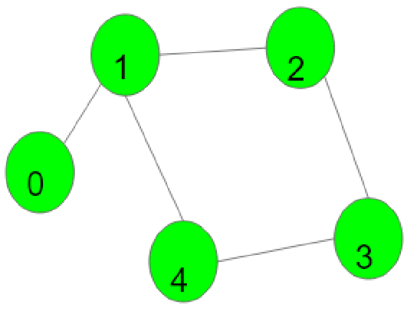
Graphs
Lesson 5

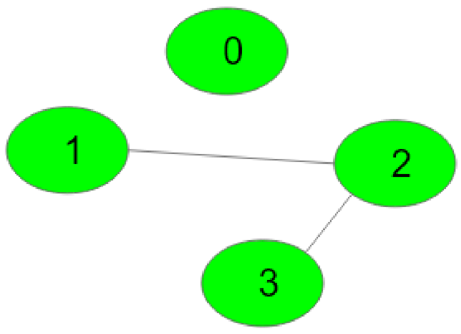
Detect cycle in an undirected graph

Given an undirected graph, how to check if there is a cycle in the graph?
Example 1:
Input:



Output: 1
Example 2:

Input:



Output: 0
Explanation: No cycle in the graph.

Expected Time Complexity: O(V + E)

Expected Space Complexity: O(V)

Constraints:

1 ≤ V, E ≤ 105

Approach: Run a DFS from every unvisited node. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that joins a node to itself (self-loop) or one of its ancestors in the tree produced by DFS. To find the back edge to any of its ancestors, keep a visited array and if there is a back edge to any visited node then there is a loop and return true.

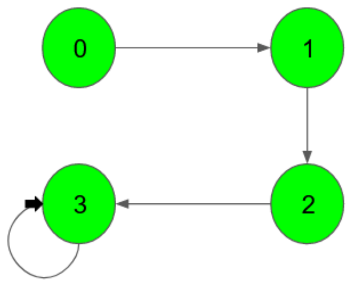
```
bool cycle(vector<int>adj[],int x,vector<bool>&visited,int parent)
   {
       visited[x]=true;
       for(int i=0;i<adj[x].size();i++)
       {
          if(!visited[adj[x][i]] )
          {
             if(cycle(adj,adj[x][i],visited,x))
                return true;
          }
          else if(adj[x][i]!=parent || adj[x][i]==x)
          {
             return true;
          }
       }
       return false;
   }
        bool isCycle(int V, vector<int>adj[])
        {
           vector<bool>visited(V,false);
           for(int i=0;i<V;i++)
           {
              if(!visited[i] && cycle(adj,i,visited,-1))
               return true;
           }
           return false;
        }
```

Detecting Cycle in a Directed Graph

Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, check whether it contains any cycle or not.

Example 1:

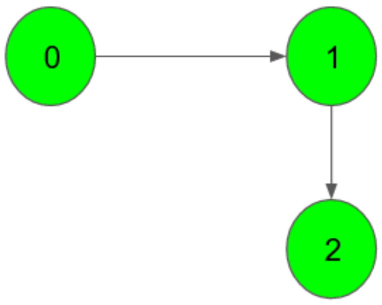
Input:



Output: 1

Example 2:

Input:



Output: 0

Expected Time Complexity: O(V + E)

Expected Auxiliary Space: O(V)

Constraints:
1 ≤ V, E ≤ 105

Why can't we apply the approach used in the above question??

```
bool cycle(vector<bool>&visited,vector<bool>&curr, int x , vector<int>adj[])
   {
       if(!visited[x])
       {
           curr[x]=true;
           visited[x]=true;
           for(int i=0;i<adj[x].size();i++)
           {
               if(!visited[adj[x][i]])
               {
                   if(cycle(visited,curr,  adj[x][i] , adj))
                       return true;
               }
               else if(curr[adj[x][i]]==true)
                   return true;
           }
       }
       curr[x]=false;
       return false;
   }
        bool isCyclic(int V, vector<int> adj[])
        {
            vector<bool>visited(V,false);
            vector<bool>curr(V,false);
            for(int i=0;i<V;i++)
            {
                if(!visited[i] && cycle(visited,curr,i,adj))
                    return true;
            }
            return false;       t}
```

Disjoint Set (Or Union-Find) (Detecting Cycle in an Undirected Graph)

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

The Union-Find Algorithm can be used to check whether an undirected graph contains a cycle or not. Note that we have discussed an algorithm to detect a cycle. This is another method based on Union-Find. This method assumes that the graph doesn’t contain any self-loops.

```cpp
vector<int> dsuf;
//FIND operation
int find(int v)
{
        if(dsuf[v]==-1)
                return v;
        return find(dsuf[v]);
}

void union_op(int fromP,int toP)
{
        fromP = find(fromP);
        toP = find(toP);
        dsuf[fromP] = toP;
}

bool isCyclic(vector<pair<int,int>>& edge_List)
{
        for(auto p: edge_List)
        {
                int fromP = find(p.first);  //FIND absolute parent of subset
                int toP = find(p.second);

                if(fromP == toP)
                        return true;

                //UNION operation
                union_op(fromP,toP);    //UNION of 2 sets
        }
        return false;
}
```

Union-Find Algorithm
(Union By Rank and Path Compression)

The idea is to always attach a smaller depth tree under the root of the deeper tree. This technique is called union by rank. The term rank is preferred instead of height because if the path compression technique is used, then rank is not always equal to height. Also, size (in place of height) of trees can also be used as rank. Using size as rank also yields worst-case time complexity as O(Logn) .
The second optimization to naive method is Path Compression. The idea is to flatten the tree when find() is called. When find() is called for an element x, the root of the tree is returned. The find() operation traverses up from x to find the root. The idea of path compression is to make the found root the parent of x so that we don't have to traverse all intermediate nodes again. If x is the root of a subtree, then the path (to root) from all nodes under x also compresses.
The two techniques complement each other. The time complexity of each operation becomes even smaller than O(Logn). In fact, amortized time complexity effectively becomes a small constant.

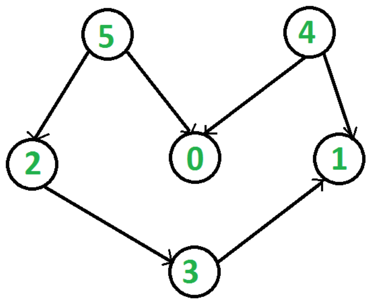
```
struct node {
        int parent;
        int rank;
};

vector<node> dsuf;

int find(int v)
{
        if(dsuf[v].parent==-1)
                return v;
        return dsuf[v].parent=find(dsuf[v].parent);    //Path Compression
}

void union_op(int fromP,int toP)
```

```
{
        if(dsuf[fromP].rank > dsuf[toP].rank)  //fromP has higher rank
                dsuf[toP].parent = fromP;
        else if(dsuf[fromP].rank < dsuf[toP].rank)     //toP has higher rank
                dsuf[fromP].parent = toP;
        else
        {
                //Both have same rank and so anyone can be made as parent
                dsuf[fromP].parent = toP;
                dsuf[toP].rank +=1;                   //Increase rank of parent
        }
}

bool isCyclic(vector<pair<int,int>>& edge_List)
{
        for(auto p: edge_List)
        {
                int fromP = find(p.first);  //FIND absolute parent of subset
                int toP = find(p.second);

                if(fromP == toP)
                        return true;

                //UNION operation
                union_op(fromP,toP);      //UNION of 2 sets
        }
        return false;
}
```

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, the topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with an in-degree as 0 (a vertex with no incoming edges).



In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, the DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sort.

Algorithm to find Topological Sorting

We can modify DFS to find the Topological Sorting of a graph. In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices. In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print the contents of the stack. Note that a vertex is pushed to the stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.

DFS approach/recursive:-https://sapphireengine.com/@/g5hhp2

Complexity Analysis:

Time Complexity: O(V+E).

Auxiliary space: O(V).

The extra space is needed for the stack.

```
void top(int src , vector<int>&vis , vector<int> adj[] , vector<int>&v)
    {
      if(vis[src])
        return;
      vis[src]=1;
      for(int i=0;i<adj[src].size();i++)
      {
        if(!vis[adj[src][i]])
          top(adj[src][i]  , vis  , adj  ,  v);
      }
      v.push_back(src);
    }
    //Function to return list containing vertices in Topological order.
    vector<int> topoSort(int V, vector<int> adj[])
    {
        vector<int>v(V),vis(V,0);
        for(int i=0;i<V;i++)
        {
            if(!vis[i])
            {
                top(i,vis,adj,v);
            }
        }
        reverse(v.begin(),v.end());
        return v;
        // code here
    }
```

Kahn’s algorithm for Topological Sorting

Compute in-degree for each of the vertices present in the DAG and initialize the count of visited nodes as 0.Pick all the vertices with in-degree as 0 and add them into a queue.Remove a vertex from the queue and then.

1. Increment count of visited nodes by 1.
2. Decrease in-degree by 1 for all its neighbouring nodes.
3. If the in-degree of a neighbouring node is reduced to zero, then add it to the queue.

Repeat until the queue is empty.If the count of visited nodes is not equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

Time Complexity: O(V+E).

https://sapphireengine.com/@/ktoaan

```
vector<int> topoSort(int V, vector<int> adj[])
        {
            int indegree[V]={0};
            vector<int>vis;
            for(int i=0;i<V;i++)
            {
                for(int j=0;j<adj[i].size();j++)
                {
```

```
            indegree[adj[i][j]]++;
        }
    }

    queue<int>q;
    for(int i=0;i<V;i++)
    {
        if(indegree[i]==0)
            q.push(i);
    }
    int cnt=0;
    while(!q.empty())
    {
        int x=q.front();
        q.pop();
        vis.push_back(x);

        for(int i=0;i<adj[x].size();i++)
        {
            indegree[adj[x][i]]--;
            if(indegree[adj[x][i]]==0)
                q.push(adj[x][i]);
        }
        cnt++;
    }
    if(cnt!=V)
        {
            vis.clear();
            return vis;
        }
    return vis;
}
```