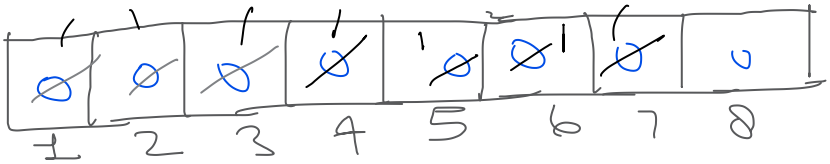
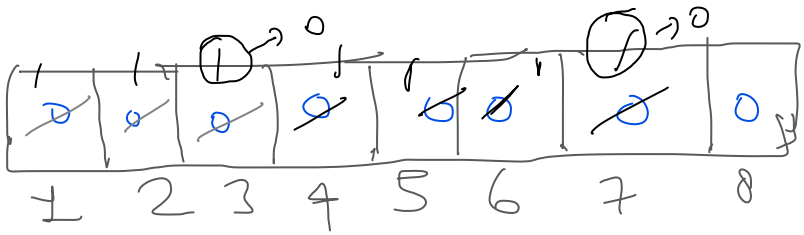
Visited :



dfsvisited :



dfs(1) : visited = true, dfs visited = true

↓

dfs(2) : V = true, dv = true

↓

dfs(3) : V = true, dv = true

↓ ↳ dfs(8)

dfs(7) : V = true, dv = true

↳ false

→ return to 3

dfs(3) : V = true (already),

dv = true

↳ false

→ return to 2

dfs(2) : V = true (already)

dv = true already

↓

dfs(4) : V = true, dv = true.

Adj list

1 : 2

2 → 3, 4

3 → 7, 8

7 → .....

8 → 7

4 → 5

5 → 6

6 → 4

↓

dfs(5): v = true, dv = true

↓

dfs(6): v = true, dv = true

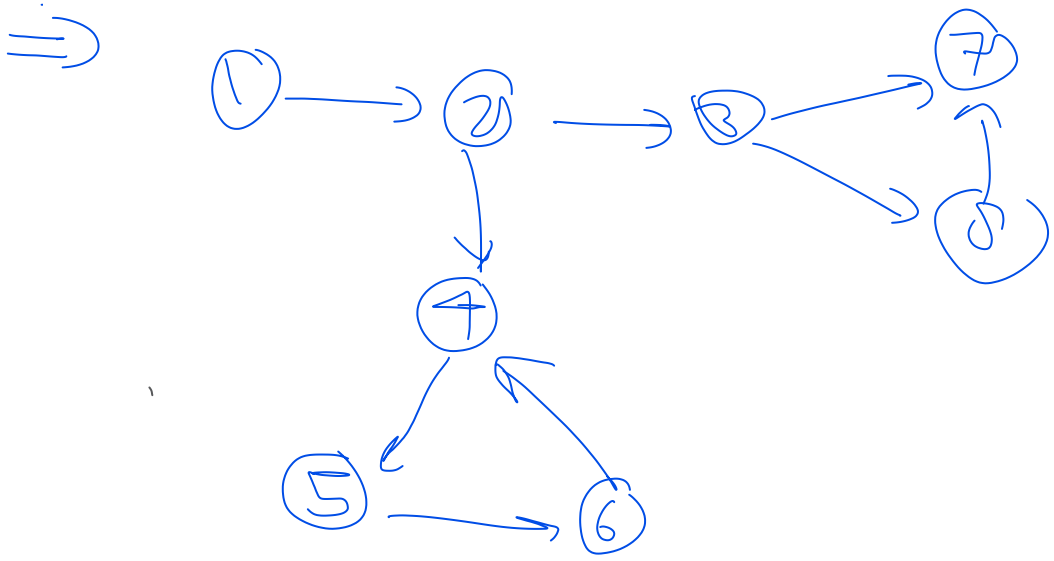
↓

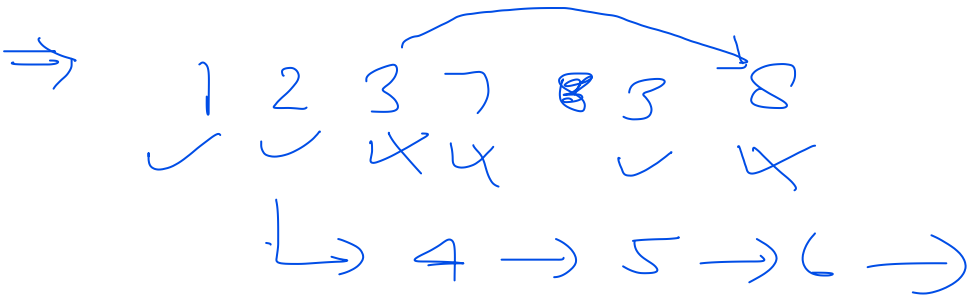
dfs(4): v = already true & 6 is not parent of 4, Hence cycle detected

→ the key concept behind this, is that

① we will call a dfs, which will go in depth and this will check if current element has previous one as parent & not visited,

② If an element doesn't have any outwarding edge to this we will return to back & we will mark dfsvisited again false.





- Problem if we apply logic of undirected graph -

⇒ 1 2 3 7 3 8 7

Here ↩

(7 is visited but not parent of 8)

→ So according to previous condition this should be a cycle

So we needed a different strategy,

which Considers one more visited array named "dfsvisited.", this will keep (DV) track of calls made by dfs for the

node.

⇒ we will do this so that if our function once visit the node & again return to its parent which is having outwarding edge towards a different element,

In this way we can avoid the case of visited of of that node because of which we were get misleaded.

So we need to check both visited & dfsvisited.

So Condition =

```
if (vis[node] = True & dfsvis[node]=T)
    return true.
```

```
for(int i=0; i<n; i++)
    if (!visited[i])
        bool ans = dfs(adj, vis, i, -1)
        if(ans){
```

} to handle case of Disconnected graph

```
        return true;
    }
}
return false.
```

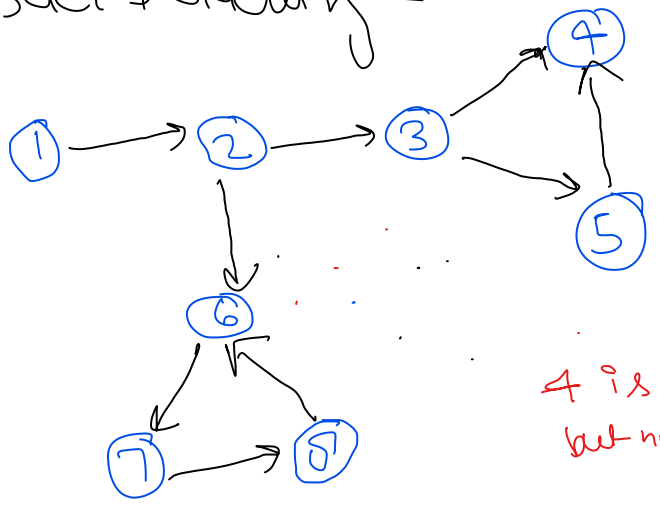
(2) Create Adjlist

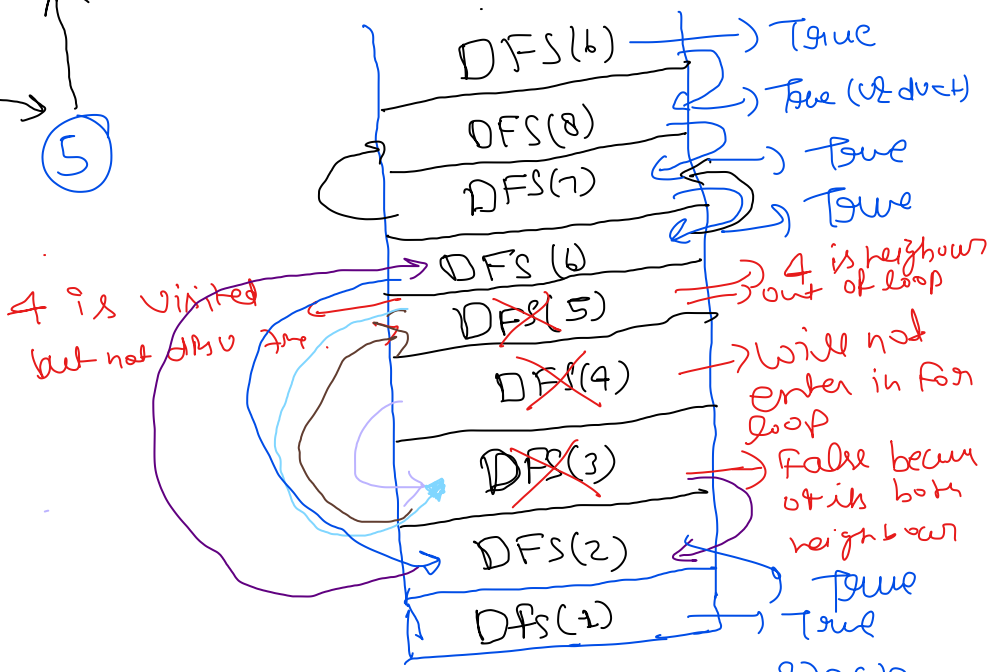
(3) Cycle check condition:

```
visited[node] = true;
dfsvisited[node] = true;
for(auto i : adjlist)
{
    if(!visited)
    {
        bool cycle = dfs(v, du, i, node)
        if (cycle)
            { return true }
    }
    else if (visited[i] && dfsvisi[i])
        {
            return true;
        }
}
dfsvisi[node] = false;
return false;
```

Imp

* Dry run the code with understanding of Recursion & Backtracking –





So final answer = True

- Adjlist –

1 → 2
2 → 3, 6
3 → 4, 5
4 → ....
5 → 4
6 → 7
7 → 8
8 → 6

- Point to point Discription –

① DFS(1) → (V = True, dV = True) → 2 is not visited so mark call DFS(i), where i is nothing but neighbour of 1, Here neighbour = 2,

② DFS(2) → (V(2) = True, dV(2) = True) → 3, 6 not visited

let's choose 3 & make a call DFS(3),

③ DFS(3) → (v[3] = T, dv[3] = T) → its neighbour 4 is not visited so call DFS(4).

④ DFS(4): (visi[4] = T, dfsv[4] = T) → no neighbours, we will not go through loop so (dfsv[4] = false) return false.

↳ This will make backtrack. & we will return to the its parent 3, now check other neighbour of 3 that is 5,

so call DFS(5)

⑤ DFS(5): (vis[5] = T, dv[5] = T) → 4 is neighbour, which is already marked visited so from condition out of loop this will return false (for backtrack) with marking 'dfsvis[5] = false', so return to its parent DFS(3).

⑥ DFS(3): out of loop both of its neighbours, visited with (dfsvis) so this will return to DFS(2), by making DFS(3) false.

⑦ DFS(2): 2 → 6, now we will go with DFS(6).

⑧ DFS(6): mark vis[6] = T & dfsvi[6] = True,
call DFS(7)

⑨ DFS(7): mark it visited, call DFS(8).

⑩ DFS(8): Since 6 is (visited & dfsvisited), both are True, so return true with base condition.

now backtracking

⑪ DFS(8) = True

↳ DFS(7) & if(DFS(8)) {return True}

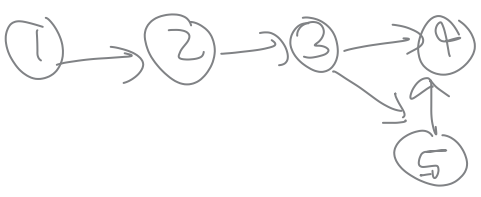
↳ DFS(6) { if(DFS(7)) { return True}

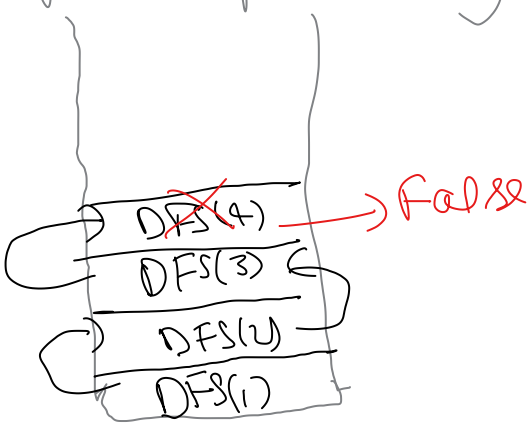
↳ DFS(2) { true}

↳ DFS(1) { true}

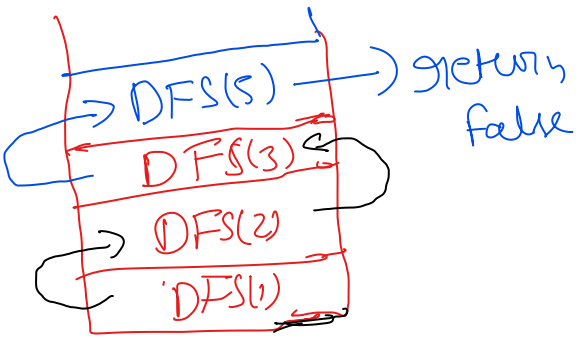
so true

- lets take another type of Graph & Dry run -





So Stack will look like



DFS(5): visited but not dfsVisited.

So false.

⇒ DFS(5) = false

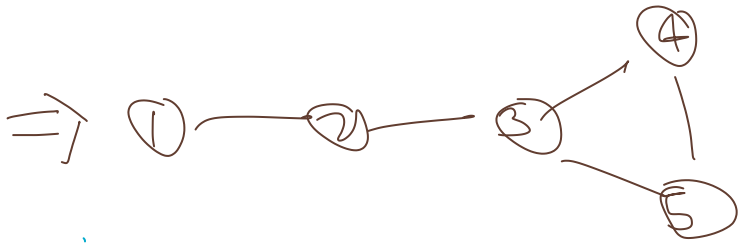
↳ DFS(3) = false

↳ DFS(2) = false

↳ DFS(1) = false

So return false

- lets write code for unordered graph using DFS.



⇒ DFS(1) → DFS(2) → DFS(3) → DFS(4) → DFS(5)

DFS(3)

↳ if 3 is visited & 5 is parent of 3 so true

- So we can write function like -

```
CCycle( adj, vis, src, parent)
{
    vis[src] = true;
    for (auto i: adj)
```

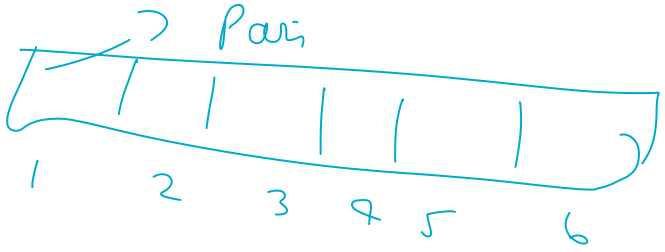
```
{
    if (vis[i] != True)
    {
        bool cycle = ccycle(i, src);
        if (cycle) { return true; }
    }
    else if (vis[i] && i != parent[i])
    { return true; }

    return false.
}
```

else → Parent

<-, ->

- vector < pair <int, int>> edge -

⇒ edge[i].first
edge[i].second



Lecture 6:

* Topological Sort :

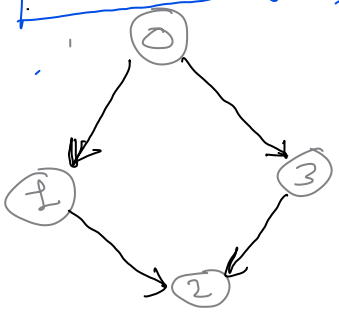
① DAG (Directed Acyclic Graph)

② linear Ordering gives. of vertices such that

For every edge u→v, u always appears before v.

→ let's take an example-

⇒ 0, 1, 3, 2 → valid TS (as 0 is before 1 & 3, 1 is before 2, 3 is before 2)



Topological sort.

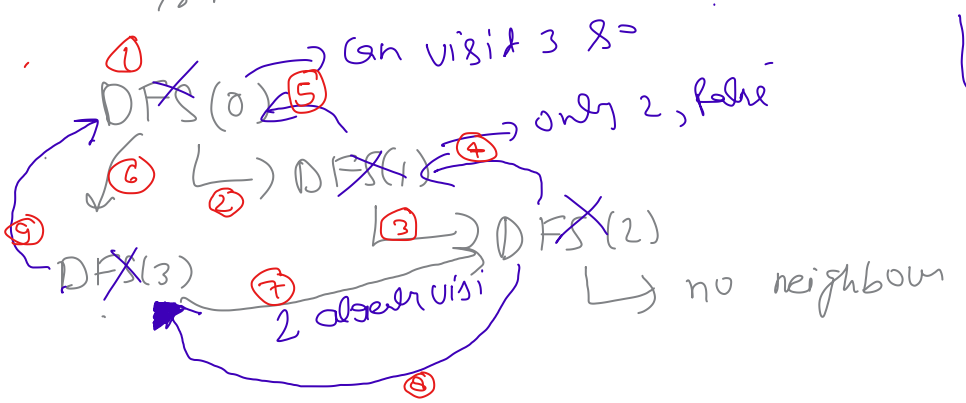
0 → 1, 3
1 → 2
2 → ...
3 → 2

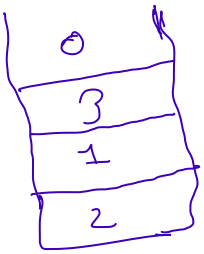
→ Adjlist

3 2 1 0 → not a valid T.S.

By DFS call

① If node is not visited, mark it visited, & call its neighbour, if no neighbour return false & return to its parent call, by putting it in Stack, do same for the other calls.





So code will be like.

```
• void topoSort(adj, &visited, &S, src)
  {
      visited[src] = true;
  → for (auto i: adj[src]){
        if(!visited[i])
            DFS(i)
    }
  → S.push(src);
  }
```

→ Same to handle topological sort

→ return false; → no need → void return type

}

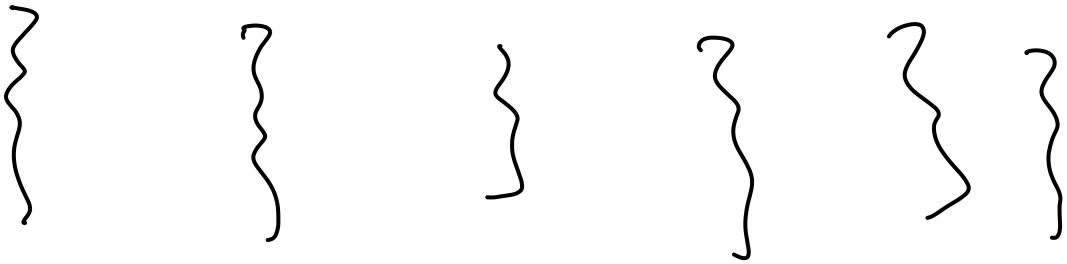
- in given main():

```
→ vector<int> ans;
→ Stack<int> s;
→ for(int i=0; i<v;i++)
        {  if(!visited[i])
              toposort(i,v,adj,s);
        }
→ while(!s.empty) {
        ans.push(s.top());
      }
→ return ans;
```

- Code for Topological Sort -

```cpp
#include <bits/stdc++.h>

void prepareAdj( unordered_map<int, list<int>> &adjlist, vector<vector<int>> &edges){
    for(int i=0;i<edges.size();i++){
        int u = edges[i][0];
        int v = edges[i][1];

        adjlist[u].push_back(v);
       //  adjlist[v].push_back(u);
    }
}

void toposort(unordered_map<int, list<int>> &adjlist, unordered_map<int, bool> &visited, int src, stack<int> &s){

    visited[src] = true;

    for(auto i: adjlist[src]){
        if(!visited[i]){
            toposort(adjlist, visited, i, s);
        }
    }
    s.push(src);

}
vector<int> topologicalSort(vector<vector<int>> &edges, int v, int e)  {
    // Write your code here
    unordered_map<int, list<int>> adjlist;
    unordered_map<int, bool> visited;
    vector<int> ans;
    stack<int> s;

    prepareAdj(adjlist, edges);

    ///Handling case for Disconnected Graph
    for(int i=0;i<v;i++){
        if(!visited[i]){
            toposort(adjlist, visited, i, s);
        }
    }

    while(!s.empty()){
        ans.push_back(s.top());
        s.pop();
    }

    return ans;

}
```

- Code for to Detect cycle in Directed graph Using DFS-



```
#include<bits/stdc++.h>

void prepareList(unordered_map<int, list<int>> &adjlist, vector < pair < int, int >> & edges ){
   for(int i=0;i<edges.size();i++){
    int u=edges[i].first;
    int v=edges[i].second;

    adjlist[u].push_back(v);
  }
}

// void printList(unordered_map<int, list<int>> adjlist){
//   for(auto i:adjlist){
//     cout<<i.first<<"--> ";
//     for(auto j: i.second){
//       cout<<j<<" ";
//     }
//     cout<<endl;
//   }
// }

bool isCycle(unordered_map<int, list<int>> &adjlist, unordered_map<int, bool> visited, int src,unordered_map<int, bool>
&dfsvisit){

  visited[src] = true;
  dfsvisit[src] = true;
  for(auto i: adjlist[src]){
    if(!visited[i]){

bool check = isCycle(adjlist, visited, i, dfsvisit);

if(check){
       return true;

}

}

else if(dfsvisit[i] ){
     return true;
   }
  }


dfsvisit[src] = false;

return false;
}
int detectCycleInDirectedGraph(int n, vector < pair < int, int >> & edges) {

  // Write your code here.

  unordered_map<int, list<int>> adjlist;
  unordered_map<int, bool> dfsvisit;
  unordered_map<int, bool> visited;
  prepareList(adjlist, edges);
  // printList(adjlist);

  //to handle case of components
  for(int i=0;i<n;i++){
    if(!visited[i]){
      bool ans = isCycle(adjlist, visited, i, dfsvisit);
```

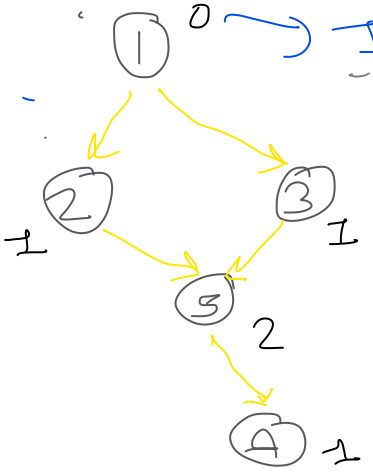
```
        if(ans){
            return 1;
          }
        }
      }
      return 0;
    }
```

---

lecture 7:

- Topological Sort using BFS:

[kahn's Algo]



Indegree-

0	1	1	1	2
1	2	3	4	5

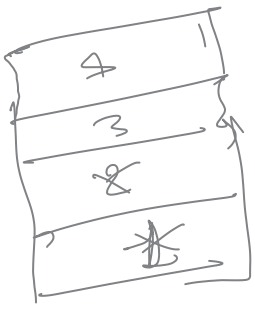
- Strategy, we will apply here.

→ ① Get InDegree to all vertex

→ ② Do bfs in a way such that we target the element with '0' Indeg ie & find its neighbour to perform operations

1 → done

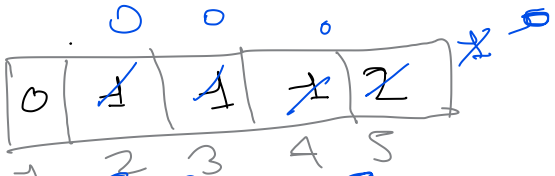
& decrease Indegree by 1 for the 2, 3 neighbours because if we pop() its parent one edge will be remove to all its neighbour.

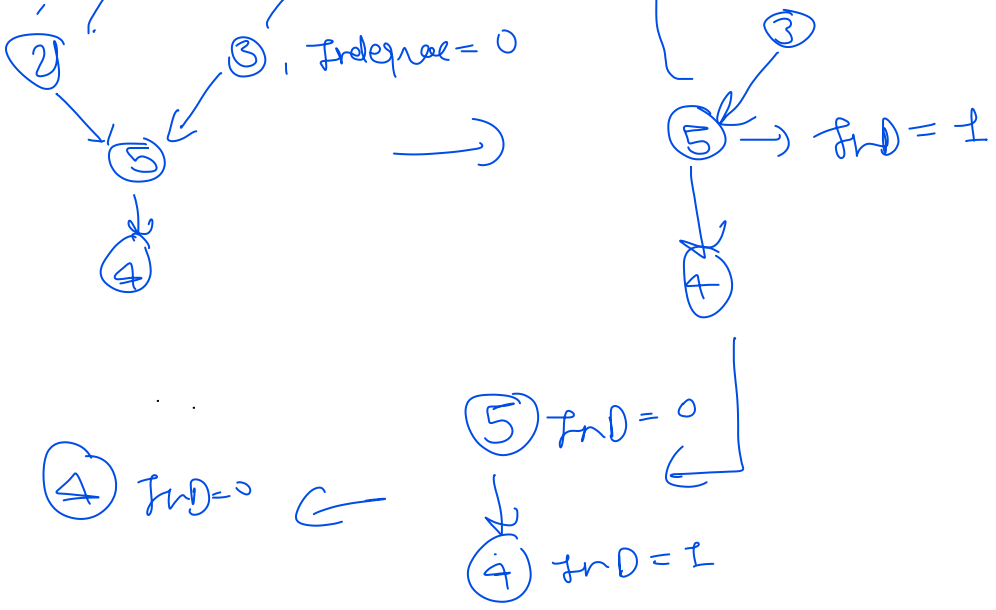


→ ③ Push element in arr.

→ (9)

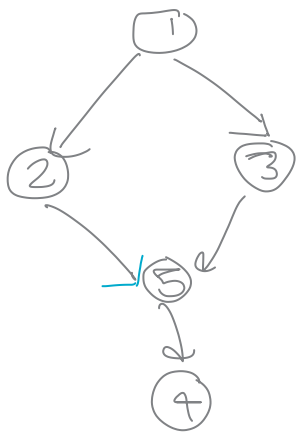
Delete (1)





(7) → To get InDegree for all vertex

↳



1 → 2, 3
2 → 5
3 → 5
4 → 0....
5 → 4.

⇒ for (auto i : adj)

{ for (auto j : i.second)

{ Indegree[j]++

}.

}

→ toposort(adj, visited, src, q)

{

visited[src] = true;

q.push[src] =

while (!q.empty)

{ if (indegree[src]==0){

int f_N = q.front(); q.pop();

for(auto i: adj[f_N])

{

if(!visited)

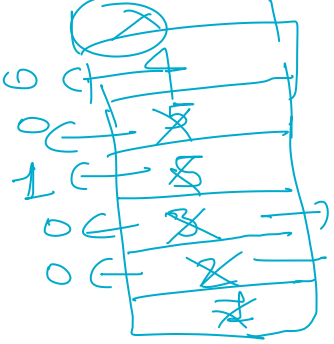
{ visited[i]= true;

q.push(i);

indeg[i]--

}

}



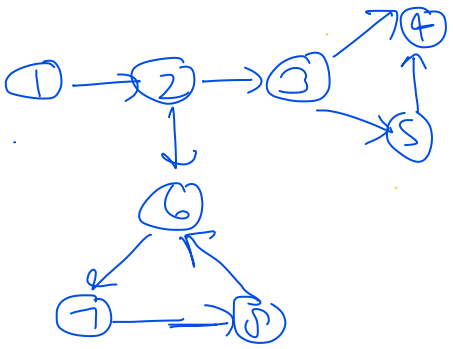
```
#include <bits/stdc++.h>
using namespace std;

// Function to prepare the adjacency list from edges
void prepareAdj(unordered_map<int, list<int>> &adjlist, vector<vector<int>> &edges) {
    for (int i = 0; i < edges.size(); i++) {
        int u = edges[i][0];
        int v = edges[i][1];
        adjlist[u].push_back(v);
    }
}

// Function to perform BFS-based topological sort
void toposort_BFS(unordered_map<int, list<int>> &adjlist, unordered_map<int, bool> &visited, queue<int>
&q, vector<int> &indegree, vector<int> &ans) {
    while (!q.empty()) {
        int front_node = q.front();
        q.pop();

        if (!visited[front_node] && indegree[front_node] == 0) {
            ans.push_back(front_node);
            visited[front_node] = true;

            for (auto i : adjlist[front_node]) {
                if (!visited[i]) {
                    indegree[i]--;
                    if (indegree[i] == 0) {
                        q.push(i);
                    }
                }
            }
        }
    }
```

```
// Function to perform topological sort on the graph
vector<int> topologicalSort(vector<vector<int>> &edges, int v, int e) {
    unordered_map<int, list<int>> adjlist;
    unordered_map<int, bool> visited;
    vector<int> ans;
    vector<int> Indegree(v, 0); // Initialize indegree vector with size v and all values 0
    queue<int> q;

    prepareAdj(adjlist, edges);

    // Find indegree for all nodes
    for (auto i : adjlist) {
        for (auto j : i.second) {
            Indegree[j]++;
        }
    }

    // Push elements with 0 indegree
    for (int i = 0; i < v; i++) {
        if (Indegree[i] == 0) {
            q.push(i);
        }
    }

    // Handling case for disconnected graph
    for (int i = 0; i < v; i++) {
        if (!visited[i] && Indegree[i] == 0) {
            toposort_BFS(adjlist, visited, q, Indegree, ans);
        }
    }

    return ans;
}
```

* lecture 8:

Detect Cycle in Directed Graph using BFS-

⇒



⇒ ① We will apply here a simple method, we will check if topological sort gets successfull here then this will give a DAG.

② For that we will maintain a count & increase it every time while getting front element from queue & pop() that.

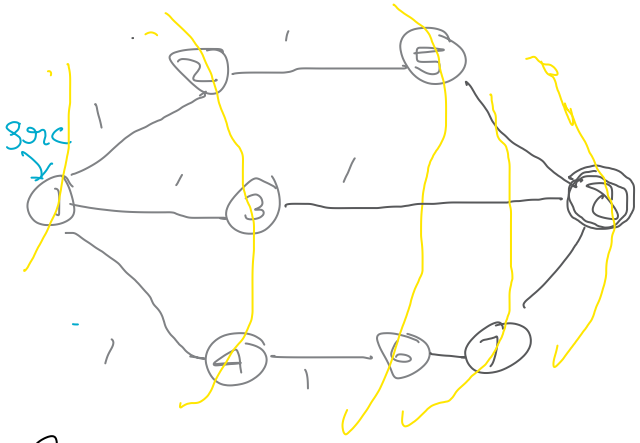
③ We will check if (count == n) that means DAG or cycle not present else cycle present.

So Code —

```
checkCycle(adj, q, Indegree, n)
{ while(!q.empty()) {
    int front_node = q.front();
    q.pop();
    cnt++;
```

```
for(auto i : adj[front_node])
    {
        indegree[i]--;
        if(indegree[i]==0)
            q.push(i);
    }
}
if(cnt==n)
    return false;
else return true;
}
```

lecture-8: Shortest path in Undirected Graph:

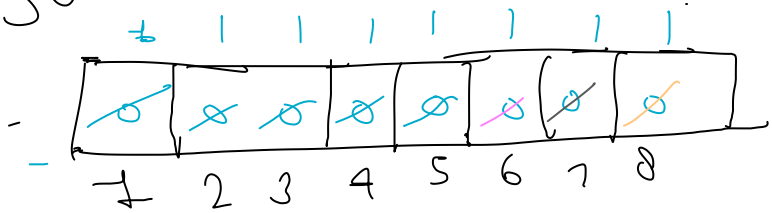


1 → 2, 3, 4
2 → 5, 1
3 → 8, 1
4 → 6, 1
5 → 8, 2
6 → 7, 4
7 → 6, 8
8 → 5, 7

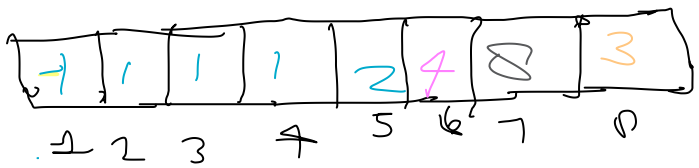
- Path 1: 1 → 2 → 5 → 8
- Path 2: 1 → 3 → 8
- Path 3: 1 → 4 → 6 → 7

* Approach: we will get through BFS which works level by level so gives shortest path.
→ we need to maintain one extra DS here which will keep track of parent of each node

So

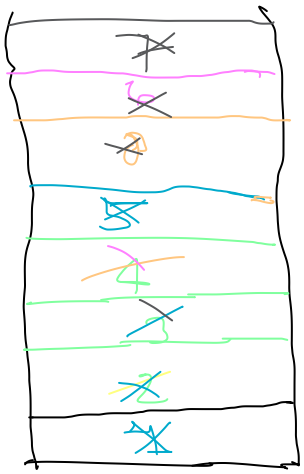


= Visited



= Parent

So 8 → 3 → 1



Src = 1

So path

or

So what we gonna do here is -

→ while(!q.empty())

{

f_N = q.front();

q.pop();

for (auto i : adj[f_N])

{ if(!visited[i])

q.push(i);

Parent[i] = f_N;

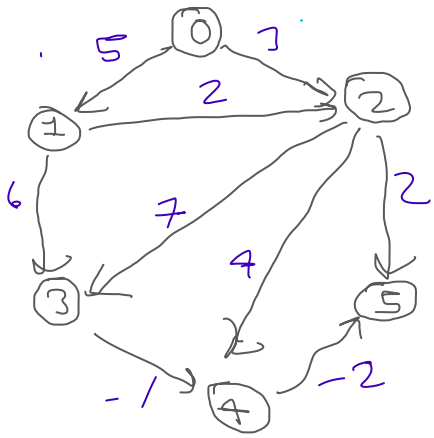
}

}

```
vector<int> shortestPath( vector<pair<int,int>> edges , int n , int m, int s , int t ){


 unordered_map<int, list<int>> adj;
 unordered_map<int, bool> visited;
 unordered_map<int, int> parent;
 vector<int> ans;


 // Create adj list
 for(int i=1; i<edges.size();i++){
  int u = edges[i].first;
  int v = edges[i].second;

  adj[u].push_back(v);
  adj[v].push_back(u);
 }


 queue<int> q;
 q.push(s);
 visited[s] = true;
 parent[s] = -1;

 while(!q.empty()){
  int f_n = q.front();
  q.pop();

  for(auto i: adj[f_n]){
   if(!visited[i]){

    visited[i] = true;
    parent[i] = f_n;
    q.push(i);
   }
  }
 }


     // short_path(adj, visited, parent, s, t);

        // for (auto i : parent) {
        //         cout << i.first << " " << i.second<<endl;

        // }

 // Prepare shortest Path

 int curr_node = t;
 ans.push_back(t);

 while(curr_node != s){
  curr_node = parent[curr_node];
  ans.push_back(curr_node);
 }

 reverse(ans.begin(), ans.end());
 return ans;


}
```

(giving the problem of Time exceed).

* lecture 10 : Shortest path in Directed Acyclic Graph :



① Topological Sorting

② Distance array update.

So

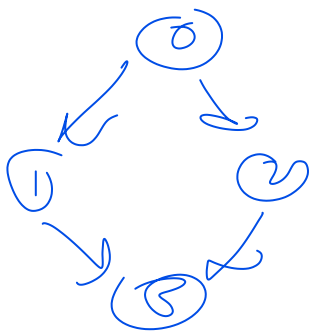
adjlist -

- 0 → [1,5], [2,3]
- 1 → [3,6], [2,2]
- 2 → [3,7], [4,4], [5,2],
- 3 → [4,-1]
- 4 → [5,-2]
- 5 → . . . . .

So

unordered_map<int, list<int,int>> adj;

⇒ for(auto i : adj)
{
if(!visited[i])
DFS(i)
}
S.push(src);

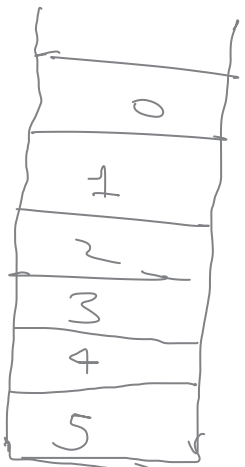


Stack

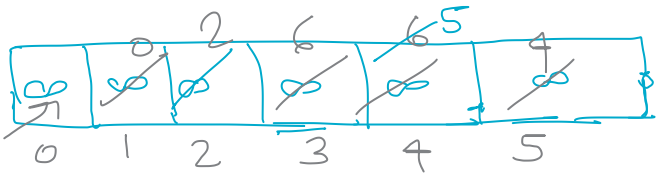
visited

By calling DFS.

⇒ for(i : adj[src])
{if(!visited[i])
DFS(i);
}
S.push(i);



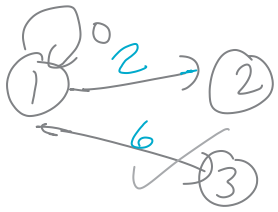
So.

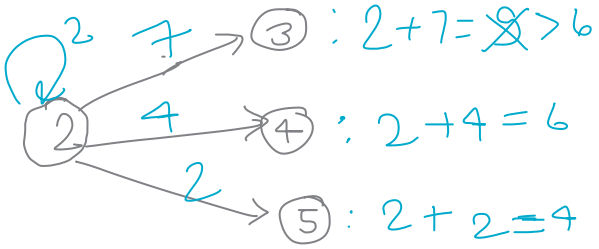


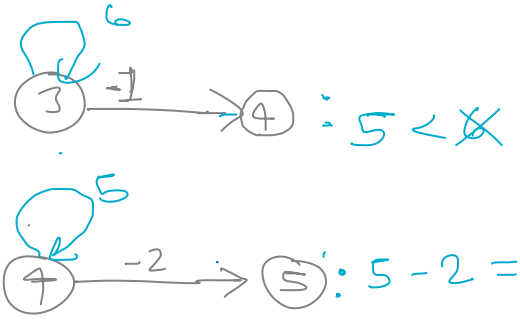
now.

⇒ top = 0 (ignore), Here src = 1 & dest = 5

So top = 1,







1 → 0 : ∞

1 → 1 : 0

1 → 2 : 2

1 → 3 : 6

1 → 4 : 5

1 → 5 : 3

So Shortest path : (1 —6→ 3 —-1→ 4 —-3→ 5)

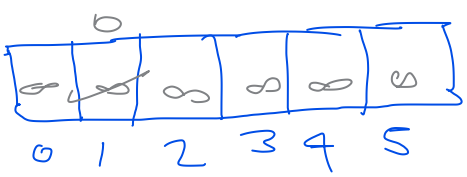
Cost = 3.

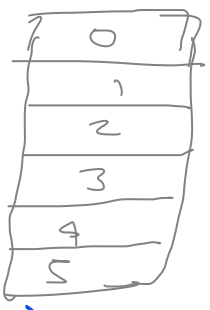
So Shortest path → {inf, 0, 2, 6, 5, 3}

So Shortest = 1 → 5 : 3

---

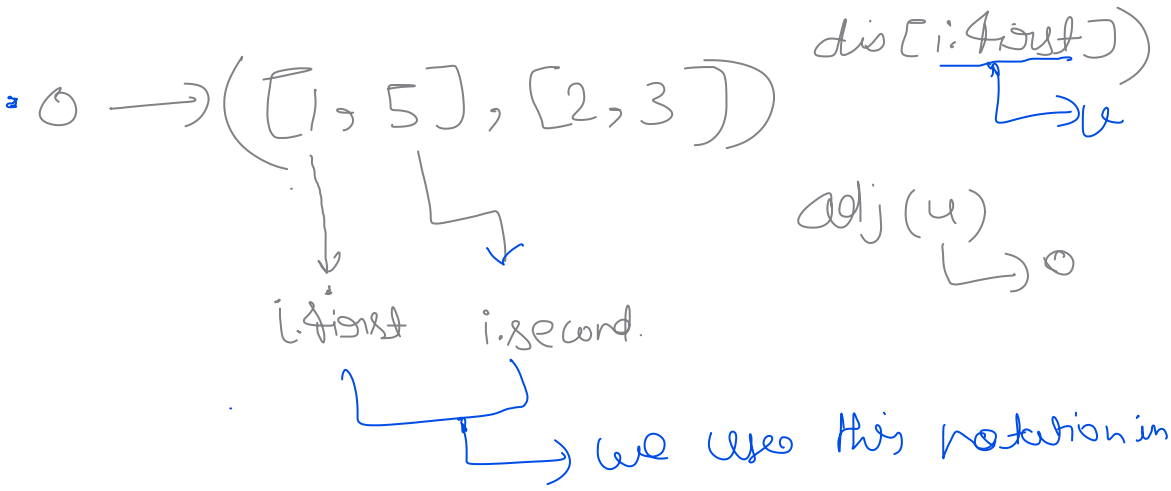
dist =





→ top = 0 ; dis[0] = 0

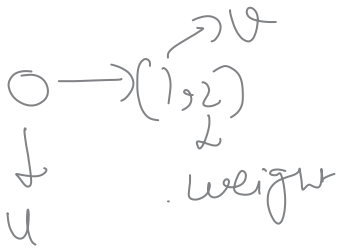
→ top = 1, dis[1] = ∞ 4 if (dis[0] + i.sec <



⇒ So if (dist[u] + w < dist[v])

dist[v] = dis[u] + weight.

→ Here, { v = i.first.
u = s.top()
weight = i.second



```cpp
#include<bits/stdc++.h>
#include<map>
using namespace std;
class Graph{
    public:
    void prapre_list(int u, int v, int w, unordered_map<int, list<pair<int,int>>> &adj){
        pair<int, int> p = make_pair(v,w);
        adj[u].push_back(p);
    }

    void topo(unordered_map<int, list<pair<int,int>>> &adj, stack<int> &s, unordered_map<int, bool> &visit,int &src){
        int top = src;
        visit[top] = true;
        /// agr ye adj ka leta to poora pointer aata but aj[top] mtlb
        /// u is given there, all the things will work for j then;

        for(auto i: adj[top]){
            int u = top;
            int v = i.first;
            int w = i.second;
            if(visit[i.first] == false){
                topo(adj, s,visit, i.first);
            }

        }
        s.push(src);
    }

    void getShortest(stack<int> &s, int src, vector<int> &distance, unordered_map<int, list<pair<int,int>>> &adj){

        distance[src] = 0;

        while(!s.empty()){
            int u = s.top();
            s.pop();
            if(distance[u] != INT_MAX){
                for(auto i: adj[u]){
                int v = i.first;
                int w = i.second;

                    if(distance[u] + w < distance[v]){
                        distance[v] = distance[u] + w;
                    }
                }
            }

        }
    }
};

int main(){
    unordered_map<int, list<pair<int,int>>> adj;
    Graph g;
    g.prapre_list(0,1,5,adj);
    g.prapre_list(0,2,3,adj);
    g.prapre_list(1,2,2,adj);
    g.prapre_list(1,3,6,adj);
    g.prapre_list(2,3,7,adj);
    g.prapre_list(2,4,4,adj);
    g.prapre_list(2,5,2,adj);
    g.prapre_list(3,4,-1,adj);
    g.prapre_list(4,5,-2,adj);
    int n = 6;
```

```
    for(auto i:adj){
        cout<<i.first<<"-->";
        for(auto j:i.second){
            cout<<"( "<<j.first<<","<<j.second<<" ),";
        }
        cout<<endl;
    }

    /// Creating topo
    stack<int> s;
    unordered_map<int, bool> visit;

    for(int i=0; i<n; i++){
        if(!visit[i]){
            g.topo(adj, s, visit, i);
        }
    }

    // printing topo
    while(!s.empty()){
        cout<<s.top()<<" ";
        s.pop();
    }
    cout<<endl;
    /// finding shortest path
    int src = 1;
    int dst = 5;

    vector<int> distance(6);
    for(int i=0;i<n;i++){
        distance[i] = INT_MAX;
    }


    g.getShortest(s, src, distance, adj);

    cout<<" answer: id"<<endl;

    for(int i=0;i<distance.size();i++){
        cout<<distance[i]<<" ";
    }cout<< endl;

}
```

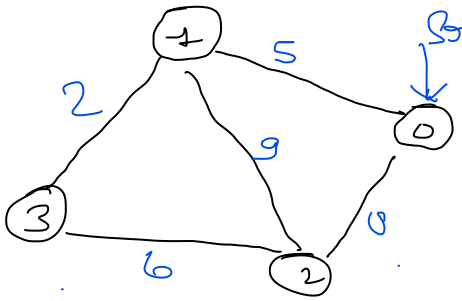
Printing of corresponding list.

Creating of Topo.

→ make all distance element to Infinity (∞).

---

lecture 11: Dijkstra's Shortest Path.



Adj list

0 → [1,5], [2,8]

1 → [0,5], [2,9], [3,2]

2 → [0,8], [1,9], [3,6]

3 → [1,2], [2,6]

0 → 0 : 0

0 → 1 : 5

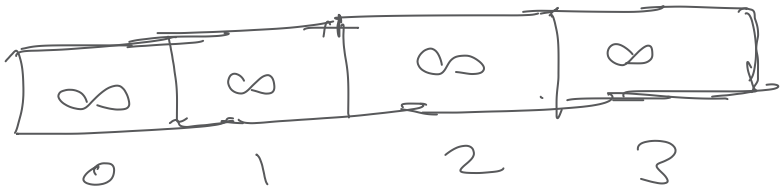
0 → 2 : 8

0 → 3 : (5+2=7)

So {0, 5, 8, 7}

⇒ Approach:

distance:



DS

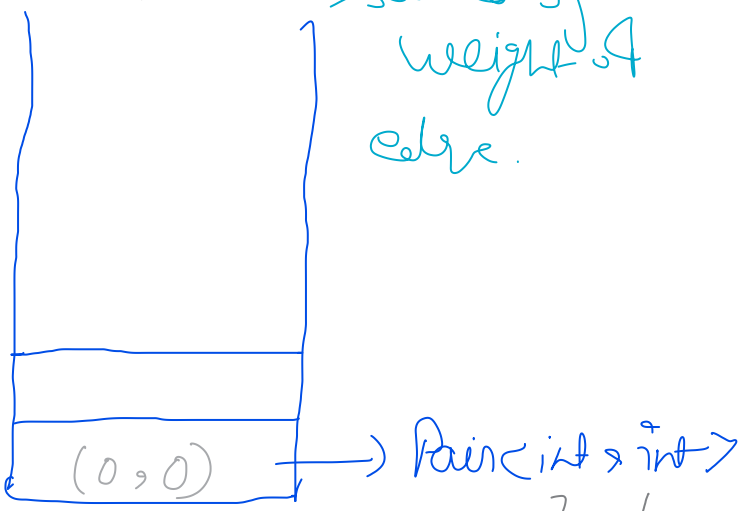
Priority queue

Set.

(A type of queue which stores element in sorted order)

So we will go by Set

Sorted by weight of edge.

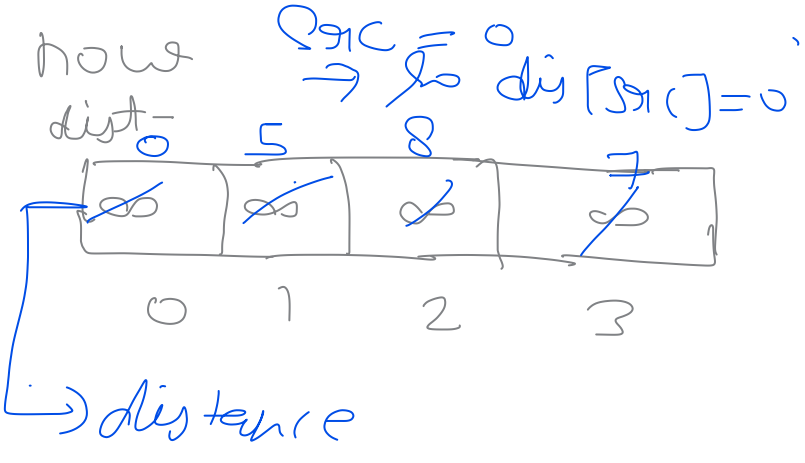


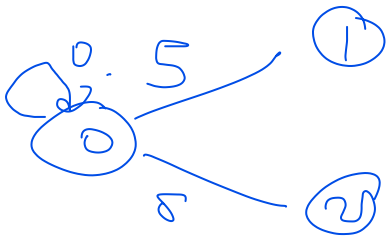
Set

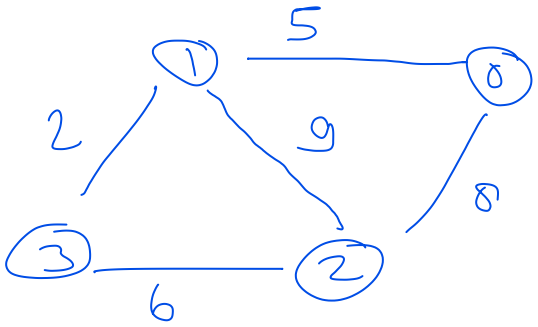
dist node

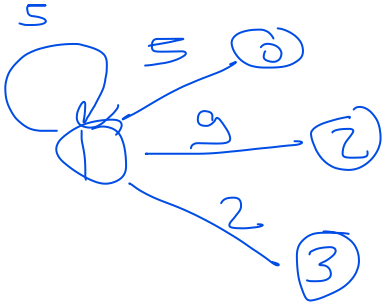
→ P = make_pair(0,0)

→ S.insert(P)









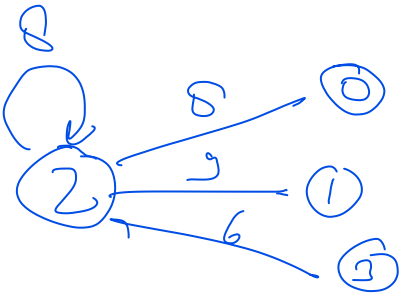
adj list

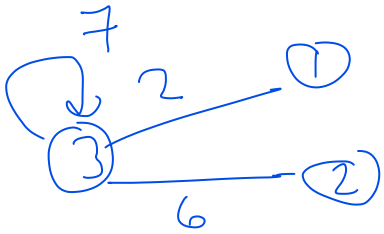
0 → [1,5], [2,8]

1 → [0,5], [2,9], [3,2]

2 → [0,8], [1,9], [3,6]

3 → [1,2], [2,6]





We can't use topo sort because undirected graphs are also allowed 4.

So Pseudo code -

(1) make Distance array give infinity to all. using ~~topo sort~~.

(2) Do relaxation to find shortest path for each node from src.

(3) Use priority queue or set to store pair (w, v), which will give weight in sortest order.

- adj
- visited<int, bool>
- distance<int, int>
- set<pair<int, int>>

So src → 0

dist[src] = 0

s.insert(make_pair(0, src))

dis[src]

→ call function(&s, &dist, &vis, &src)

top = src;

dis[top] = 0;

while(!s.empty())

{

int u = top;

if(!=)

for(auto i : adj[top])

{ int v = i->second;

int w = i->first;

if(!visited[v])

{ if(dis[v]+w<dis[v])

dis[v] = u+w

```
    }        s.push(u);
             }
```

→ 2D array

```
So given (vector<vector<int>> &vec, int
    vertices, int edge, int src){

}
```

→ see the question 0 1 7 (u, v, w)

→ at every u, we need to push (v & w) & from v, we need to push (u & w), max edge.

→ we will create distance vector with size of vertices, to store distances of all elements from source. & initially set '∞' to all

→ make iteration using priority queue/set, we will use set.

→ dis[src] = 0;

→ St.insert(make_pair(0, src));

* lets understand concept of sets-

→ to store unique element a specific order that is ascending by default, or this doesn't allow duplicate elements.

→ typically implemented as balanced binary search tree (usually a Red-Black tree).

→ Associative containers, meaning that they store element in a way that allows for fast retrieval based on keys.

- set<int> S → initialization
- S.insert(5); → inserting of element.
- S.erase(5); → erase '5'
- for (auto i = S.begin();
  i != S.end; i++)
  cout << *i << ;

```
while(!s.empty())
{
    auto top = *(s.begin());
}
```

→ we need pointer to access the elements.

→ make a while loop till (s.empty)
  access its first element named top;
→ find w(u,v) from top (which is a pair of w(u,v) only
→ erase s.begin().
→ traverse neighbour in adj with u,
→ Do relaxation by finding (v,w)
→ auto record = find(m-p(w(v), v))
→ if(u+w<v) then find (w,v) in set & erase that w(u)
→ update distance
→ insert it in set using pair. → linear
→ return distance.

T.C = O(E log V)

S.C = O(N+E)