**19I510 Design and Analysis of Algorithms**

**Exercise 1 a) – Analysis of Algorithms**

1. Lowest Common Ancestor in a Binary Search Tree

Given values of two values n1 and n2 in a Binary Search Tree, find the Lowest Common Ancestor (LCA). You may assume that both the values exist in the tree.



**Algorithm:**

1. Create a recursive function that takes a node and the two values n1 and n2.
2. If the value of the current node is less than both n1 and n2, then LCA lies in the right subtree. Call the recursive function for the right subtree.
3. If the value of the current node is greater than both n1 and n2, then LCA lies in the left subtree. Call the recursive function for the left subtree.
4. If both the above cases are false then return the current node as LCA.

**Input Format**The first contains two space separated integers P and Q, the nodes whose LCA to find.

**Output Format**print the LCA of nodes P and Q

**Constraints:**

1 <= N <= 100

0 <= Node Data <= 10^3 and Node Data != -1

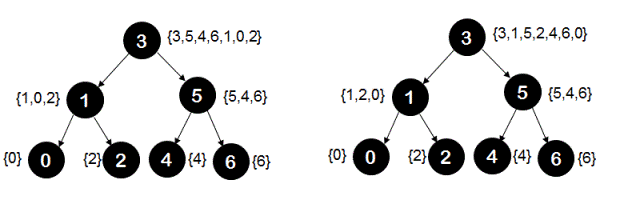
**Sample Input:**

10 14  
**Sample Output:** 12

**Sample Input:**

10 22  
**Sample Output:** 20

1. Identical BST

Given two arrays which represent a sequence of keys. Imagine we make a Binary Search Tree (BST) from each array. We need to tell whether two BSTs will be identical or not without actually constructing the tree.

**Algorithm:**

* According to BST property, elements of the left subtree must be smaller and elements of right subtree must be greater than root.
* Two arrays represent the same BST if, for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees.
* The idea is to check of if next smaller and greater elements are same in both arrays. Same properties are recursively checked for left and right subtrees. The idea looks simple, but implementation requires checking all conditions for all elements.

**Input Format**

The first line contains elements of the first binary search tree. The line consists of values of nodes separated by a single space.

The second line contains elements of the second binary search tree. The line consists of values of nodes separated by a single space.

**Output Format**

print in a single line either “True” (if the two trees are identical) or “False” otherwise.

**Constraints :**

0 <= N <= 10^5

0 <= M <= 10^5

1 <= Node Data <= 10^9

**Sample Input**

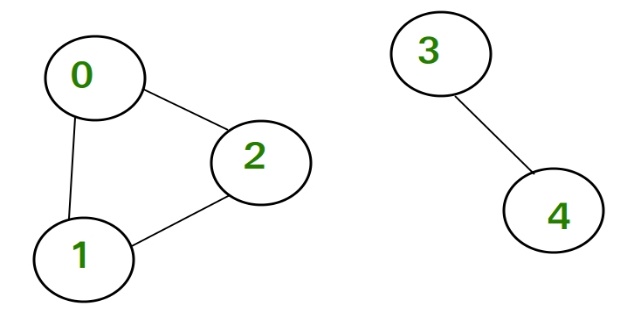
3 1 5 0 2 4 6

3 1 5 0 2 4 6

**Sample Output**  
True

1. Count the number of reachable nodes.

Given an undirected graph and a set of vertices, count the number of reachable nodes from the given head node using a depth-first search.

Consider below undirected graph with two disconnected components:

**Algorithm:**

* Take the following global variables:

2D array ‘Graph’, to store graphs.

‘Visited’ array to mark each node whether it is visited or not.

 Let ‘countNodes(n, m, edges)’ be the function that counts the number of nodes reachable from each node. It returns the array of size n.

* Clear graph, initialize the visited array to false.
* Run a loop from 0 to 'm' :
  + Add the undirected edge between edges[i] [0] and edges[i][1].
* Take an array 'ans' to store the answer.
* Run a loop from 0 to 'n'.
  + Perform a dfs call and store the return value in variable ‘count’ i.e count = dfs(i).
  + Store ‘count’ at ‘ans[i]’.
  + Mark visited node 'i' to false for the next dfs call.
* Return ‘ans’.

**Description of dfs(n) function**

* Mark visited[s] equal to true.
* Take a variable ‘count’ and initialize it to 0.
* Travel through its adjacents and store the current adjacent in variable ‘adj’.
  + If visited[adj] is false.
    - Update the count of nodes in this component i.e count += dfs(adj).
* Return ‘count’ + 1.

**Input Format**The first line of contains two integers ‘N’ and ‘M’, denoting the number of nodes and the number of edges.

The next ‘M’ lines contain two space-separated integers ’u’ and ‘v’, denoting the edge between ‘u’ and ‘v’.

Node ‘i’ to start with where 0 <= ‘i’ <= ‘N’ - 1.

**Output Format**Outputs the number of nodes reachable from node ‘i’.

**Constraints:**

1 <= N <100, M <= N\*(N-1)

0 <= u, v <= N-1

**Sample Input**

Input :

5 4

0 1

0 2

1 2

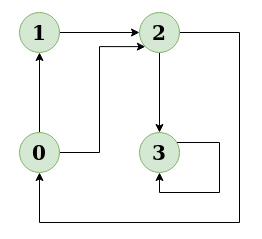
3 4

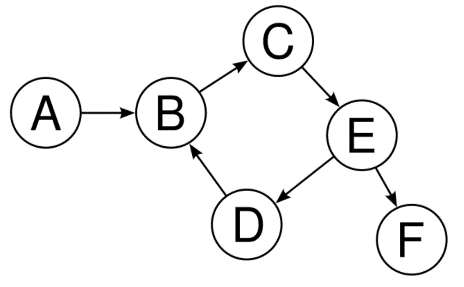
0

**Sample Output**  
3

1. Detect cycle in a directed graph

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false.





**Algorithm:**

1. Create the graph using the given number of edges and vertices.
2. Create a recursive function that initializes the current index or vertex, visited, and recursion stack.
3. Mark the current node as visited and also mark the index in recursion stack.
4. Find all the vertices which are not visited and are adjacent to the current node. Recursively call the function for those vertices, If the recursive function returns true, return true.
5. If the adjacent vertices are already marked in the recursion stack then return true.
6. Create a wrapper class, that calls the recursive function for all the vertices and if any function returns true return true. Else if for all vertices the function returns false return false.

**Input Format**The first line is an integer ‘N’ representing the number of nodes in the graph.

The second line contains a given integer ‘M’ representing the number of edges.

The next ‘M’ lines contain two space-separated integers ’u’ and ‘v’, denoting the edge between ‘u’ and ‘v’.`  
**Output Format**print true if a cycle is present in the given directed graph else print false.

**Constraints :**

1 <= N <= 100

1 <= M <= min(100,N(N-1))

0 <= u, v <= N-1

**Sample Input:**

**N = 4, M = 6**

**0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3**

**Sample Output:**

**True**