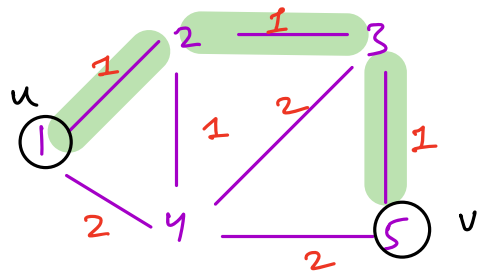
Graphs 3: Dijkstra Algo & Topological Sort

Question

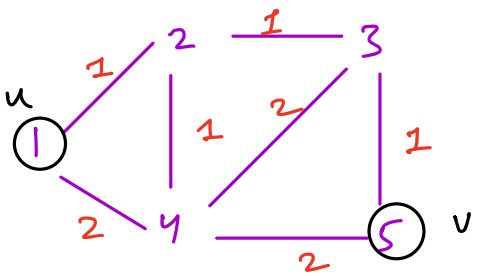
find min. weight to travel from u to v in a given connected simple graph.

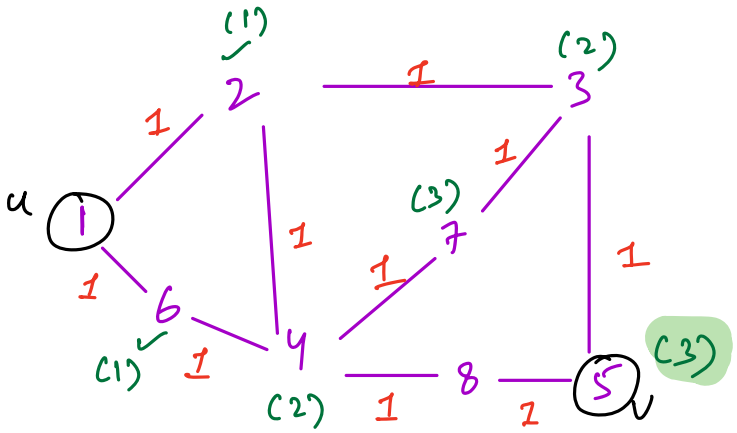
1 <= weight of an edge <= 2



ans = 3

modify the graph by creating dummy node b/w every 2 weight edges to make each edge weight as 1.





Code (for modifying graph)

```
void processEdges(N, int[][] A) {
    dummy_node = N+1;  // dummy_node is a var. name
    for(int[] edge : A) {
        u = edge[0], v = edge[1], w = edge[2];
        if(w == 1) {
            graph[u].add(v);
            graph[v].add(u);
        }
        else { // w=2
            graph[u].add(dummy_node);   // dummy node b/w u & v
            graph[dummy_node].add(u);
            graph[v].add(dummy_node);
            graph[dummy_node].add(v);

            dummy_node++;

        }
    }
}
```

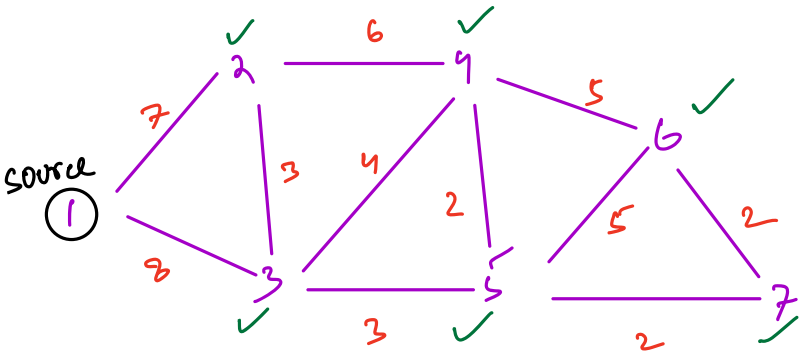
we can max. have V extra nodes.

TC = O(V+E)

SC = O(V+E)

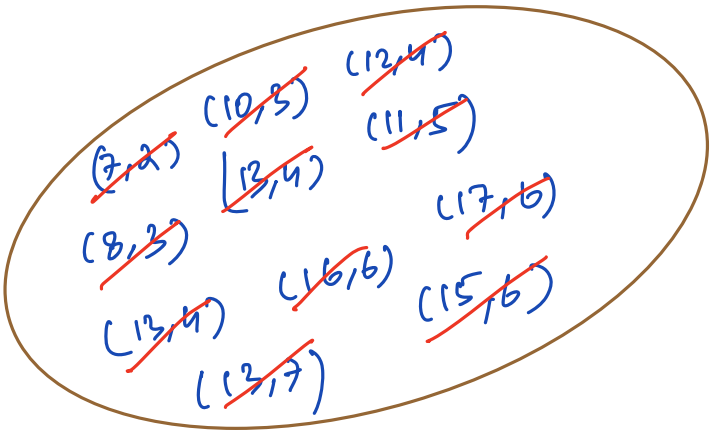
Question

There are N cities, you are living in city 1. Find min. distance to reach every other city from city 1. Return answer as array



1	2	3	4	5	6	7
0	7	8	12	11	15	13

distance array



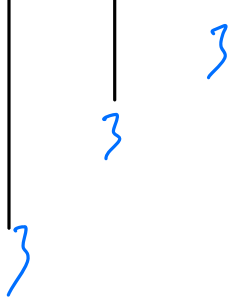
min Heap

{ edge wt. + d[curr node], connecting node }

```
int dis[N+1]
fi, dis[i] = ∞

Heap<int, int> h;

dis[1] = 0;

for( pair p : graph[1] ) {
    h.insert({ p.wt, p.v });
}
while( ! h.isEmpty() ) {
    pair p = h.getMin();
    d = p.wt; // distance
    v = p.v; // node
    if( dis[v] < d ) { // skip if this distance is greater
        continue;          than already known distance
    }
    dis[v] = d;
    for( pair x : graph[v] ) {
        new_dis = x.wt + dis[v];
        if( new_dis < dis[x.v] ) {
            h.insert( {new_dis, x.v} );
```

x.v is a node connecting to v

x.wt is their weight

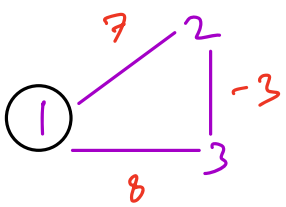


$TC = O(E \log E)$

$SC = O(E + V)$

if we have to create adjacency list

* Dijkstra algo does not work on negative weights



0	∞	4
1	2	3

(∞,2) (8,3) (1,2) (4,3) . . .

Negative weights lead to unexpected behaviour & give incorrect answer.

Question

Given N courses & pre-requisite of each course.

Check if it is possible to complete all courses.

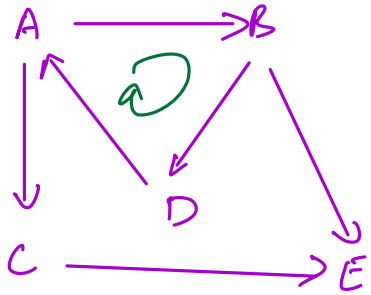
N = 5

A ⟶ B, C

B ⟶ D, E

C ⟶ E

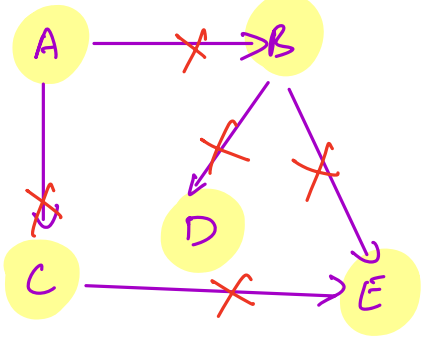
D ⟶ A



if cycle is present ⇒ not possible

else possible

2. If it is possible to complete courses, find any one order of completing the courses.



o/p → A B D C E

A C B D E ?

A B C D E } other possible orders

⋮

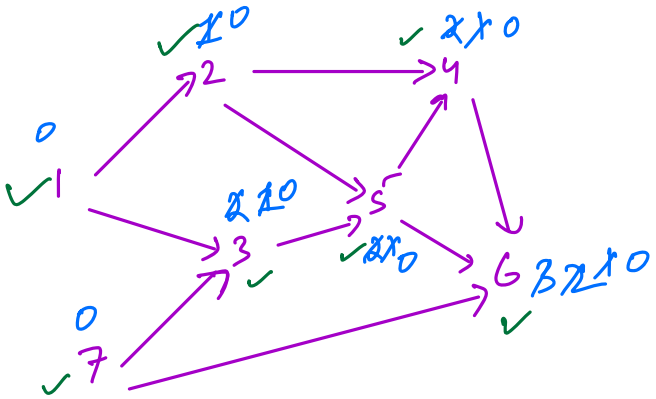
Topological Sort

Linear arrangement of nodes s.t. if there is any edge from $i \rightarrow j$ then i should come before j in answer.

Ques → How to find topological sort / order ?

1. Left to Right

No incoming edge ⇒ no pre-requisite (indegree = 0)



o/p → 1 2 7 3 5 4 6

Sol:

1. find indegree of all nodes
2. Store all nodes with indegree = 0 in any DS like (queue, hashset, ...)
3. Select any node from queue, print it as answer, iterate to all neighbours & reduce their indegree by 1.
   If indegree becomes 0, then add it in queue.

Code

```
in[V]
fi, in[i] = 0
for(u = 0 to V-1){
    for(int v : graph[u]){ // u → v edge
        in[v]++;
    }
}
Queue<int> q;
for(i = 0 to V-1){
    if(in[i] == 0) q.enqueue(i)
```

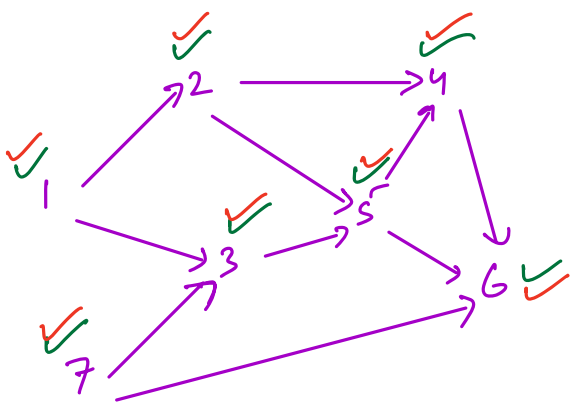
```
}
while( !q.isEmpty() ){
    x= q.dequeue()
    print(x);
    for( int v : graph[x]){
        in[v]--;
        if(in[v] == 0)  q.enqueue(v);
    }
}
```

$TC = O(V+E)$

$SC = O(V)$

2. Right to left

A node x can be printed if all the nodes that can be travelled from x are already visited.



o/p → 6 4 5 2 3 1 7

↓ reverse

7 1 3 2 5 4 6

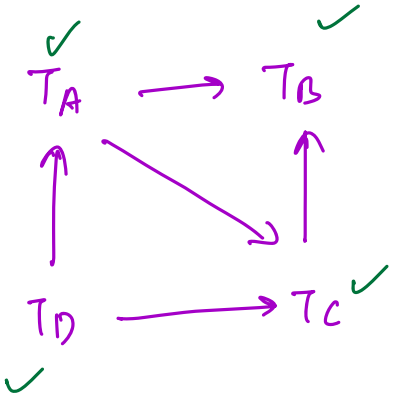
Code

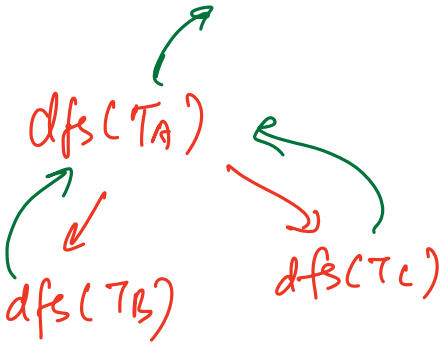
```
bool visit[V]
fi, visit[i] = false
for (i=0 to V-1) {
    if (!visit[i])
        dfs(i)
}
void dfs (u) {
    visit[u] = true
    for(int v : graph[u]){
        if (!visit[v])
            dfs(v)
    }
    print(u) → reverse topological order
}
```

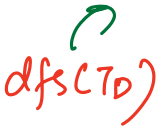
[store in array & reverse it in the end]

TC = O(V+E)

SC = O(V)







T_B T_C T_A T_D

↓ reverse

T_D T_A T_C T_B