USN | | | | | | | | | | |

CMRIT

Internal Assessment Test 1 – Apr 2026

Sub:	**Analysis and Designs of Algorithms**					Sub Code:	BCS401	Branch:	ISE	
Date:	15-04-2026	Duration:	90 min's	Max Marks:	50	Sem/Sec:	IV / A,B,C			OBE

1. Explain the algorithm design and analysis process. Illustrate the various steps involved in algorithmic problem solving with a neat flow diagram. (10M CO1 L2)

Scheme:
Diagram – 4M
Six points – Each 1 M

An algorithm is a finite sequence of unambiguous instructions for solving a problem, i.e for obtaining a required output for any legitimate input in a finite amount of time.

A Sequence of steps one typically goes through in designing and analyzing an algorithm:

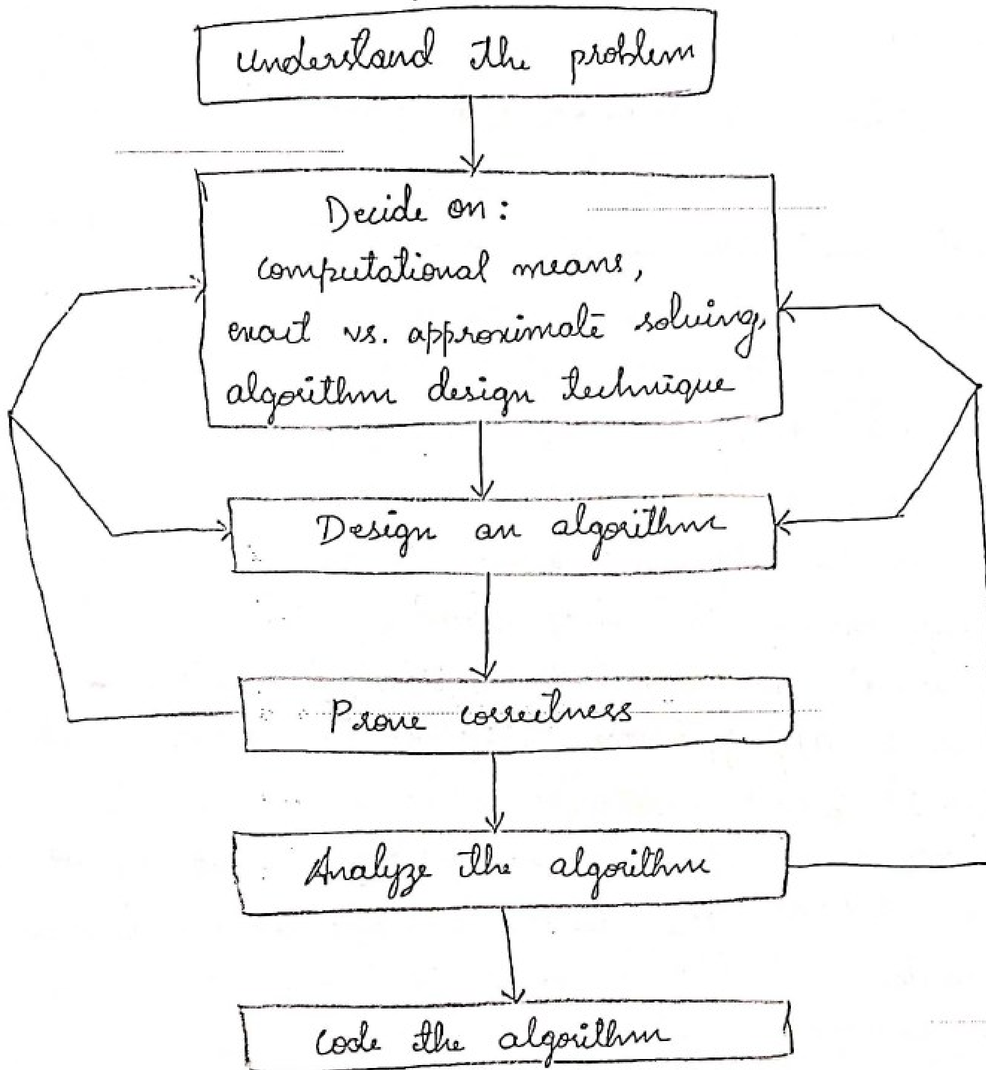


Fig : Algorithm design and analysis process.

We can consider algorithms to be procedural solutions to problems.

1) Step 1: Understanding the Problem

From a practical perspective, the 1st thing we need to do before designing an algorithm is to understand completely the problem given. Read the problem's description carefully and ask questions if any doubts about the problem, do a few small examples by hand, think about special cases, & ask questions again if needed.

Often we will not find a readily available algorithm and will have to design our own.

An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle. If we fail to do this, our algorithm may work correctly for a majority of inputs, but crash on some "boundary" value.

A correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

2) Step 2: Decide on -

- Computational means

Once we completely understand a problem, we need to ascertain the capabilities of the computational device the algorithm is intended for.

The vast majority of algorithms in use today are still destined to be programmed for a computer closely resembling the von Neumann machine.

The essence of this architecture is random-access machine (RAM). Its central assumption is that instructions are executed one after another, one operation at a time. Accordingly, algorithms designed to be executed on such machines are called Sequential algorithms.

The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e in parallel. Algorithms that take advantage of this capability are called Parallel algorithms.

The speed and memory available on a particular computer system are other factors to consider.

3) Exact vs Approximate Problem Solving

The next decision is to choose between solving the problem exactly or solving it approximately.

An algorithm which solves the problems exactly is called an exact algorithm. (Precise solution)

An algorithm which solves the problem approximately is called an approximation algorithm. (approximate solution)

Why opt for an approximation algorithm?

1) There are important problems that simply cannot be solved exactly for most of their instances.

Egs: extracting square roots, solving non linear equations, evaluating definite integrals.

2) Available exact algorithms are unacceptably slow, because of the problem's intrinsic complexity.

3) An approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

4) • Algorithm Design Techniques

An algorithm design technique (or "Strategy" "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Design an algorithm to solve a given problem.

Algorithm design techniques make it possible to classify algorithms according to an underlying design idea, ∴ they can serve as a natural way to both categorize and study algorithms.

5) Step 3: Design an Algorithm (and Data Structures)

Designing an algorithm for a particular problem maybe a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined.

• Choosing data structures appropriate for the operations performed by the algorithm is important.

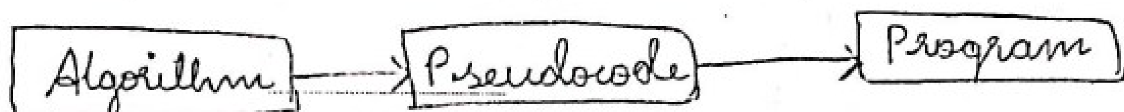
Algorithms + Data Structures = Programs.

In the new world of object-oriented programming, data structures remain crucially important for both design and analysis of algorithms.

Methods of Specifying an Algorithm

Once the algorithm is designed, it should be specified in some fashion. The two options that are most widely used for specifying algorithms : Pseudocode & Flowchart.

Pseudocode is a mixture of a natural language and programming language-like constructs.
Pseudocode is a step by step description of an algorithm.
Pseudocode is the intermediate state between an idea and its implementation (code) in a high-level language.



Flowchart is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.
Flowchart is a pictorial representation of flow of an algorithm.

Step 4: Proving an Algorithm's correctness

Once an algorithm has been specified, we have to prove its correctness, i.e prove that algorithm yields a required result for any legitimate input in a finite amount of time.

b) Design an algorithm that can handle a range of inputs that is natural for the problem at hand.
(i.e. set of inputs it accepts)

Step 6 : Coding an Algorithm

Algorithms ultimately has to be implemented as computer Programs. Programming an algorithm presents both peril and an opportunity.

The peril (threats) lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.

The validity of the program is established by testing and debugging programs, test and debug program throughly when an algorithm is implemented.

Code optimization may be required, such improvements can speed up a program (running time).

2. Explain the general plan for analyzing the efficiency of recursive algorithms. Develop a recursive algorithm to compute factorial of a positive integer and derive its efficiency. (10M CO1 L3)

2.4 Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, then the worst-case, average case and best case efficiencies must be investigated separately.
4. Setup a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

While solving the recurrence we will use the forward and backward substitution method. And then correctness of formula can be proved with the help of mathematical induction method.

Eg 1: Computing factorial of some number n

```
Algorithm Factorial(n)
// Computes n! recursively
// Input : A nonnegative integer n
// Output : The value of n!
if n==0 return 1
else return Factorial(n-1) * n
```

Scheme:
Five steps – 5M
Algorithm – 1M
Recurrence relation – 1M
Solution – 2M
Order of growth – 1M
(Solution to be given)

Mathematical Analysis:

Step 1: The factorial algorithm works for input size n

Step 2: The basic operation in computing factorial is multiplication.

Step 3: The recursive function call can be formulated as
$F(n) = F(n-1) * n$ where $n > 0$

Then the basic operation multiplication is given as $M(n)$.

$M(n)$ is multiplication count to compute factorial (n).

$M(n) = M(n-1) + 1$

These multiplications are required to compute factorial $(n-1)$ ... To multiply factorial $(n-1)$ by n

Step 4: In step 3 the recurrence relation obtained is
$M(n) = M(n-1) + 1$

Initial condition : $M(0) = 0$

Initial condition is obtained by looking at the condition that makes the recursive call to stop.

• The Recurrence relation $M(n)$ is given by:

$$M(n) = \begin{cases} 0 & \text{if } n = 0 \\ M(n-1)+1 & \text{otherwise} \end{cases}$$

• Backward Substitution :

$M(n) = M(n-1)+1$ —— ①

$M(n-1) = M(n-2) + 2$ —— ②

$M(n-2) = M(n-3) + 3$ —— ③

From the substitution methods we can establish a general formula as:

$$M(n) = M(n-i) + i$$

Now let us prove correctness of this formula using mathematical induction as follows:

From the initial condition, i.e $n=0$, substitute $i = n$ in the above equation to get

$$M(n) = M(n-i) + i$$
$$= M(n-n) + n$$
$$= M(0) + n$$
$$= 0 + n$$

$$M(n) = n$$

Thus the time complexity of factorial function is $\Theta(n)$.

3. Design a Quick Sort algorithm. Obtain its best case, average case and worst case efficiency. Apply the algorithm to sort a given list of elements. 5, 3, 1, 9, 8, 2, 4, 7. (10M CO2 L3)

Quick Sort:

* Quick Sort is the other important sorting algorithm that is based on the divide-and-conquer approach.

* Quick Sort divides its input elements according to their value.

* It is divided into partition. i.e A partition is an arrangement of the array's elements so that all the problem 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.

$$\underbrace{A[0] \cdots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \cdots A[n-1]}_{\text{all are } \geq A[s]}$$

* After the partition is achieved, A[s] will be in its final position in the sorted array and we can continue ~~saying~~ sorting the 2 subarrays to the left and to the right of A[s] independently.

Pseudocode of quicksort: call Quicksort (A [0 - - n-1])

Scheme:
2 Algorithms – 4M
Example workout – 3M
Recurrence relation – 1M
Order of growth – 2M (Solution to be given by master theorem)

Algorithm 1:

quicksort (A [l ... r])

// sorts a subarray by quicksort

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

// output: subarray A [l ... r) sorted in ascending order.

if l < r

    s ← partition (A [l ... r]   // s is a split partition

    quicksort (A [l ... s-1])

    quick sort (A [s+1 ... r])

Partitioning:

* we start by selecting a pivot - an element with respect to those value we are going to divide the subarray.

* There are several different strategies for selecting a pivot where in here we use the simplest strategy of selecting the subarray's first element: p = A [l].

* Make first element as the pivot element P.

* set i (low index to point to 0) and j to high which is n-1.

* Increment i until A[i] ≥ p. If condition is met stop incrementing i.

* Decrement j until A[j] ≤ p. If condition is met stop decrementing j.

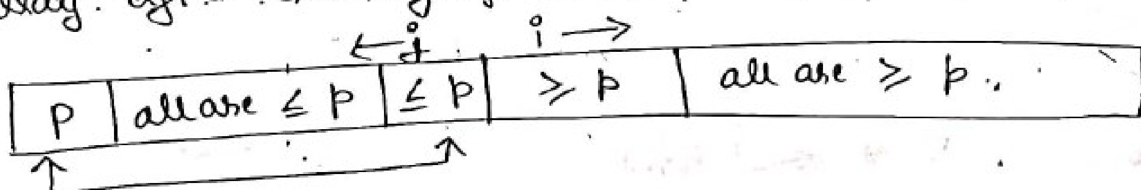
* compare the positions of i and j (three case may arise <, >, =)

i) If i<j performs swap A[i], A[j] and continue incrementing i and from the same position.

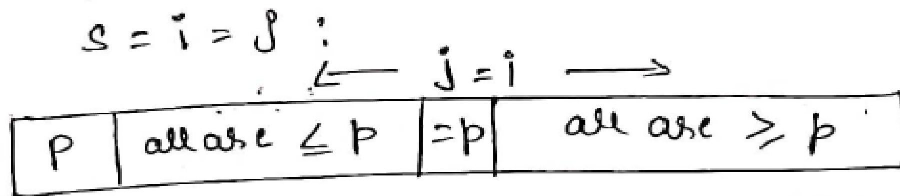
* ~~If scanning index has crossed~~

* If scanning index i and j have not crossed, i.e i<j, we simply exchange A[i] and A[j] and resume scanning by incrementing i and decrementing j, respectively.

* If i > j then swap pivot element with a[j]; return j, as the split position. If the scanning indices have crossed over i.e, i > j, we will have partitioned the subarray after exchanging the pivot with A[j]:



* Finally, if the scanning indices stop while pointing to the same element. i.e, $i = j$, the value they are pointing to must be equal to p. Thus, we have the subarray partitioned; with the split position $s = i = j$:



* We combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$. j is the split position.

Algorithm:

```
Hoare Partition (A[l...r])
// partitions a subarray by Hoare's Algorithm, using the first
   element as a pivot
// Input: subarray of array A[0..n-1], defined by its left &
          right indices l and r (l < r)
// output: Partition of A[l.. r], with the split position
           returned as this function's value.
p ← A[l].
i ← l ; j ← r+1
repeat
      repeat i ← i+1 until A[i] ≥ p
      repeat j ← j-1 until A[i] ≤ p
      swap (A[i], A[j])
until i ≥ j
swap (A[i], A[j])        // undo last swap when i ≥ j
swap (A[l], A[j])
return j.
```

Analysis according to master theorem:

Best-case:

The best case of quick sort occurs, when the problem's instance is divided into 2 equal parts on each recursive call of the algorithm. So the recurrence relation will be.

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

// it is written as $2T(n/2)+$

$\therefore a = 2, b = 2, f(n) = n = n^1 = n^d$, so $d = 1$.

So, according to master theorem $T(n)$ is given by

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

For the quick sort algorithm, $a = b^d$ holds good.

$\therefore T(n) = \Theta(n^d \log n) = \Theta(n^1 \log_2 n) \in \Theta(n \log n)$.

$$\boxed{T(n) \in \Theta(n \log n)}$$

Worst case:

The worst case of the quick sort occurs, when at each invocation of the quick sort algorithm, the current array is partitioned into 2 subarrays with one of them being empty.

This situation occurs when the input list is arranged in either ascending or descending order.

Ex: 10 11 12 13.

10 11 12 13

10 11 12 13

10 11 12 13

The recurrence relation for the above situation is:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(0) + T(n-1) + Cn & \text{otherwise.} \end{cases}$$

T(0) → To sort left subarray; T(n-1) → To sort right subarray.

$\therefore T(n) = T(0) + T(n-1) + Cn$

$T(n) = T(n-1) + Cn$ // $T(0) = 0$

By the method of backward subsitution,

$T(n) = T(n-1) + Cn$

$= T(n-2) + C(n-1) + Cn$

$= T(n-3) + C(n-2) + C(n-1) + Cn$

$= T(n-4) + C(n-3) + C(n-2) + C(n-1) + Cn$

⋮

$= T(n-i) + C(n-i-1) + (n-i-2) \ldots + (n-3) + (n-2) + (n-1)$

$= T(n-n) + C[(n-(n-1)) + (n(n-2)) + \cdots + (n-3) + (n-2) + (n-1) + n$

$= T(0) + C[1 + 2 + 3 + 4 \ldots + (n-3) + (n-2) + (n-1) + n]$

$= C[1 + 2 + 3 + 4 + \ldots (n-3) + (n-2) + (n-1) + n]$ // $T(0) = 1$

$T(n) = n(n+1)/2 = n^2$

$$\boxed{T(n) \in \Theta(n^2)}$$

Average Case Analysis:

The average case of the quick sort will appear for typical or random or real time input.

* where the given array may not be exactly partitioned into 2 equal parts as in the best case or the sub-arrays are not skewed as in the worst case.

Here, the pivot element is placed at an arbitrary position from 1 to n.

Let k be the position of the pivot element, as shown,

1, 2, 3 . . . . . k − 1	k	k+1, k+2, . . . n

↑ ↑

Then,

$$T(n) = T(k-1) + T(n-k) + (n+1)$$

$$T(n) = (n+1)\, 2 \log (n+1) = n \log n$$

$$\boxed{T(n) = \in \Theta(n \log n)}$$

Array: 5, 3, 1, 9, 8, 2, 4, 7

Pivot = 5
i starts from left + 1
j starts from right end

```
Index:   0   1   2   3   4   5   6   7
Array:   5   3   1   9   8   2   4   7
         P
             i                       j
```

Step 1:
i moves right until element > pivot
3 < 5 → move

1 < 5 → move
9 > 5 → stop

```
Index:  0  1  2  3  4  5  6  7
Array:  5  3  1  9  8  2  4  7
      P        i      j
```

Step 2:
j moves left until element < pivot
7 > 5 → move
4 < 5 → stop

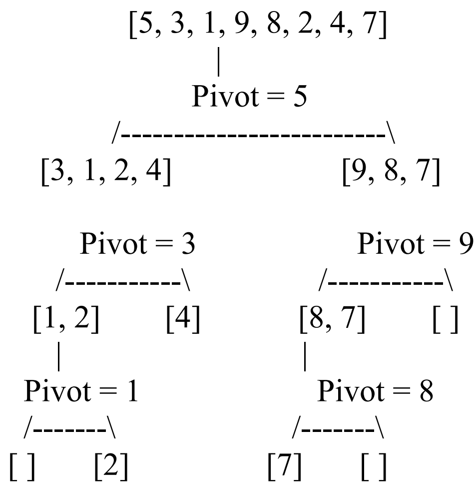
```
Index:  0  1  2  3  4  5  6  7
Array:  5  3  1  9  8  2  4  7
      P        i    j
```

Now swap elements at i and j

Swap 9 and 4

Array after swap:
5, 3, 1, 4, 8, 2, 9, 7

Initial Array
5, 3, 1, 9, 8, 2, 4, 7



Final Combination:

[1, 2] + [3] + [4] + [5] + [7, 8] + [9]

Sorted Array:
1, 2, 3, 4, 5, 7, 8, 9

4. Define Topological Sorting. Explain the different approaches used for topological sorting such as Source Removal Method and DFS-based Method with examples. (10M CO2 L2)

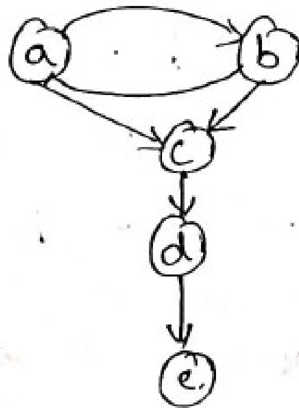
Topological sorting:

* Before we move to the topological sorting, we must know about the graphs.

* A Directed graph or digraph, is a graph with directed ~~graphs themselves~~. specified for all its edges.

* A directed cycle in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.

Ex: 1)



Scheme:
Topological sort – 2M
DFS Method – 4M
Source removal method – 4M

a; b, a is a directed cycle in the above graph.

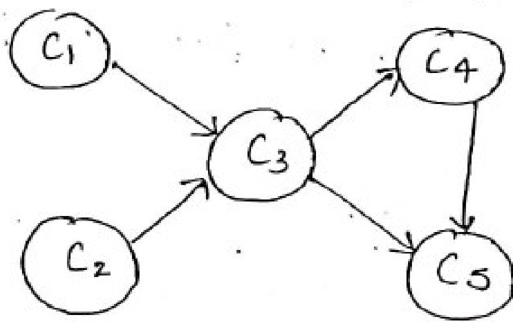
* conversely, if a DFS forest of a digraph has no back edges, the diagraph is a dag, an acronym for directed acyclic graph.

Ex: 2)

Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.

* The courses can be taken in any order as long as the following courses prerequisites are met:

C1 and C2 has no prerequisites, C3 requires C1 and C2, C4 requires C3 and C5 requires C3 and C4.



* The student can take only one course per term.
* The question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends.

This problem is called topological sorting.

* For topological sorting to be possible, a diagraph to must be a dag.
* There are two efficient algorithms that both verify whether a diagraph is a dag and, if it is, produce an ordering of vertices that sloves the topological sorting problem.

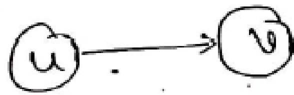
First Algorithm: Depth-First Search (DFS):

* DFS is an algorithm used for traversing or searching tree or graph data.

* It starts at a designated node and explores as far as possible along each branch before backtracking.
* Along the path, or order, it in which vertices become dead-ends, they are popped out of the traversal stack.
* Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal.

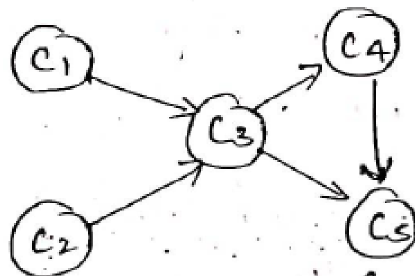
Ex:

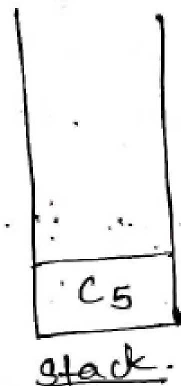
* When a vertex v is popped off a DFS stack, no vertex u with an edge from u to v can be among the vertices ~~popped off before v.~~



* Hence, any such vertex u will be listed after v in the popped-off order list, and before v in the reversed list.

In Ex2:





Let us consider the edge as $C_1 \rightarrow u$

So, the traversal starts from C_1

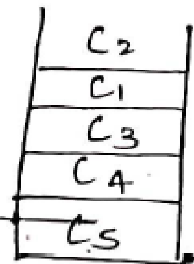
i) $C_1 \longrightarrow C_3 \longrightarrow C_4 \longrightarrow C_5$

ii) since C_5 doesnot have any vertex further connected it is moved into the stack and ~~Starts~~ DFS again starts with C_4.

iii) Again C_4 does not have any path except C_5, C_4 is also popped into the stack.

So on, for C_3 and C_1 also.

iv) Now, we start DFS from C_2, as C_2 does have traversal from $C_2 \rightarrow C_3$, But C_3 is also already in the stack. We stop at C_2.



* When we, reverse the order of the stack, we get the topological sorting of the given graph.

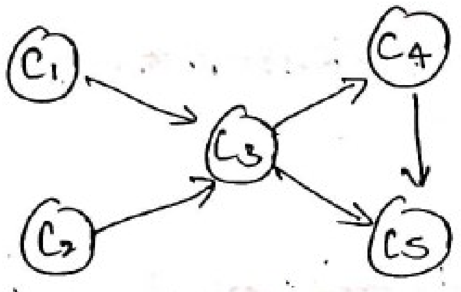
i.e C_2, C_1, C_3, C_4, C_5.

Second Algorithm: Decrease - (by - one) - and - conquer

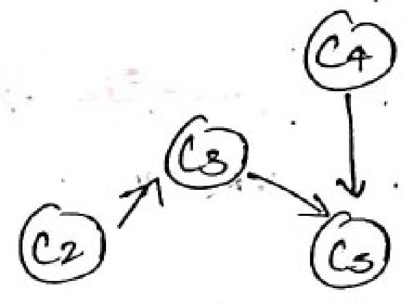
* Repeatedly, Identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.

* The order in which the vertices are deleted yields a solution to the topological sorting problem.

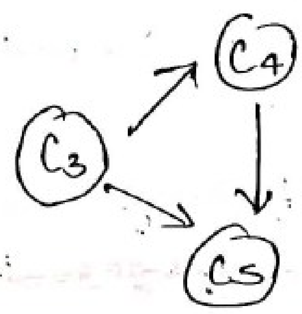
Ex:



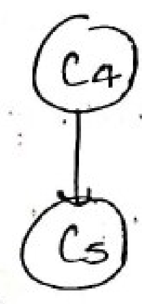
delete C_1 →



delete C_2 →



delete C_3 →



delete C_4 →

delete C_5 →

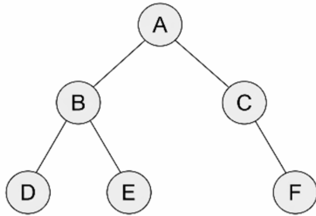
The solution obtained is C_1, C_2, C_3, C_4, C_5.

* Note:

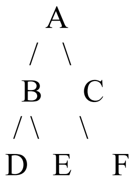
The solution obtained by the source-removal algorithm is different from one obtained by the DFS-algorithm.

* Both are correct, of the topological sorting problem.

5.a. Write algorithms for Preorder, Inorder and Postorder traversals of a tree. Obtain the traversals for a given tree. (6M CO2 L3)



Scheme:
Algorithm – 3M
Solution – 3M



Preorder Traversal (Root → Left → Right)

Algorithm:

```
void preorder(struct node *root)
{
   if(root != NULL)
   {
      printf("%c ", root->data);
      preorder(root->left);
      preorder(root->right);
   }
}
```

Traversal:

A, B, D, E, C, F

Inorder Traversal (Left → Root → Right)

Algorithm:

```
void inorder(struct node *root)
{
   if(root != NULL)
   {
      inorder(root->left);
      printf("%c ", root->data);
      inorder(root->right);
   }
}
```

Traversal:

D, B, E, A, C, F

Postorder Traversal (Left → Right → Root)

Algorithm:

```
void postorder(struct node *root)
{
   if(root != NULL)
   {
      postorder(root->left);
      postorder(root->right);
      printf("%c ", root->data);
   }
}
```

Traversal:

D, E, B, F, C, A

Scheme:
Diagram -2M
Explanation – 2M

5.b. Explain the concept of Divide and Conquer. (4M CO2 L2)

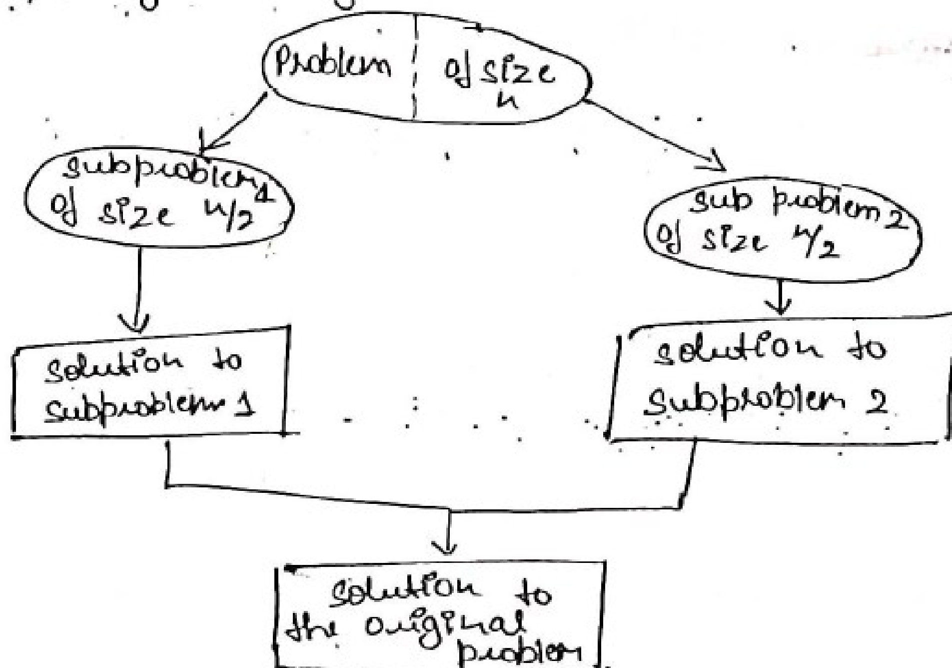
Divide - and - Conquer:

* Divide -and- conquer is probably the best-known general algorithm design technique.

* The algorithm works according to the following general plan:

1) A problem is divided into several subproblems of the same type, ideally of about equal size.
2) The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough)
3) If necessary, the solutions to the subproblems are combined to get a solution to the original problems.

* The divide-and-conquer technique depicts the case of dividing a problem into 2 smaller subproblems, by far the most widely occuring case.

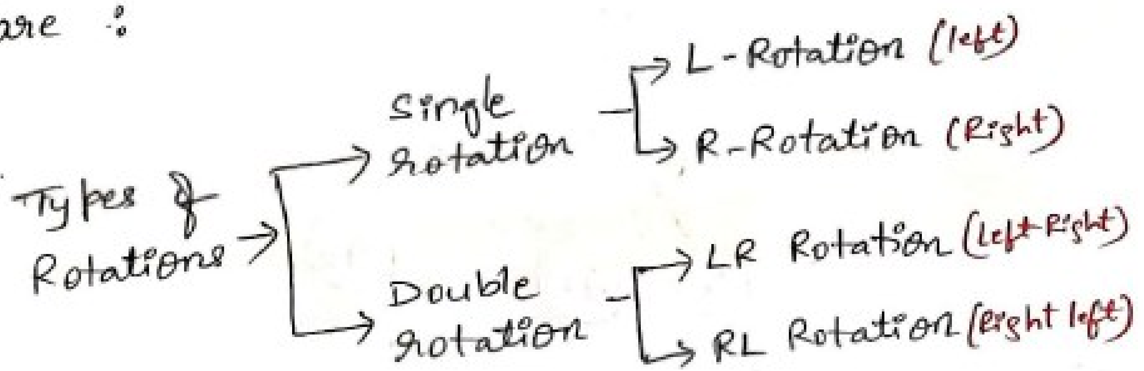


6. Define AVL Trees. Explain its four rotation types. **(10M CO3 L2)**

Scheme:
AVL Tree explanation – 2M
Each rotation (2M) – 8M

Creating an AVL Tree :-

Once an item is inserted into the tree, the tree may be unbalanced. In such case, it is necessary to balance the tree. Balancing of AVL tree is done using rotations. The various type of rotations are :



After inserting a node, trace backward towards the root and compute the balance factor along the path. If we find a node whose balance factor is other than -1, or 0 or +1, then rotation is required.

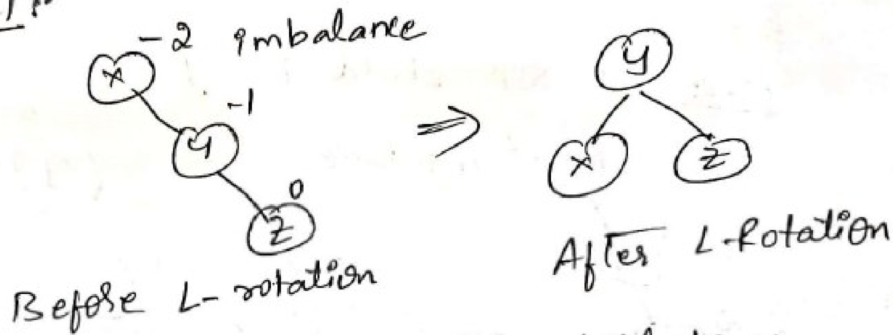
The procedure to identify L-Rotation and R-Rotation is given below :

<u>Step 1</u> : Identify the first node where balance factor is not 0 or -1 or 1 in the path while moving from node inserted towards root.

<u>Step 2</u> :- Identify two other nodes below the node where imbalance occurs in the path.

1) <u>L-Rotation</u> :- If the three nodes identified in step1 and step2 are in a straight line, single rotation is required. If the balance factor of a node, where imbalance occurs, is -2, then the tree is <u>heavy towards right.</u> so to balance it we rotate left. This is called left rotation.

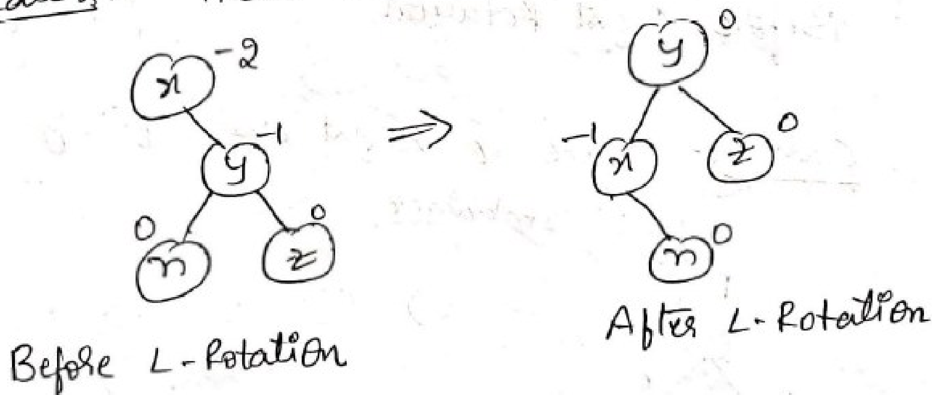
<u>Case 1</u> :-



Before L-rotation

After L-Rotation

<u>Case 2</u> :- There is a left child for y



Before L-Rotation

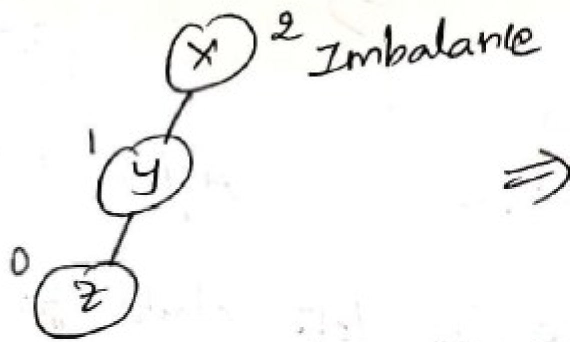
After L-Rotation

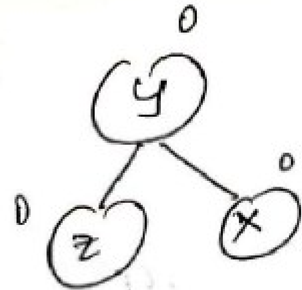
2) R-Rotation :-

If the three nodes identified in Step1 and Step2 are in a straight line, single rotation is required. If the balance factor of a node, where imbalance occurs is 2, then tree is heavy towards left. So to balance it we rotate right. This is called right rotation.

Eg:-

Case 1 : No child for y

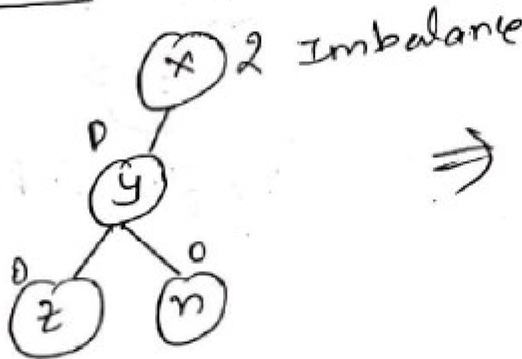


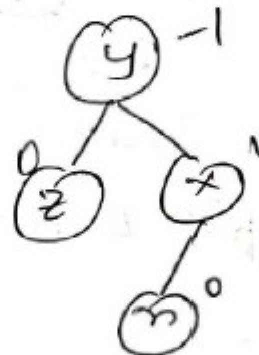


Before Right Rotation

After Right Rotation

Case 2 :- There is right child for y





Before Right Rotation

After Right Rotation

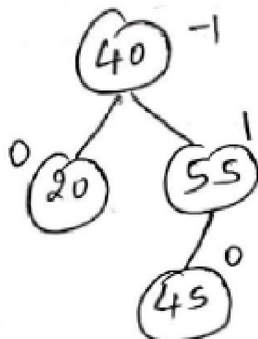
(iii) L-R Rotation :-

If the three nodes identified are not in a straight line, then double rotation is required. A double rotation is a combination of two single rotations as shown below:

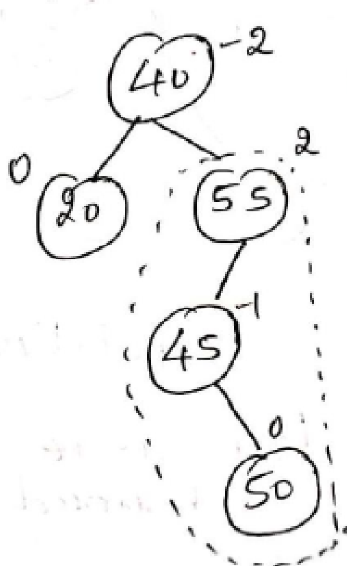
i) Assume 'x' is a node where imbalance occurs and y is its child in the path. If the balance factor of y is -1, this subtree is right heavy, so rotate left at y. Attach the parent of resulting subtree to x.

ii) Second rotation is required at x. Here right rotation is required.

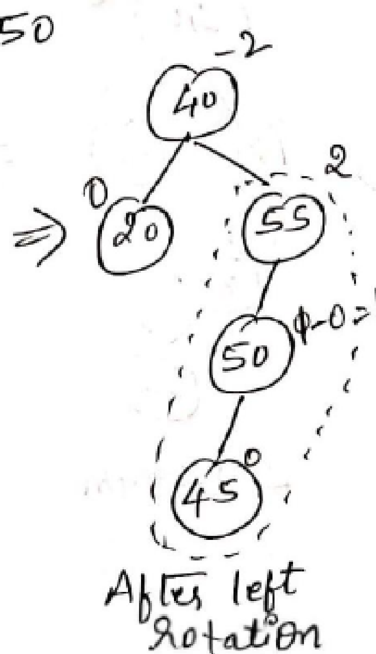
Eg: Consider the following tree



Now insert 50

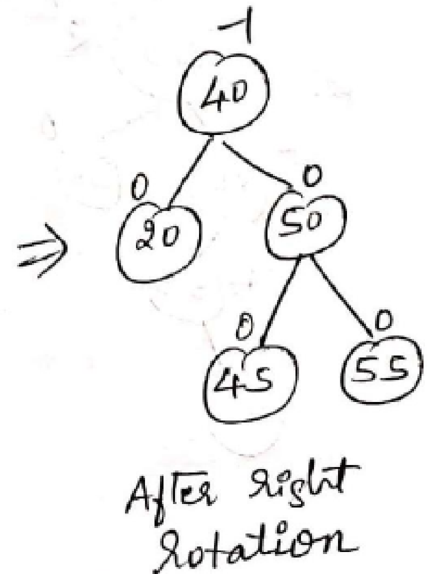


⇒



After left rotation

⇒



After right rotation

(iv) R-L Rotation :-

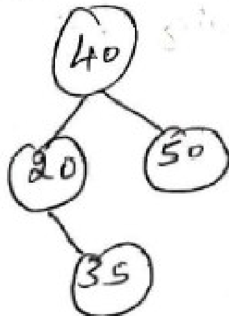
If the three nodes identified are not in a straight line, then double rotation is required.

A double rotation is the combination of two single rotations as shown below:

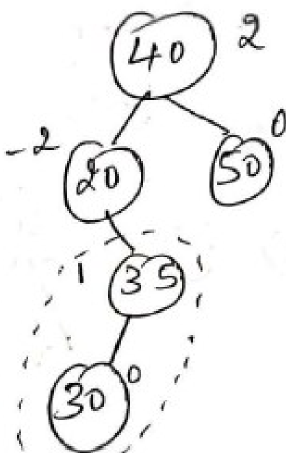
(i) Assume 'x' is a node where imbalance occurs and 'y' is its child in the path. If the balance factor of y is 1, this subtree is left heavy, so rotate right at y. Attach the parent of resulting subtree to x.

(ii) Second rotation is required at x. Here left rotation is required.

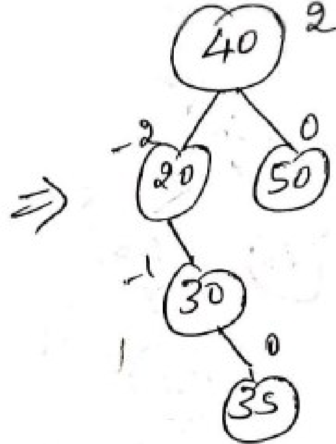
Eg:- Consider the following tree:



Now insert element 30

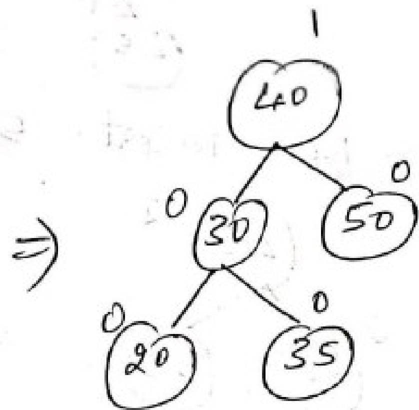


⇒



R. Rotation

⇒



L. Rotation

{The tree is balanced}