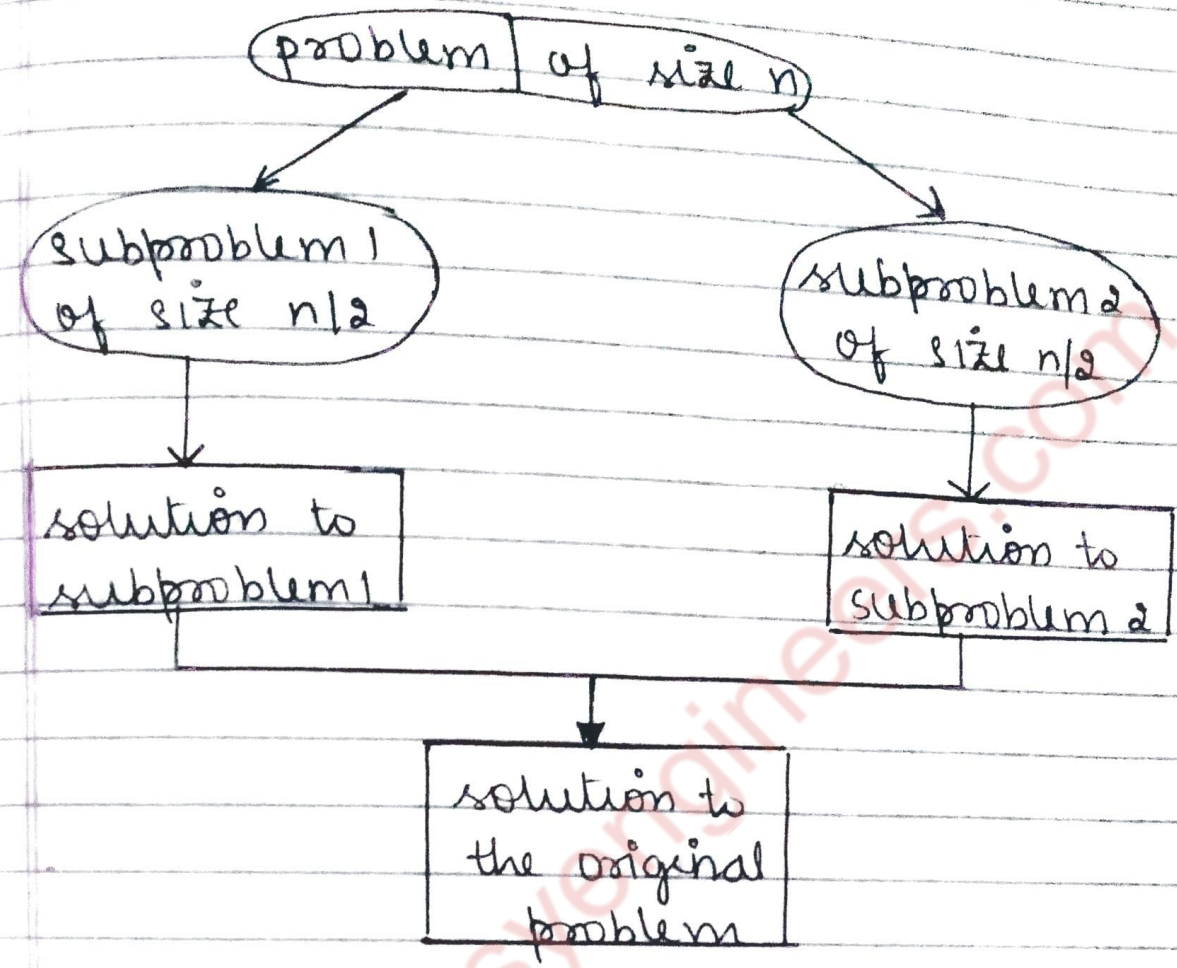
MODULE - 2

DIVIDE AND CONQUER

* General method for divide & conquer techniques :- Divide & conquer is one of the best-known general algorithm design technique. It works according to the following general plan:

1. Given a function to compute on = n' inputs the divide-and-conquer strategy suggests splitting the inputs into = k' distinct subsets, 1<k<=n, yielding = k' sub problems.
2. These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
3. If the sub problems are still relatively large, then the divide-&-conquer strategy can possibly be reapplied.
4. Often the sub problems resulting from divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm.
5. Now smaller & smaller subproblems of the same kind are generated until eventually subproblems that are small enough be solved without splitting are produced.
6. A typical case with k = 2 is diagrammati

-ally shown below



* <u>control abstraction of Divide & conquer:-</u>
A general divide & conquer design strategy (control abstraction) is illustrated as given below -

```
ALGORITHM DAnd C (P)
{
  if small (P) then return S(P) // termination
condition else
{
  divide P into smaller instances P1, P2, P3
... Pk  k ≥ 1 ; 1 ≤ k ≤ n
Apply DAnd C to each of these subproblems
return Combine (DAnd C (P1), DAnd C (P2),
  DAnd C (P3) ... DAnd C (Pk));
```

```
    }
}
```

In the above specification,

1. Initially DAndC(P) is invoked, where =P' is the problem to be solved.
2. Small(P) is a Boolean valued function that determines whether the input size is small enough that the answer can be computed without splitting. If small(P) is true then function 'S' is invoked.
3. Otherwise the problem =P' is divided into sub problems. These sub problems are solved by recursive application of DAndC.
4. Finally the solution from k sub problems is combined to obtain the solution of the given problem using combine function.

* **Binary Search**

Let a_i, $1 \le i \le n$ be a list of elements that are sorted in non-decreasing order.

The problem is to find whether a given element x is present in list or not. If x is present we have to determine a value j (element's position) such that $a_j = x$. If x is not in the list, then j is set to 0.

Recursive binary search algorithm

```
//Given an array A[low:high] of
elements in non-decreasing order.
```

$1 \le low \le high$

//Determine whether x is present, and if so return j such that x = A[j]; else return 0.

```
{
    if (low = high) then
  {
   if (x = A [high] then return h; // if small
    (P) else return 0;
  }
else
 { // reduce P into smaller sub problem.
    mid = [(low + high) / 2];
 if (x = A [mid]) then return mid;
 else if (x < A [mid]) then
          return BinSearch (A, low, mid-1, x);
   else return BinSearch (A, mid+1, high, x);
 }
```

Explanation: The problem is subdivided into smaller problems until only 1 single element is left out.

1. If low = high then it means there is only one ~~signed~~ single element. So compare it with the search element = x'. if both are equal then return the index, else return 0.
2. If low ≠ high then divide the problem into smaller subproblems.
   a. Calculate mid = (low + high)/2
   b. Compare x with element at mid if both are equal the return index mid.

3. Else if x is less than A[mid] then the element x would be in the first partition A[low: mid-1]. So make a recursive call to BinSearch with low as low and high as mid-1.
4. Else x is greater than A[mid] then the element x would be in the second partition A[mid+1 : high]. So make a recursive call to BinSearch with low as mid+1 and high as high.

* **Merge Sort**

Merge Sort is a perfect example of a successful application of the divide & conquer technique.

It sorts a given array A[1]....A[n] by dividing it into 2 halves A[1]... A[n/2] and A[n/2 +1]...A[n], sorting each of them recursively, then merging the 2 smaller sorted arrays into single sorted one.

```
Algorithm Merge sort (a, low, high)
// a (low: high) is array to be sorted
// small(p) is true if there is only 1
element
// to sort, which means the list is
already sorted
{
  if (low < high) then // if there is
{                          more than 1 element
```

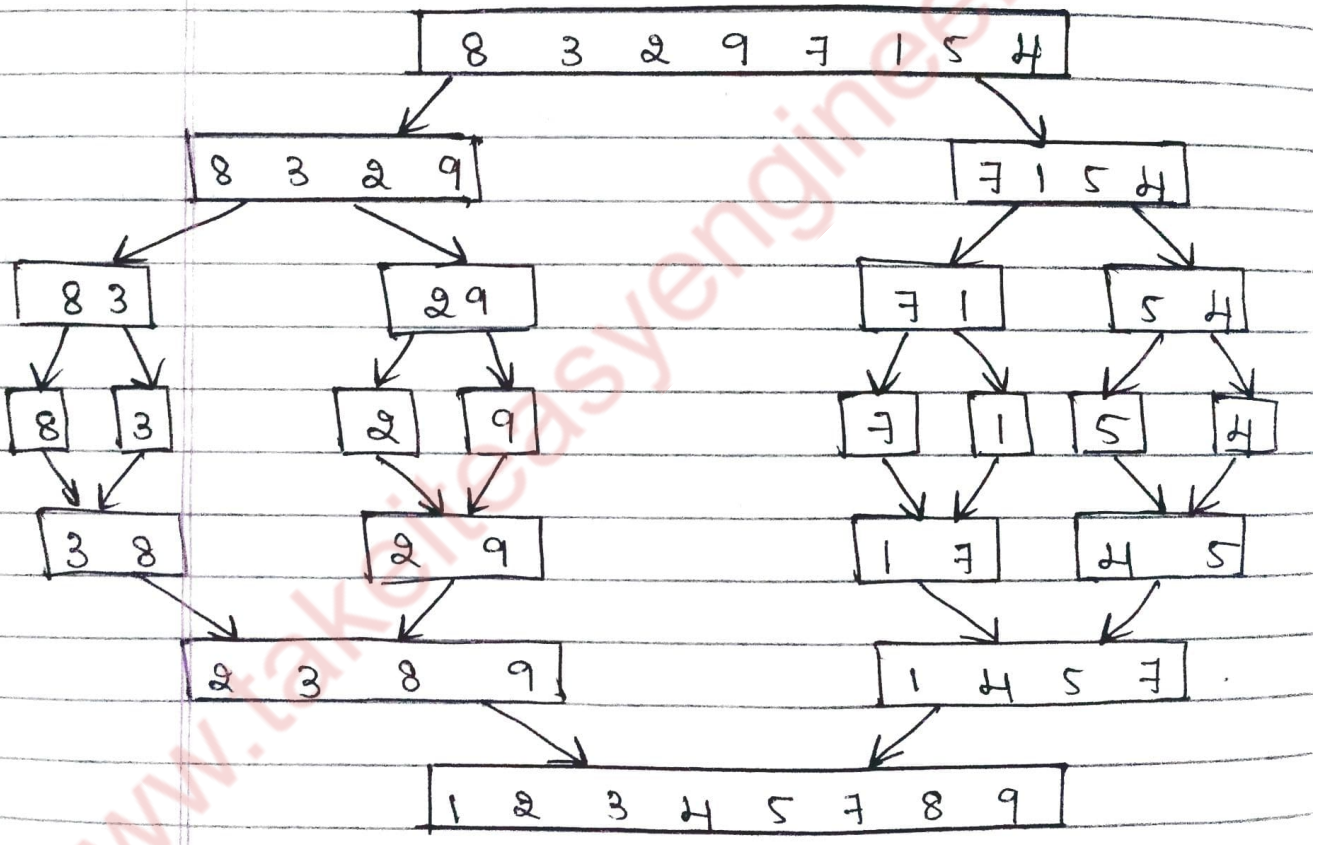
```
//Divide p into sub problems
        mid = [low+high)/2;
  //solve the sub problems
        Merge sort (a, low, mid);
        Merge sort (a, mid+1, high);
  //combine solutions
        Merge (a, low, mid, high)
    }
}
```

1. The merging of 2 sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged
2. The elements pointed to compared, & the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.
3. This operation is repeated until one of the 2 given arrays is exhausted, & then the remaining elements of other array are copied to end of new array.

Advantages :- * For large n, the no. of comparisons made by this algorithm in the average case turns out to be about 0.25 n less & hence is also in O(n log n).
* Merge sort is a stable algorithm.

Limitations: The principal shortcoming of mergesort is linear amount [O(n)] of extra storage the algorithm requires. Though merging can be done in-place, the resulting algorithm is quite complicated & of theoretical interest only.

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 as follows



Analysis: * Here the basic operation is key comparison. As merge sort execution does not depend on the order of the data, best case and average case runtime are the same as worst case runtime.

* Assuming for simplicity that total no. of elements n is a power of 2, the recurrence relation for the no. of key

comparisons C(n) is
Where, Cmerge(n) is the number of key comparison made during the merging stage.

* Let us analyze Cmerge(n), the no. of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the 2 arrays still needing to be processed is reduced by 1.

In the worst case, neither of the 2 arrays becomes empty before the other one contains just 1 element. Therefore, for the worst case,

$$C_{merge}(n) = n-1.$$

Now, $C_{worst}(n) = 2\,C_{worst}(n/2) + n-1$ for $n>1$

$$C_{worst}(1) = 0.$$

Solving the recurrence equation using master theorem: Here $a=2$, $b=2$, $f(n)=n$, $d=1$. Therefore $2=2^1$, case 2 holds in the master theorem

$$C_{worst}(n) = \Theta(n^d \log n) = \Theta(n^1 \log n)$$
$$= \Theta(n \log n)$$
$$\therefore C_{worst}(n) = \Theta(n \log n)$$

Quick Sort :

Quick sort is the other important sorting algorithm that is based on the divide and conquer approach in which divides its input elements according to

value

their ~~position~~ in the array,

It is the arrangement of the array elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it

A[0]...A[s-1] A[s] A[s+1]...A[n-1]

all are ≤ A[s] all are ≥ A[s]

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

//Sorts a subarray by quicksort
//Inputs: Subarray of array A[0..n-1], defined by its left & right
//indices l and r
//Output: Subarray A[l...r] sorted in nondecreasing order

```
if (l < r) then
    s ← Partition(A[l..r]) //s is split position
    Quicksort(A[l..s-1])
    Quicksort(A[s+1..r])
end if
```

Algorithm partition(A[l...r])
//Input: An sub-array A[l..r] of A[0...n-1]
//Output: partition position j

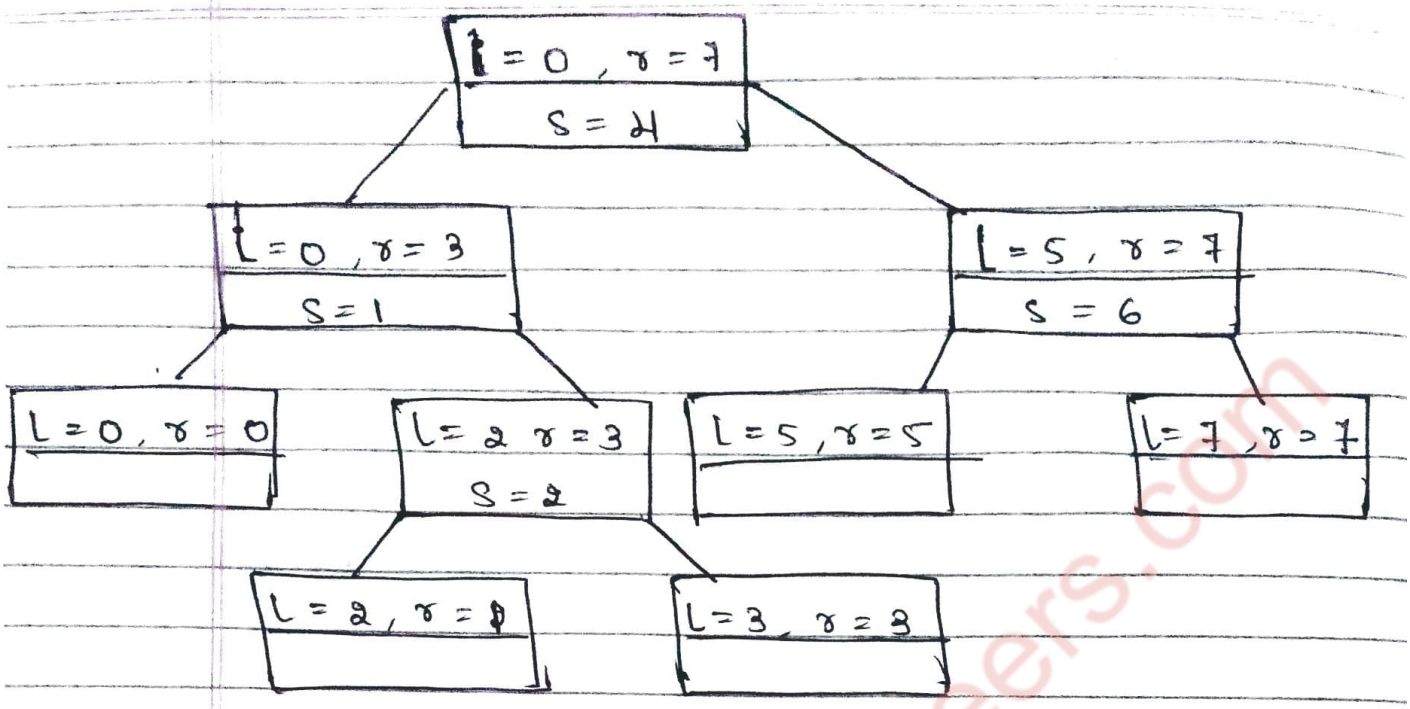
```
p ← A[l]
i ← l+1
j ← r
repeat
    repeat i ← i+1 until A[i] ≥ p
    repeat j ← j+1 until A[j] ≤ p
swap (A[i], A[j]) // last swap is invalid
until i > j
swap (A[i], A[j]) // to avoid invalid swap
swap (A[l], A[j])
    return j
```

(a)

0	1	2	3	4	5	6	7
[5]	3	1	9	8	2	4	7
[5]	3	1	9	8	2	4	7
[5]	3	1	[4]	8	2	9	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]	1	3	4	5	[8]	9	7
1	2	3	4	5	[8]	9	7
1	2	3	4	5	[8]	7	9
1	2	3	4	5	7	[8]	9

The above figure is all about the Array transformations with pivots shown in box

(b) Tree recursive calls to Quicksort with input values l & r of subarray bounds & split positions of partition obtained is as:



Analysis

1. Best case: here the basic operation is key comparison. Number of key comparisons made before a partition is achieved is n+1 if the scanning indices across & n if they coincide.

According to Master theorem,
$C_{(best)}(n) \in O(n \log_2 n)$

Solving it exactly for $n = 2^k$ yields

$$C_{best}(n) = n \log_2 n$$

2. Worst case: In worst case, all the splits will be skewed to extreme: one of the 2 subarrays will be empty, & the size of the other will be just 1 less than the size of subarray being

partitioned. The total number of key comparisons made will be equal to

$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in O(n^2)$

3. Average case: Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on randomly ordered array of size n.

$$C_{avg}(n) = \frac{1}{n} \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)]$$

$C_{avg}(0) = 0$, $C_{avg}(1) = 1$

Limitations: * It is not stable.
* It requires a stack to store parameters of subarrays that are yet to be sorted.

Strassen's matrix multiplication:

matrix multiplication

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

Normal matrix multiplication.

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

In strassen's matrix

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

Where,

$m_1 = (A_{11} + A_{22}) \times (B_{11} * B_{22})$

$m_2 = (A_{21} + A_{22}) \times B_{11}$

$m_3 = A_{11} \times (B_{12} - B_{22})$

$m_4 = A_{22} \times (B_{21} - B_{11})$

$m_5 = (A_{11} + A_{12}) \times B_{22}$

$m_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$

$m_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$

Analysis: Here the basic operation is multiplication.

M(n) is no. of multiplications made by Strassen's algorithm.

$M(n) = 7M(n/2)$ for $n>1$, $M(1)=1$

Since $n = 2^k$

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

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

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

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

Since $k = \log_2 n$

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

$\approx n^{2.807}$

$M(n) = \Theta(n^{2.807})$

Example: $A = \begin{bmatrix} 2 & 3 \\ 1 & 4 \end{bmatrix}$ $B = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$

$m_1 = (2+4) \times (1+4) = 30$

$m_2 = (1+4)1 = 5$

$m_3 = 2(2-4) = -4$

$m_4 = 4(3-1) = 8$

$m_5 = (2+3)4 = 20$

$m_6 = (1-2) \times (1+2) = -3$

$m_7 = (3-4) \times (3+4) = -7$

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

$$= \begin{bmatrix} 30 + 8 - 20 + (-7) & (-4) + 20 \\ 5 + 8 & 30 - 4 - 5 - 3 \end{bmatrix}$$

$$= \begin{bmatrix} 11 & 16 \\ 13 & 18 \end{bmatrix}$$

Finding Maximum and minimum

Algorithm based on Divide & Conquer

```
// a[1:n] is global array. parameters i
 & j are integers,
// 1 ≤ i ≤ j ≤ n. The affect is to set max
   and min to the
// largest & smallest values in a[i:j]
 respectively.
{
if (i = j) then max := min := a[i]; //Small(P)
else if (i = j - 1) then // Another case of small
                            (P)
  {
    if (a[i] < a[j]) then
    {
      max := a[j];
      min := a[i];
    }
else
```

```
    {
        max := a[i];
        min := a[j];
    }
}
else
{ // If P is not small, divide P into
      subproblems.
  // Find where to split the set
      mid := [(i+j)/2];
  // Solve the subproblems
    MaxMin (i, mid, max, min);
    MaxMin (mid+1, j, max1, min1);
  // Combine the solutions
    if (max < max1) then max := max1;
    if (min > min1) then min := min1;
  }
}
```

Analysis

$$T(n) = \begin{cases} T([n/2]) + T([n/2]) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

where n is power of 2, $n = 2^k$ for some positive integer k

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &= 2^{k-1} T(2) + \sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 \\ &= 3n/2 - 2 = O(n) \end{aligned}$$

Advantages & Disadvantages of Divide and Conquer.

Advantages : 1. **Parellelism :** Once the division phase is complete, the sub-problem are usually independent & can therefore be solved in parallel. This approach typically generates more enough concurrency to keep the machine busy & can be adapted for execution in multi-processor machines.

2. Solves difficult problems with ease.

3. **Cache performance :** Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.

Disadvantages : 1. It uses recursion that makes it a little slower & if a little error occurs in the code the program may enter into an infinite loop.

2. Usage of explicit stacks may make use of extra space.

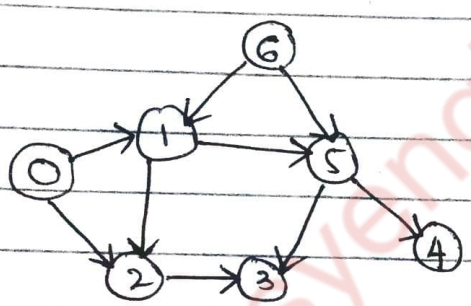
3. If performing a recursion for no. times greater than the stack in the CPU then the system may crash.

Topological sort

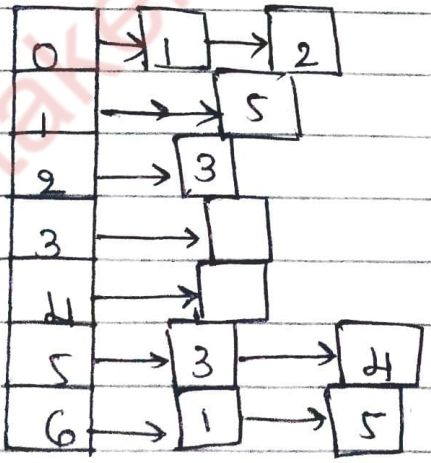
It is a linear ordering of vertices such that for every directed edge u v vertex u comes before v in ordering

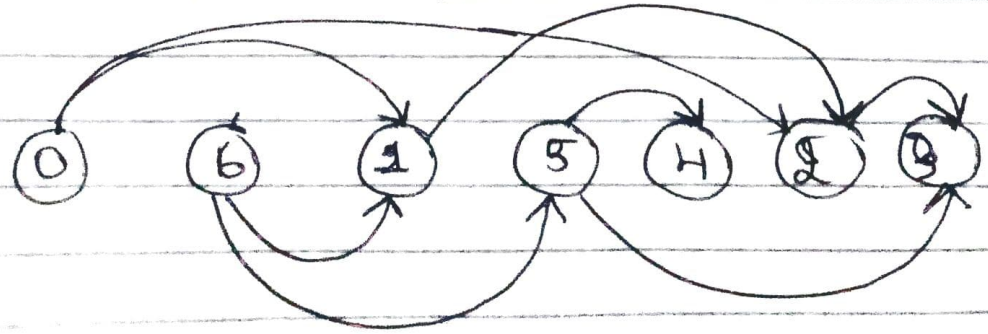
Topological sort can be solve in 2 ways:

1. DFS method

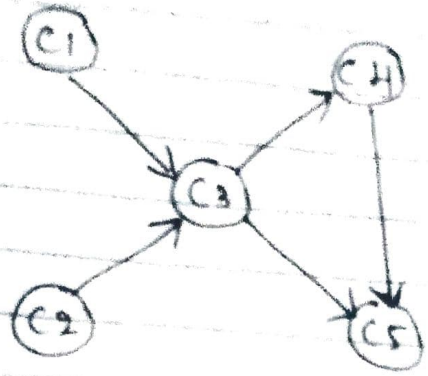


adjacency list

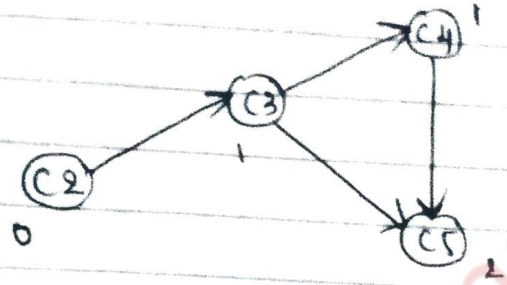




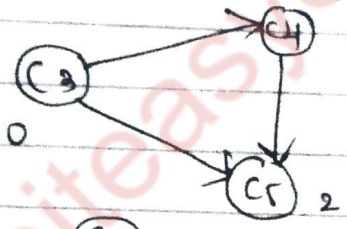
2. Source removal method



→ Remove C1



→ Remove C2



→ Remove C3

→ Remove C4

C5

→ Remove C5

then topological sort