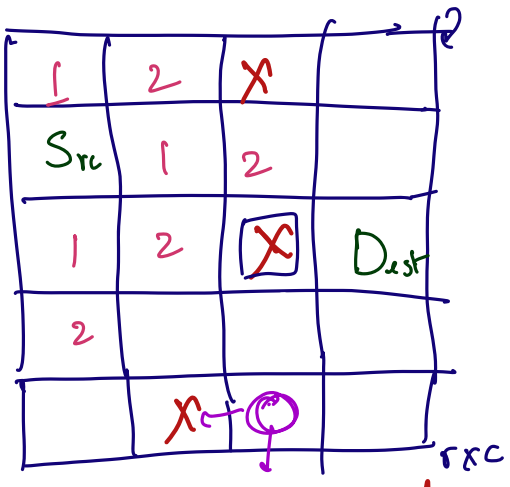
Graphs - 2

Question: shortest distance in a 2D matrix
→ You can move in all 4 directions



Ans = 4

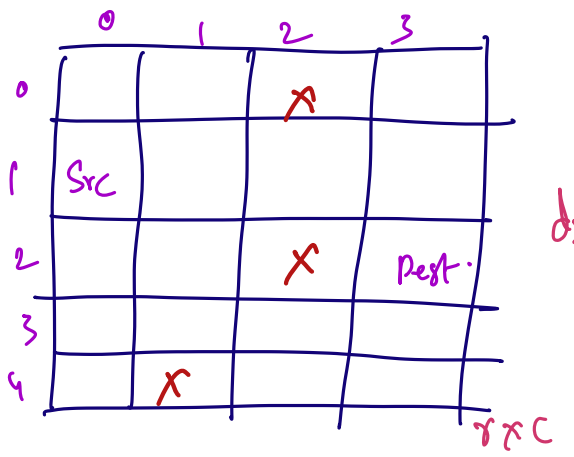
BFS

Map the matrix to graph

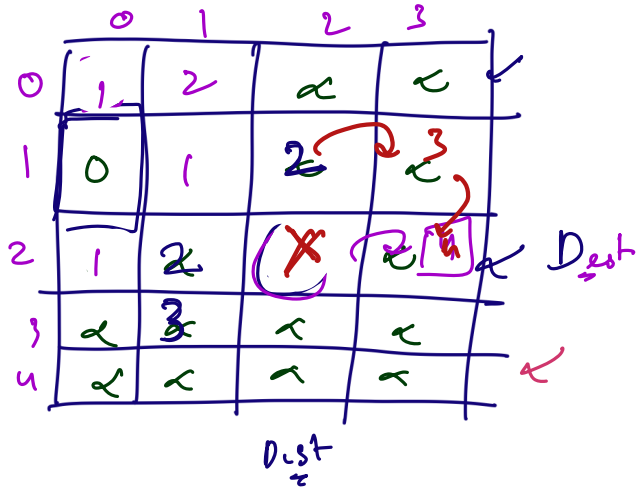
Vertices : All the cells except obstacles are vertex

Edges : 4 edges [Considering out of bounds & obstacles]

visited[]



dist



Queue:

(0,1) (2,1) (1,2)

(0,0)

T.C: $O(V+E)$ → $r \times c$ + $4 \times r \times c$ ⇒ $5 \times r \times c$ ⇒ $O(r \cdot c)$

S.C: $O(r \times c)$

Shortest distance from source to any node

in Unweighted graphs ⇒ BFS

Weighted graphs ⇒ Dijstra's → +ve Negative
Bellman ford → -ve Negative
Floyd Warshal → All pair shortest

Question: Rotten Oranges

Given a matrix of Integers,

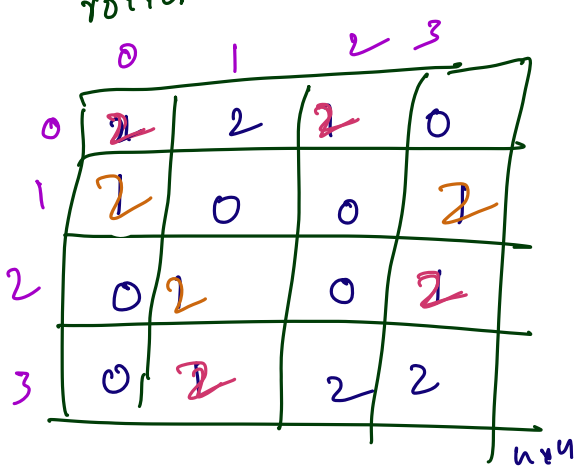
0: Empty Cell

1: Fresh orange

2: Rotten orange

Every minute, any fresh orange adjacent to a rotten orange rotten

Find min time when all oranges become rotten



T = 1

T = 2

Answer

1 2 0 0

1 0 0 1

0 1 0 0

0 1 2 2

Rotten level-order

1	2	1	0
1	0	0	1
0	1	0	1
0	1	2	2

T = 1 → BFS

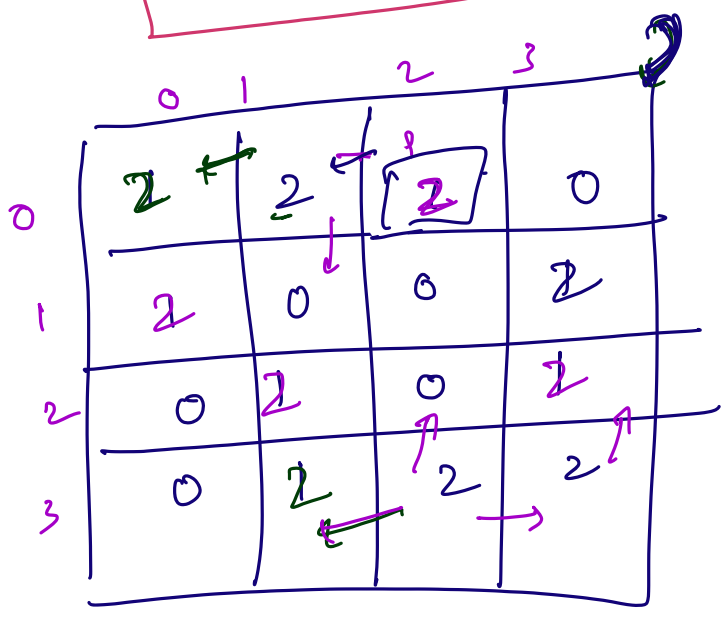
T = 2

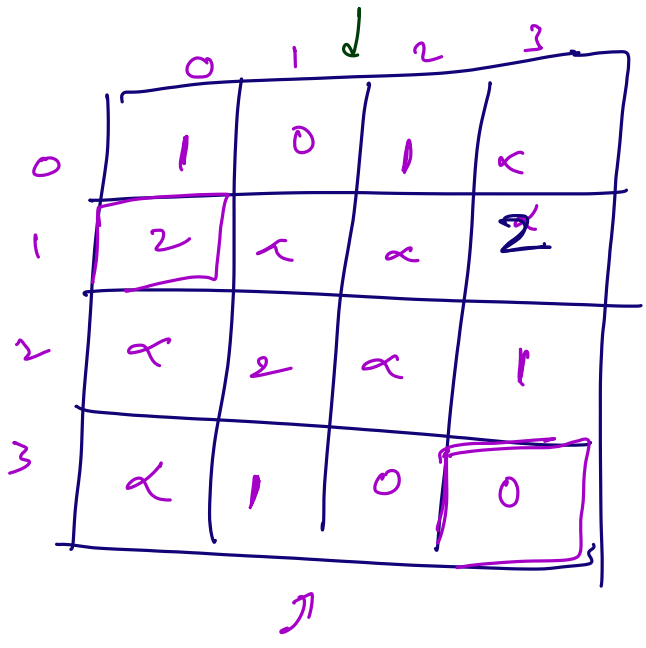
T = 3

From which cell should we start BFS?

min_time = 2

Multi-Source BFS





queue

time - rotten

(1,0)(2,1)

(1,3)

(2,3)

Vertices : Any cell which is fresh/not rotten orange

Edge : 4 Neighbour

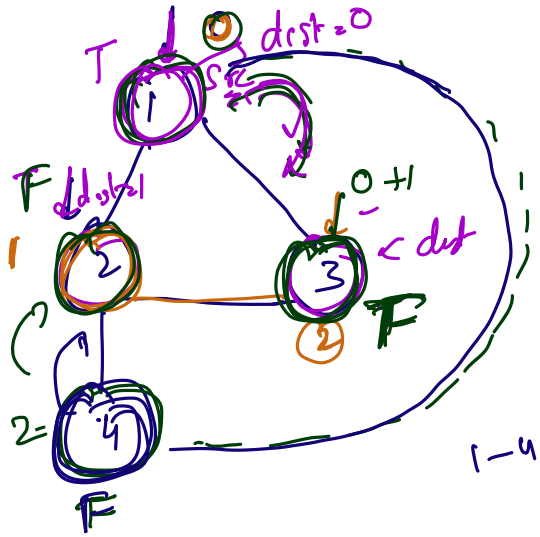
T-C : O(V+E)

↓ ↓

r*c 4*r*c ⇒ O(r*c)

Why not DFS to find Shortest Path?

→ DFS can be used!



DFS(src)

Any 2 ?

DFS(src)

(distance)

Any 2 ? !

1 : { 2, 3, 4 }

2 : { 1, 3, 4 }

3 : { 1, 2 }

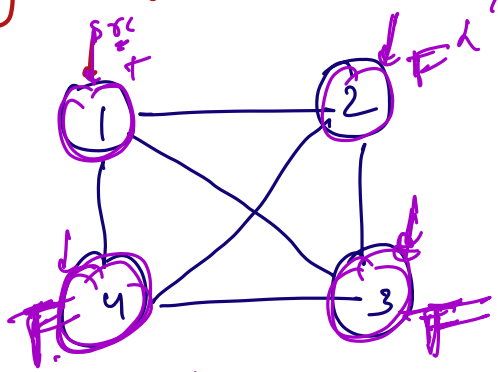
4 : { 2 }

visited[u] = false

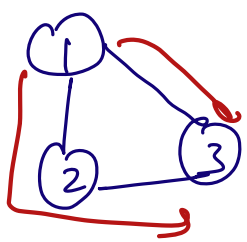
```
DFS (node, dist)
    visited[node] = true;
    for (i : adj[node]) {
        if (! visited[i])
            DFS(i)
    }
    visited[node] = false;
```

Update the Min Dist

* Try Rotten Oranges using DFS



{ 2, 3, 4 }



if(node == dest) Print this

Print all the Paths from src to Dest

DFS

(V-1) (V-2) (V-3)

2) (V-1)! ✓ Efficient

→ 1 2 3 4

→ 1 2 4 3

1 3 2 4

1 3 4 2

1 4 2 3

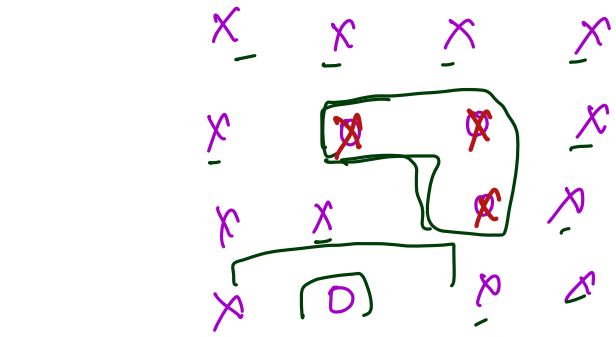
1 4 3 2

Question: Capture regions on board

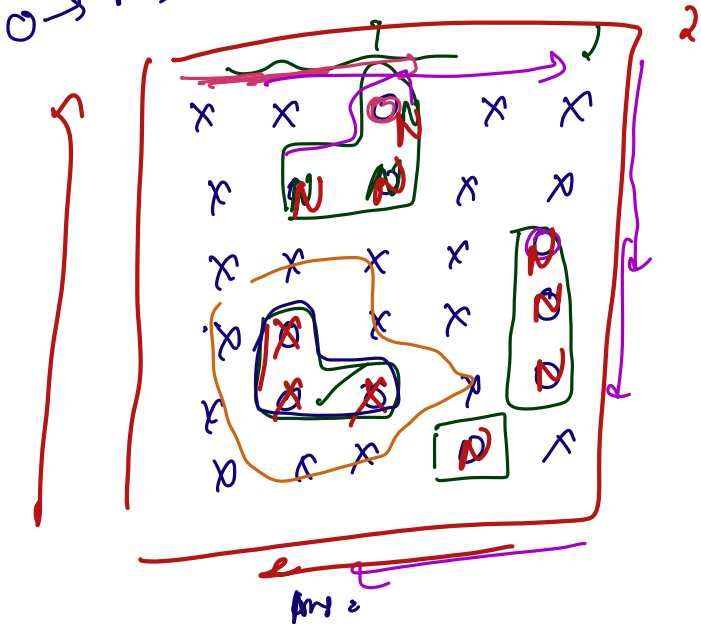
→ Given 2D board A of size N x M containing X's and O's.

→ Capture all regions surrounded by X

Neighbour: ↑ ← → ↓



O → N's



O's → X's

All the set of O's are connected components?

N → O's

Non-0 Islands → DFS

Approach:

For every component, check the boundary

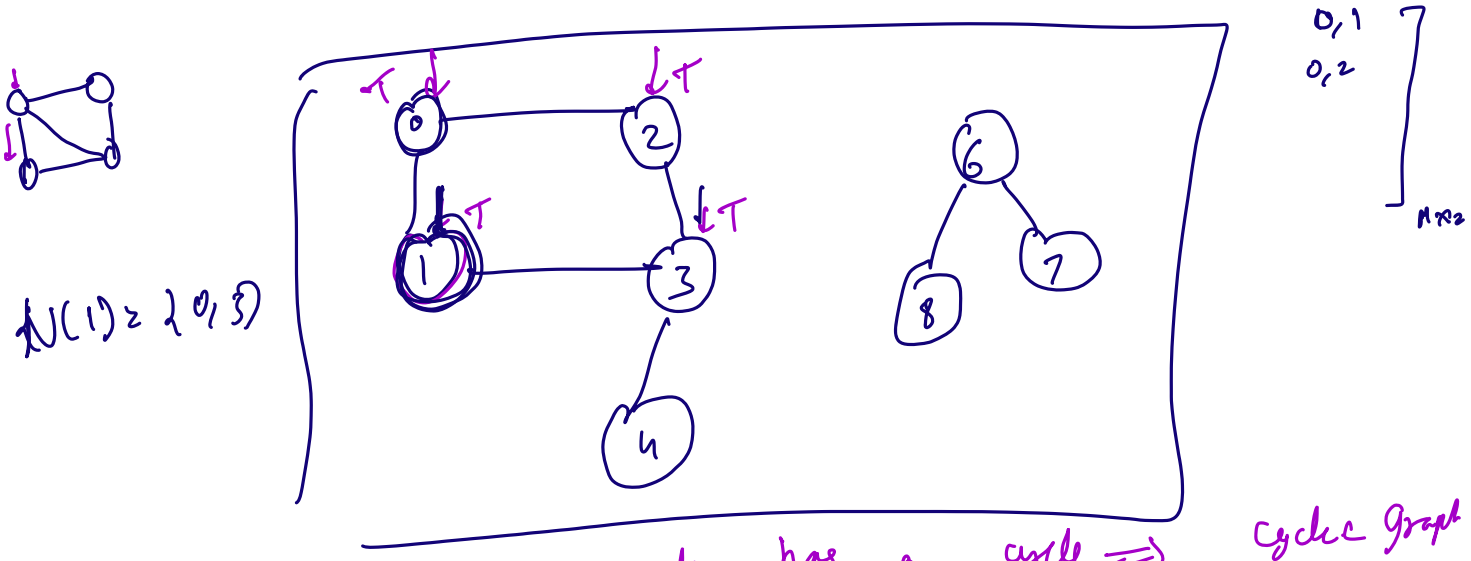
Approach:

T.C: $O(V+E)$ → $O(r \times c)$

$r \times c$ $\quad$ upper

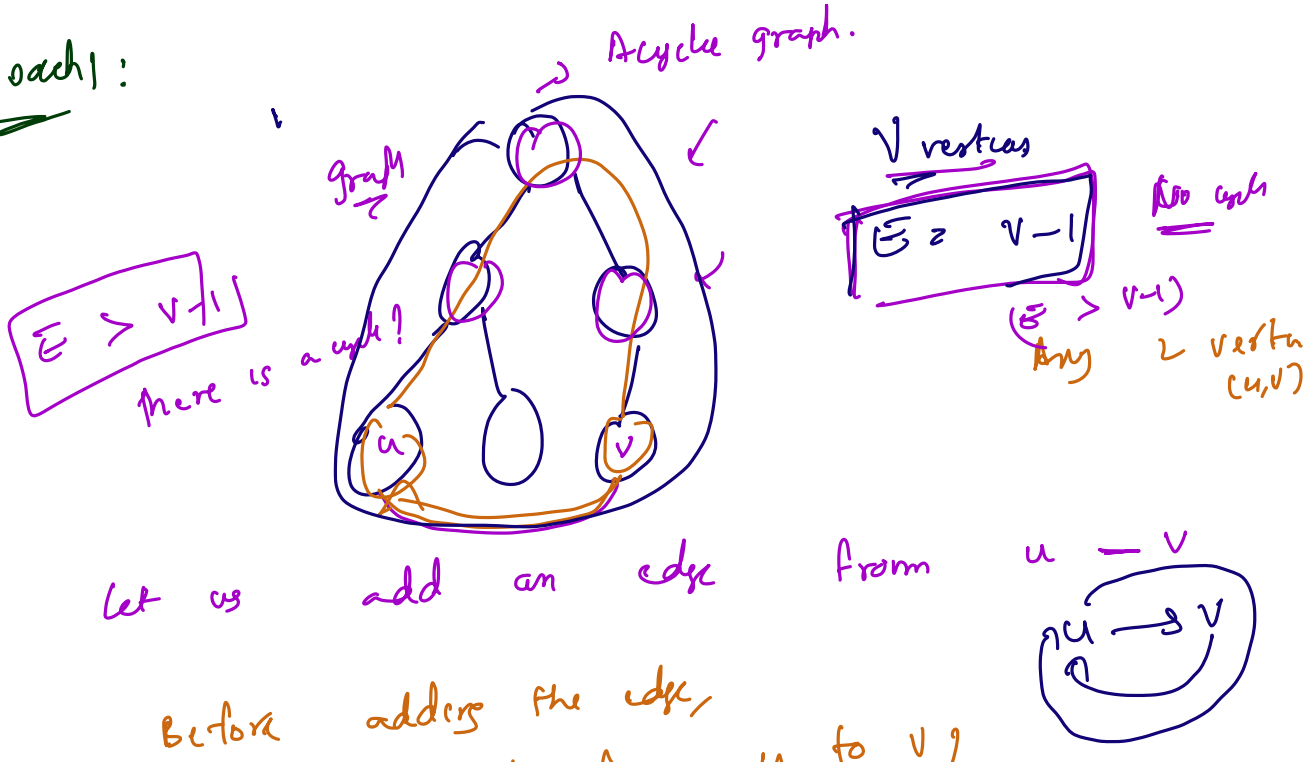
S.C: $O(r \times c)$

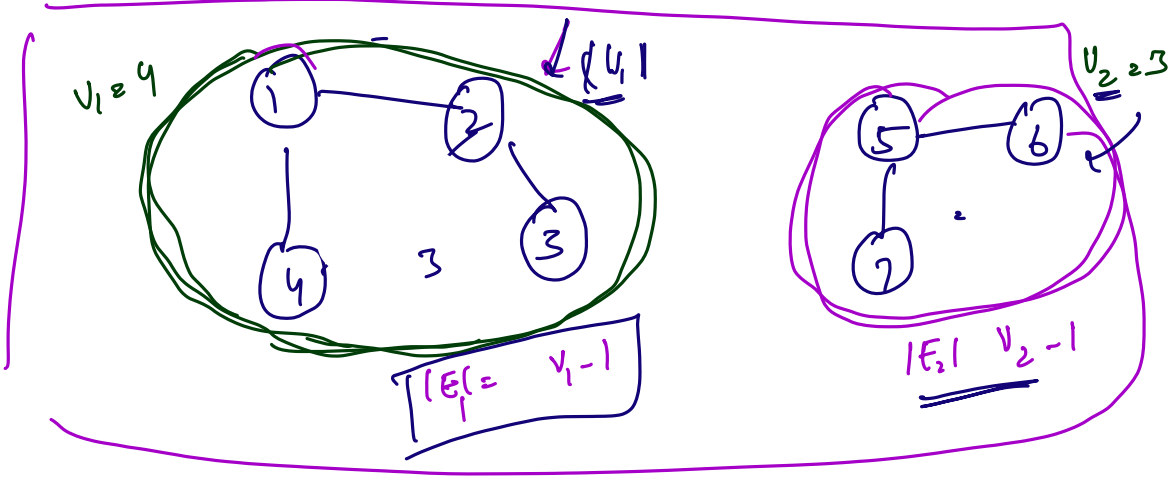
Question: Check if there is a cycle in undirected graph



If Any component has a cycle ⇒ Cyclic Graph

Approach1:





$V_3 - 1$

$V = 7$

$E = 5$ $\quad V = 7$

Comp = 2

$|E_1| + E_2 = (V_1 + V_2) - 2$

$|V| - 2.$

If $(E > |V| - 2)$ Cycle

Total No. Nodes

$V_1 - 1 + V_2 - 1$

$(V_1 + V_2) - 2$ No. of Comp

K Components on a graph

if $(E > V - K)$

Cycle

$V_1 - 1 + V_2 - 1 + V_3 - 1 + \cdots V_k - 1$

$= (V_1 + V_2 + V_3 + \cdots V_k) - K$

$E \geq V - k.$

Cycle

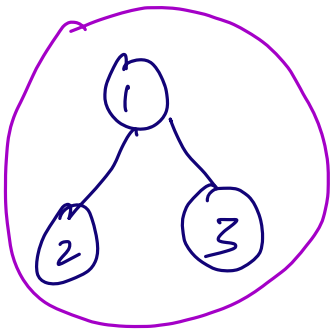
$(E = V - k)$

there n

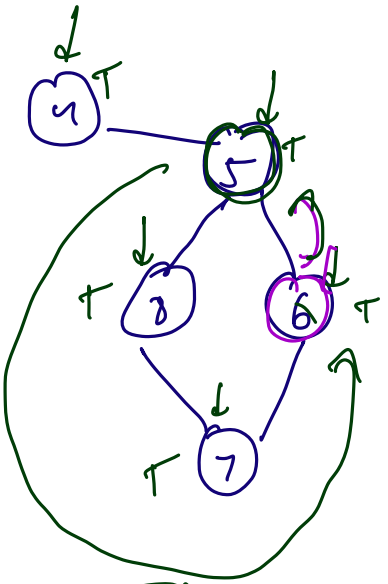
$E > V - 1$

No. of Comp

Approach



x cycle

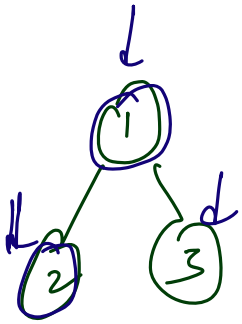


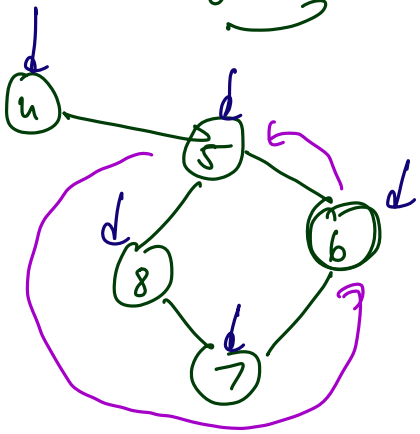
{5, 7}

5 → 3 6

6 → 3 5

cycle



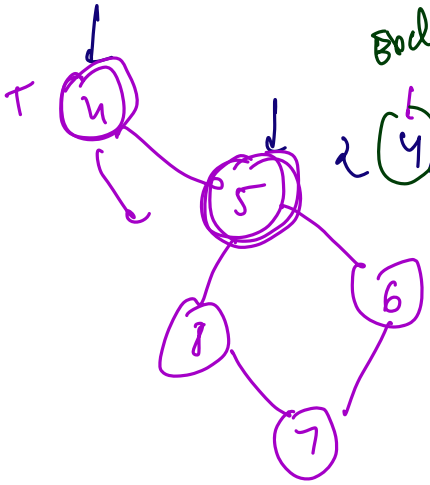


{5, 7}

u → v ∀

v → u

Means a cycle



Exclude my parent

{4, 8, 6}

```
bool isCyclic(int node, int parent){
      visited[node] = true;
      for(auto i : adj[node]){
            if(!visited[i]){
                  if(isCyclic(i, node))
                        return true;
            }
            else if (i != parent)
                  return true;
      }
return false;
}

main(){
      for(int i = 0; i < N; i++){
            if(!visited[i])
                  if(isCyclic(i, -1))
                        return true;
      }
return false;
}
```

node = 1
i = 0, 3

. . , - parent

of neighbour & visited

```
for( i : adj[node] ) {
  if( visited[i] == false)
      DFS(i)
  else if visited[i]==true &&
      i != node)
      return true.
}
```

for

→ Check cr y pproach works for directed graphs ?

BFS

Topological sort

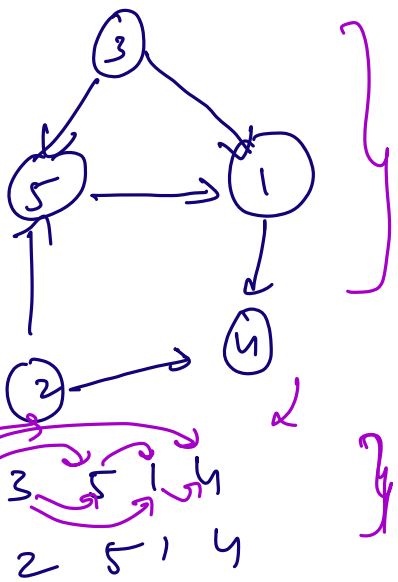
Resolving Depandancies.

Linear ordering of nodes such that there is a path from node i to node j iff node i is on the left side node 'j' in topo sort.

5 Courses where some courses have pre-requisites.

Graph

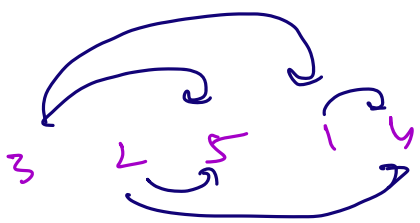
$u \rightarrow v$



preReq(1) = {3, 5}
preReq(2) = None

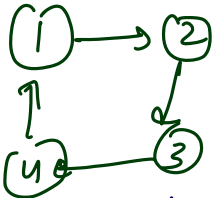
Ways : 2 3 5 1 4
3 2 5 1 4

} Multiple toposorts.



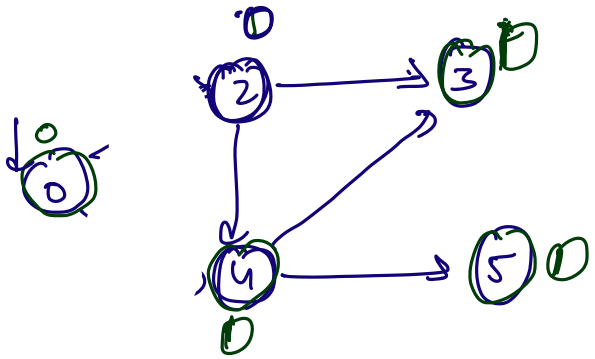
* Directed Graphs
* Acyclic Graph.

} Directed Acyclic Graphs (DAG)



→ Deadlock

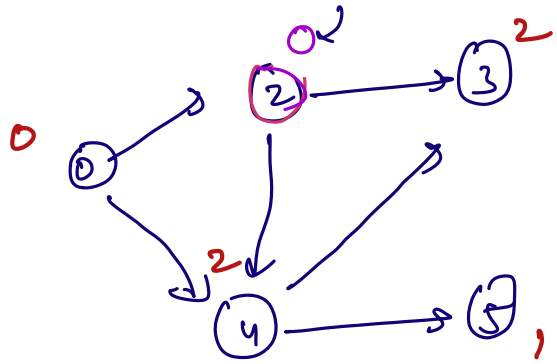
Approach: Kahn's Algorithm

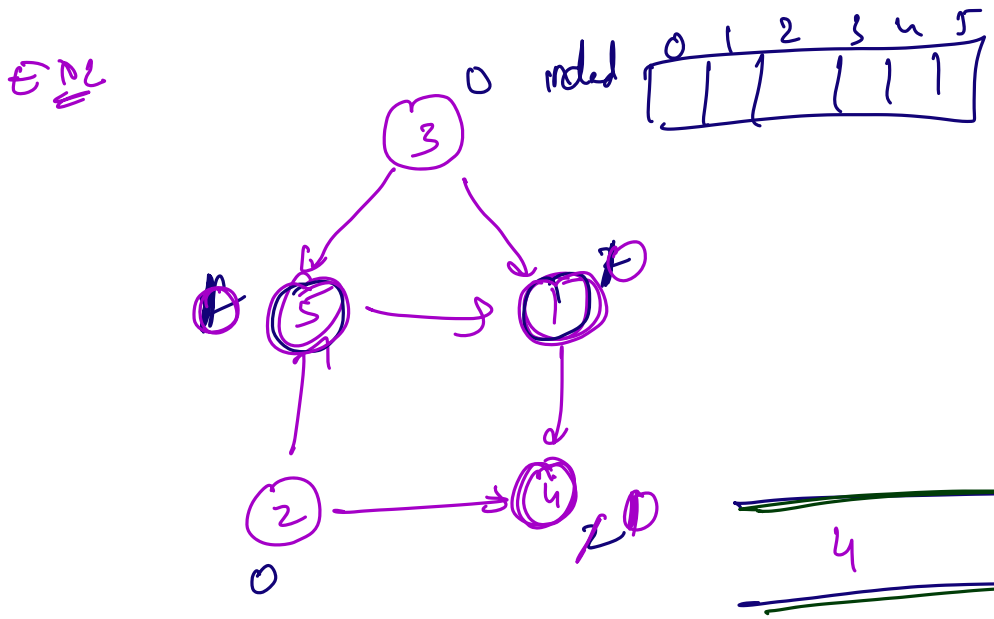


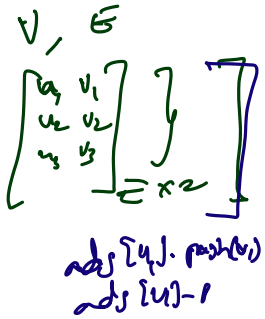
The 1st node of topo sort is a node with no incoming edges [Indegree]

Topo Sort: 0 2 4 3 5

0 2 4 5 3

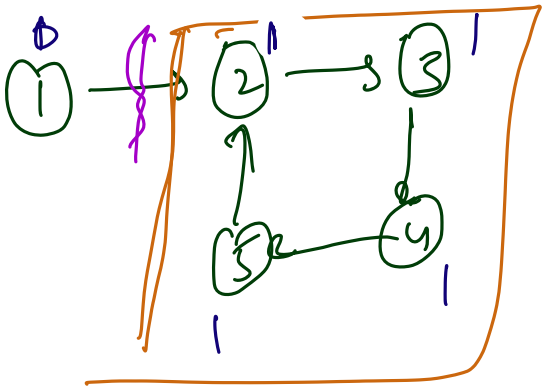






4

Output: 3 2 5 1 4



queue

One Strategy for detecting cycle in a directed graph


```
topoSort(){
      indegree[V] = {0};
      vector<int> adj[V];

      for(int i = 0; i < num_edges; i++){
            adj[B[i][0]].push_back(B[i][1]);
            indegree[B[i][1]]++;
      }
      queue<int> q;
      for(int i = 0 ; i < V; i++)
            if(indegree[i] == 0)
                  q.push(i);

      visited_nodes = 0;
      while(!q.empty()){
            int u = q.top();
            q.pop();
            visited_nodes++;
            ans.push(u);

            for( i in adj[u]){
                  indegree[i]--;
                  if(indegree[i] == 0)
                        q.push(i);
            }
      }
      if(visited_nodes != V)
            print("CYCLE");
      else
            print(ans)
}
```

O(E) { } Generating indegree Array

u → v

O(V) { }

O(V+E) { } BFS

TOPO Sort

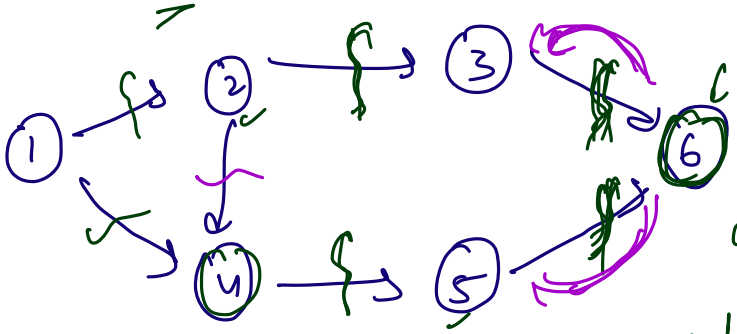
T.C: O(V+E) + O(V) + O(V) ≈ O(V+E)

S.C: O(V+E) + O(V) + O(V)

(V+E) → adj. (V) → Queue. (V) → indeg.

Approach2: DFS

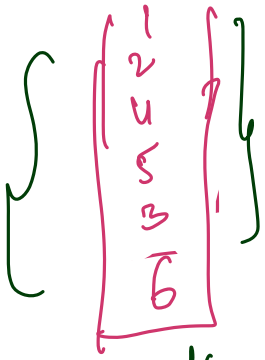
Topo Sort: 1 2 4 3 5 6
1 2 4 5 3 6
1 2 3 4 5 6



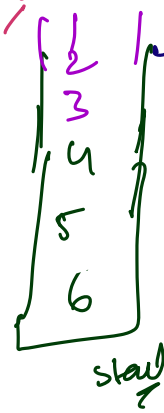
output: 6 3 5 4 2 1

Node with outdegree 0 would be the last node of topo sort.

→ If I take a stack and push all these elements into stack;

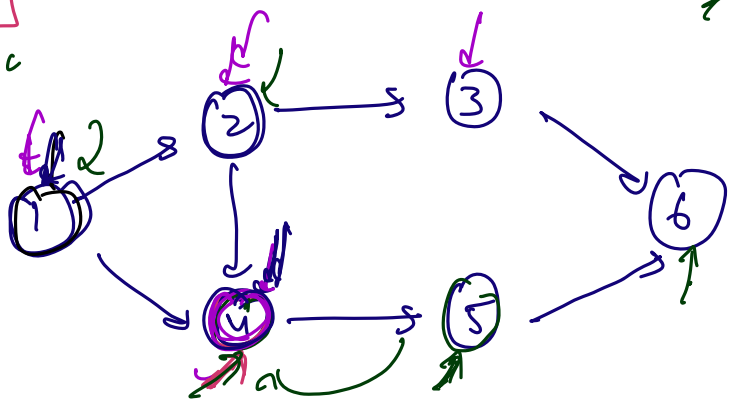


Contents of stack would be my topo sort.



⇒ 1 2 3 4 5 6

for(i=0; i<n; i++)
if(!visited)
dfs(i)



DFS(4)

Nodes Reachable from 4: {5, 6}

→ All nodes which are reachable from a node should be below the node in the stack

→ If a node is dependant on others,

dfs

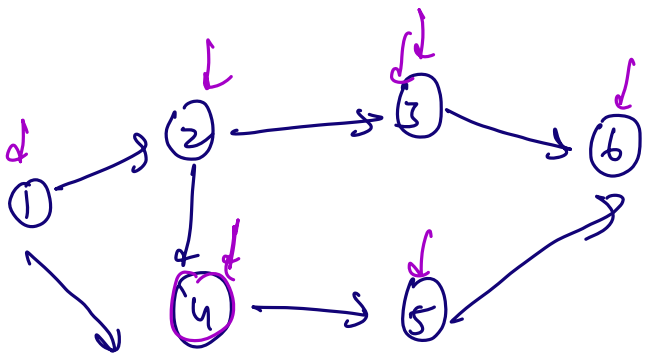
```
void topoSort(v){
      visited[v] = true;
      for(i in adj[v]){
            if(!visited[i])
                  topoSort(i);
      }
      stack.push(v);
}
main(){
      for(int i = 0 ; i < V; i++){
            if(!visited[i])
                  topoSort(i);
      }
      print(stack);
}
```

Only if we are sure there is no cycle

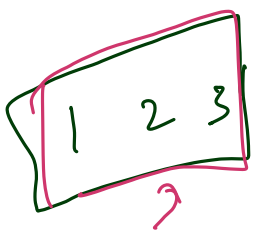
← Push on stack

for(i = 0 to V)

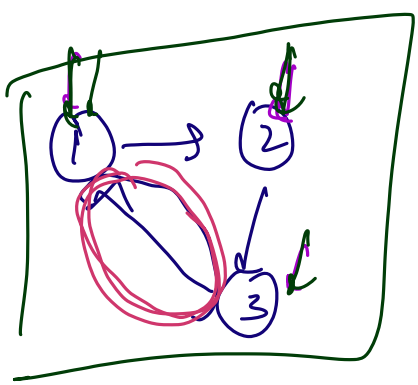
visited

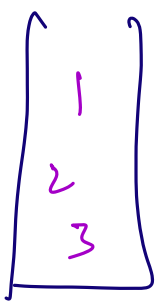


stack



1 2 4 5 3 6





1 2 3 ?

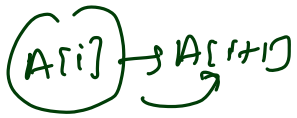
Applicable only if graph does not have a cycle.

T.C. $O(V+E)$

S.C. $O(V) + O(V+E)$

A = [1 3 4 5 6 7]

Edge from a[i] → a[i+1]



for(i=0; i ≤ _ ; i++) {

// Edge from A[i] to A[i+1]

adj[A[i]].push(A[i+1]);

}

```
bool isCyclic(int node, int parent){
     visited[node] = true;
     for(auto i : adj[node]){
          if(!visited[i]){
               if(isCyclic(i, node))
                    return true;
          }
          else if (i != parent)
               return true;
     }
return false;
}

main(){
     for(int i = 0; i < N; i++){
          if(!visited[i])
               if(isCyclic(i, -1))
                    return true;
     }
return false;
}
```

Topo Sort using DFS

```
void topoSort(v){
     visited[v] = true;
     for(i in adj[v]){
          if(!visited[i])
               topoSort(i);
     }
     stack.push(v);
}
main(){
     for(int i = 0 ; i < V; i++){
          if(!visited[i])
               topoSort(i);
     }
     print(stack);
}
```

Kahn's algo

```
topoSort(){
      indegree[V] = {0};
      vector<int> adj[V];

      for(int i = 0; i < num_edges; i++){
            adj[B[i][0]].push_back(B[i][1]);
            indegree[B[i][1]]++;
      }
      queue<int> q;
      for(int i = 0 ; i < V; i++)
            if(indegree[i] == 0)
                  q.push(i);

      visited_nodes = 0;
      while(!q.empty()){
            int u = q.top();
            q.pop();
            visited_nodes++;
            ans.push(u);

            for( i in adj[u]){
                  indegree[i]--;
                  if(indegree[i] == 0)
                        q.push(i);
            }
      }
      if(visited_nodes != V)
            print("CYCLE");
      else
            print(ans)

}
```