Graph 2

AGENDA:

- BFS
- Multisource BFS
- Rotten Oranges
- Possibility of finishing courses
- Topological sort

Madhan Kumar M S
Anuj chandil
Balaji S K
Bhavesh Rathod
Burhan
Dewnash
Gagan Kumar S
Hemant Kumar
Nikhil Pandey
Prajwal Khobragade
Purusharth A
Rajat Sharma
Rajendra
shilpa mamillapalli
Sridhar Hissaria
Sumit Adwani
Suyash Gupta
Vimal Kumar

PSP

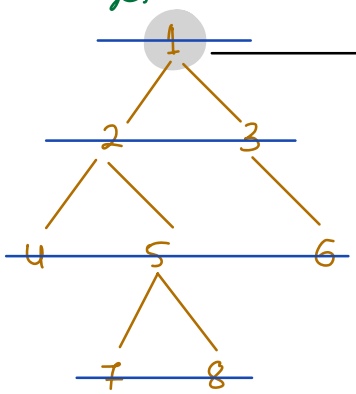
↓

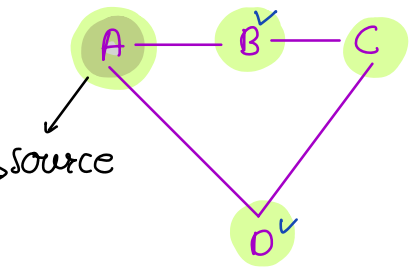
62 ⟶ 70%

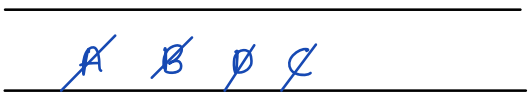
BFS { Breadth first search } Level order traversal

All trees are graph All graphs are not trees

A tree will always have n-1 edges







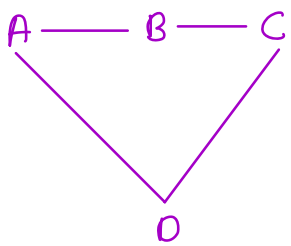
A B D C

1 2 3 4 5 6 7 8

Steps

1> source node added to queue

2> Remove front node from queue
Add all the unvisited neighbors
while adding the unvisited neighbors mark them as visited



visited T T T T
A B C D

A B D C

Pseudocode

nodes {0 - (N-1)}

Given graph {Adjacency list} source N

```
queue      // initialise
visited [N]    // false initally
queue.add (source)
visited [source] = true

while (!queue.isEmpty()) {
        node = queue.remove()
        print(node)
        for ( nei : graph [node] ) {
                if ( ! visited [nei] ) {
                        visited [nei] = true
                        queue.add (nei)
                }
        }
}
```

Tc : O(V + E)

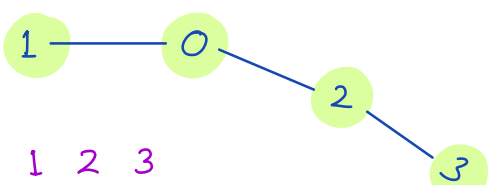
no. of node, no. of edges

Sc : O(V)

no of nodes

QUIZ:

	0	1	2	3
0	0	1	1	0
1	1	0	0	0
2	1	0	0	1
3	0	0	1	0



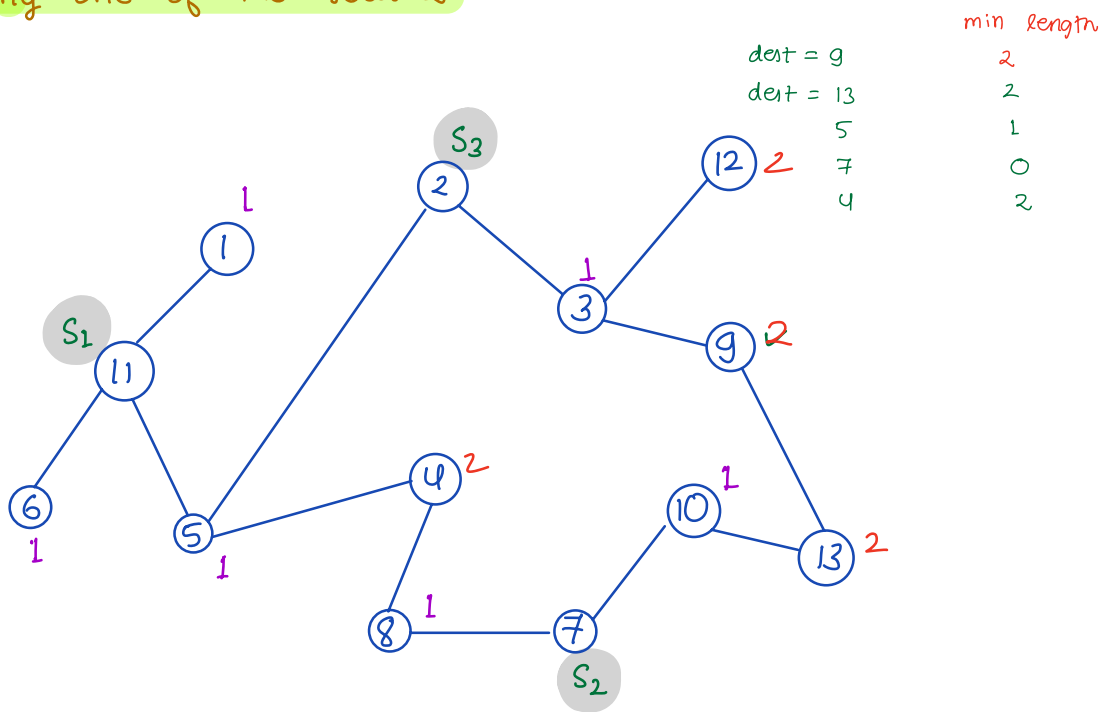
0 1 2 3

0 2 1 3

Multisource BFS *****

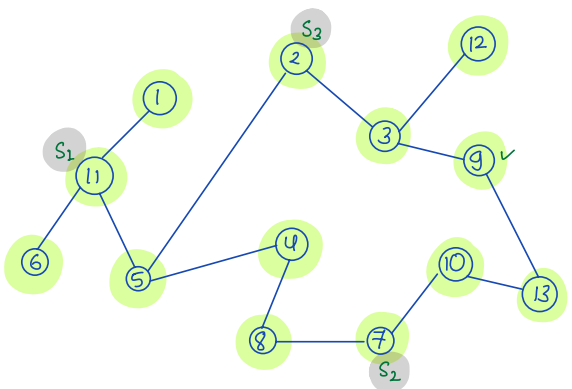
There are N no. of nodes and multisource $\{S_1, S_2, S_3\}$

Find the length of the shortest path for given destination to any one of the sources



* BFS will give you shortest path {see end of notes}

Idea ⟶ Add all sources to the queue with dist = 0



dest = 12

dist, node

Q (0,11) (0,2) (0,7)

(1,5), (1,6), (1,1)

(1,13), (1,8), (1,10)

(2,4) (2,12), (2,9)

(2,13)

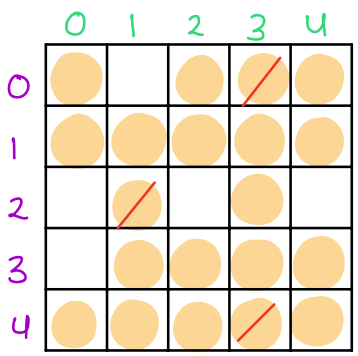
Rotten Oranges **** Amazon

Given a matrix [R][C] with 3 values

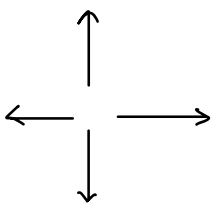
0 — Empty
1 — Fresh
2 — Rotten

Find the earliest time when all become rotten.

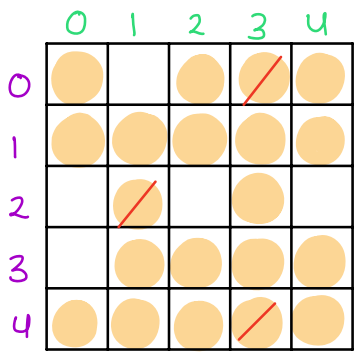
Note: Return -1 if its not possible.



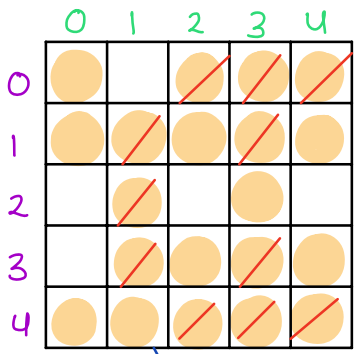
valid directions



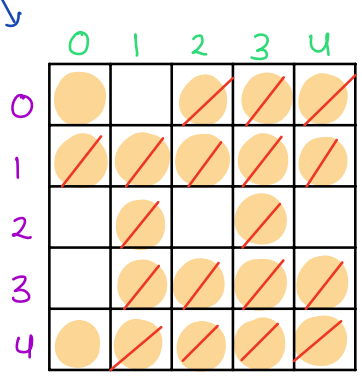
t = 0



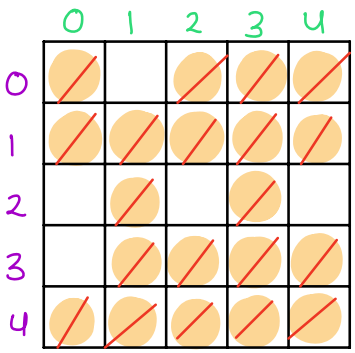
t = 1

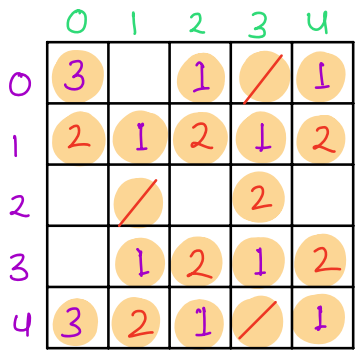


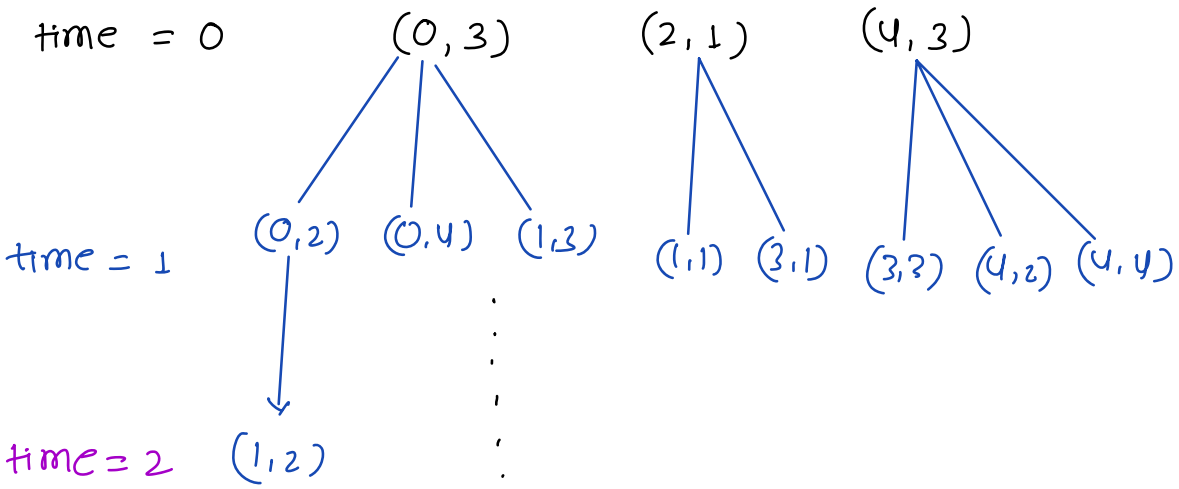
t = 2



t = 3







no. of rows → R

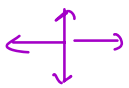
no. of cols → C

Pseudo code

```
q    // initialize
mintime = 0
for r ⟶ 0 to R-1 {
    for c ⟶ 0 to C-1 {
        if ( A[r][c] == ROTTEN )
            q.add ( [ 0 , r , c ] )
                      time  row  col
```

	UP	Left	Down	Right
DR =	-1	0	1	0
DC =	0	-1	0	1

```
while (!q.isEmpty()) {
    time, r, c = q.remove()
    mintime = time
    for i ⟶ 0 to 3 {   // iterate all direction
        nr = r + DR[i]
        nc = c + DC[i]
        // check if nr and nc are within matrix
        if( A[nr][nc] == FRESH) {
            A[nr][nc] = ROTTEN
            q.add((time+1, nr, nc))
        }
    }
}
```



```
// Check for fresh orange
for r ⟶ 0 to R-1 {
    for c ⟶ 0 to C-1 {
        if( A[r][c] == FRESH )
            return -1
    }
}
print(mintime)
```

TC: O(RC)

SC: O(RC)

Break : 22 : 40

Read

↓

Flipkart Grocery has several warehouses spread across the country and in order to minimize the delivery cost, whenever an order is placed we try to deliver the order from the nearest warehouse.

Therefore, each Warehouse is responsible for a certain number of localities which are closest to it for deliveries, this **minimizes the overall cost for deliveries**, effectively managing the distribution workload and minimizing the overall delivery expenses.

Problem statement:-

You are given a 2D matrix **A** of size **NxM** representing the map, where each cell is marked with either a **0 or a 1**. Here, a **0** denotes a locality, and a **1** signifies a warehouse. The objective is to calculate a new **2D matrix** of the same dimensions as **A**.

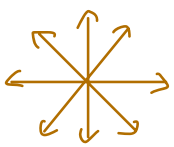
In this new matrix, the value of each cell will represent the minimum distance to the nearest warehouse. For the purpose of distance calculation, you are allowed to move to any of the **eight adjacent cells** directly surrounding a given cell.

Input

1	0	0
0	0	0
0	0	0

Output

0	1	2
1	1	2
2	2	2



Input

1	0	0
0	0	0
0	0	1

Output

0	1	2
1	1	1
2	1	0

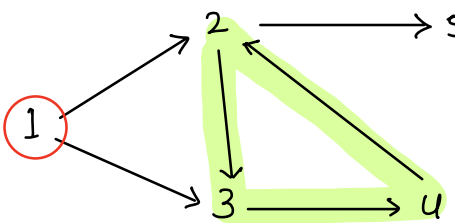
Possibility of finishing the courses

Given N courses with pre-requisites.
Check if its possible to finish all the course

course	pre-requisite for
1	2, 3
2	3 5
3	4
4	2

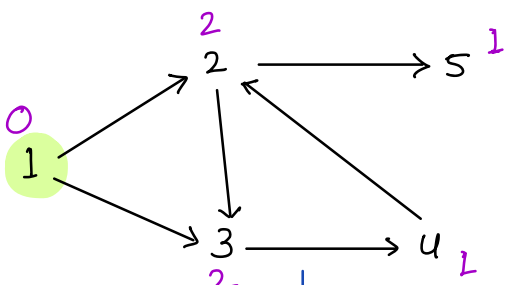
{ 1 is pre-req for 2 & 3 }

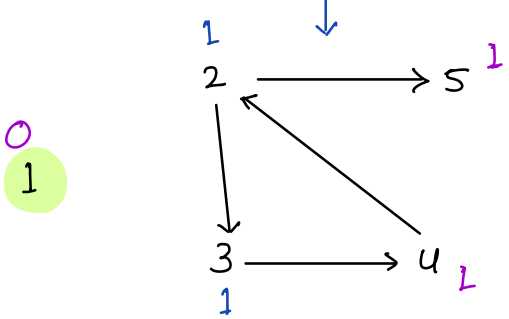
NOTE : In case of a cycle courses cannot be completed.
not possible



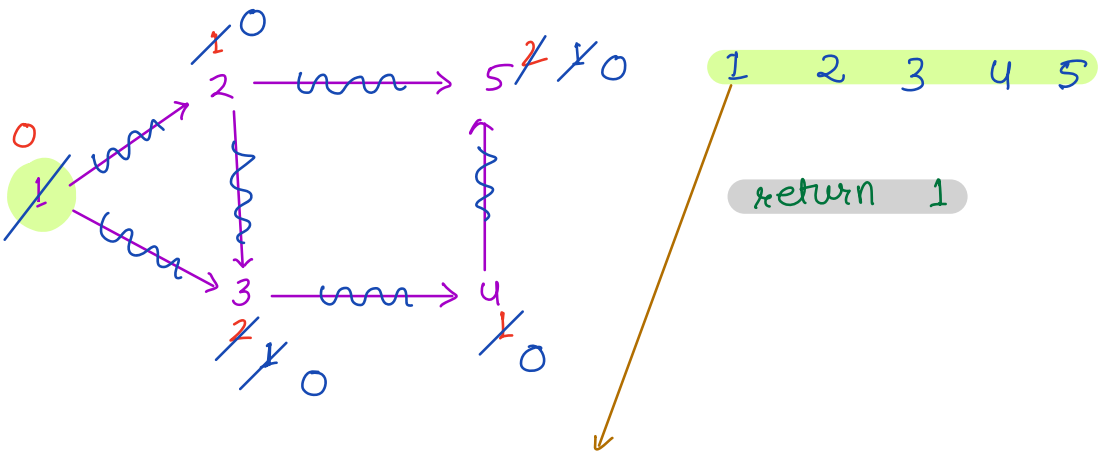
deadlock

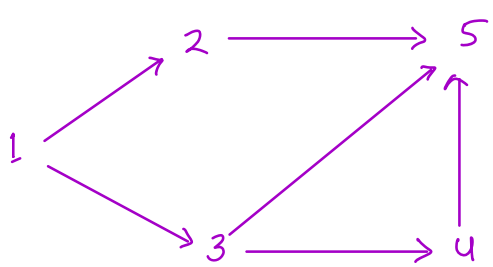
job → experience





unable to proceed
return 0





edges are uni-directional

Is the order 1 2 3 4 5 valid ? yes

1 3 2 4 5 valid ? yes

1 3 4 2 5 valid ? yes

Multiple ways to complete the course

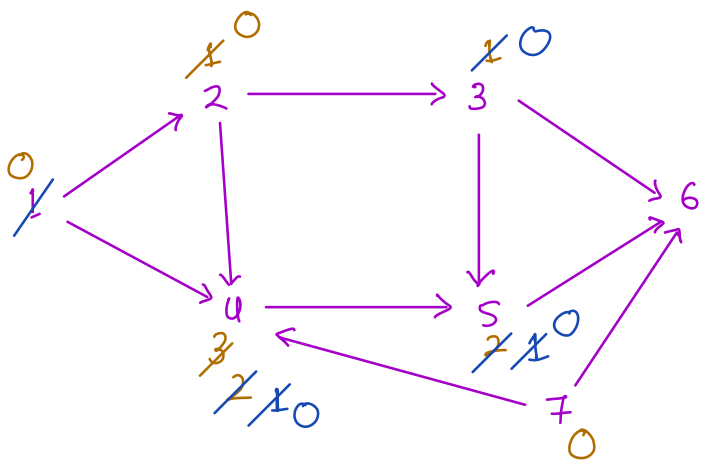
Topological Sort

work only on directed acyclic graph.

If there is an edge $u \longrightarrow v$

u always comes before v in topological ordering.

Find the topological sort of below graph { Kahn's algo }



1 7 2 3 4 5 6

Step 1 ⟶ calculate indegree for all nodes

for u, v in edges u ⟶ v

indegree [v] ++

⟶ Put all the nodes with indegree == 0 inside a queue

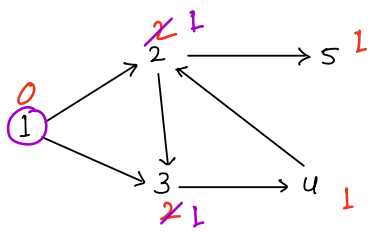
⟶ Remove q front { node }

goto all neighbors of node and −1 indegree

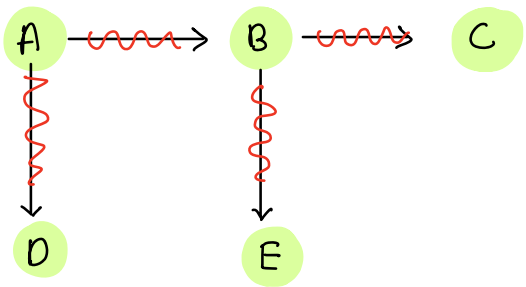
if indegree of nei == 0

add them to q

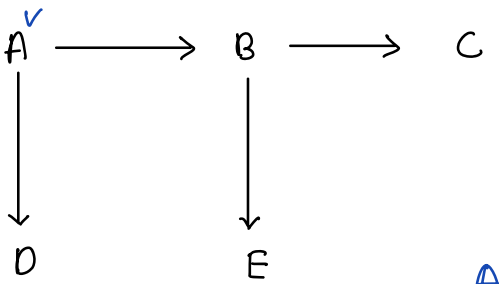
If topological sort via the above method doesn't contain N nodes ⟹ there is a cycle



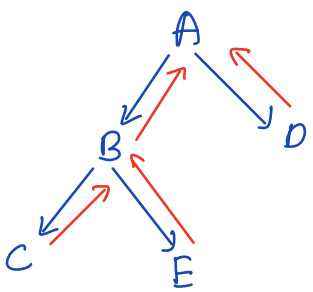
Topological Sort Another Approach {Right to Left}



C E B D A



C E B D A



Just do a post order on graph

NOTE : Topo sort is only applicable on DAG

Check for cycle first.

Pseudocode

```
// given input N, M
```

construct adjacency list.

```
list < list <int> > graph  // Adjacency list
visited []
```

V times

```
for node ⟶ 1 to N {
    if (!visited [node]) dfs (node)
}
```

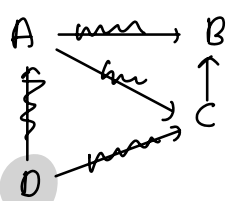
```
void dfs ( node ) {
    // mark the node as visited
    visited [node] = true

    // traverse to all unvisited nei of node
    for ( nei : graph.get(node) ) {
        if (! visited [nei]) {
            dfs(nei)
        }
    }

    print (node)   { reverse topo sort }
}
```

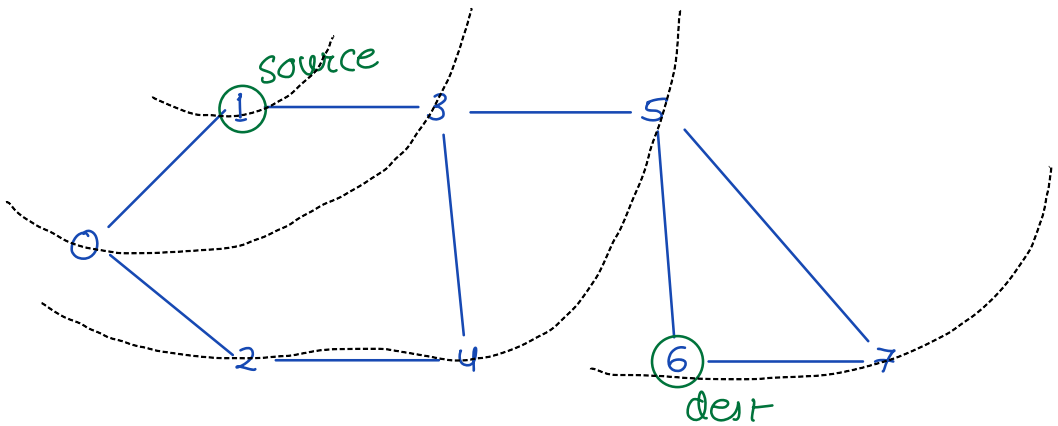
TC: O(V+E)

SC: O(V+E)

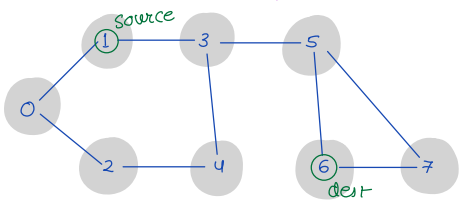


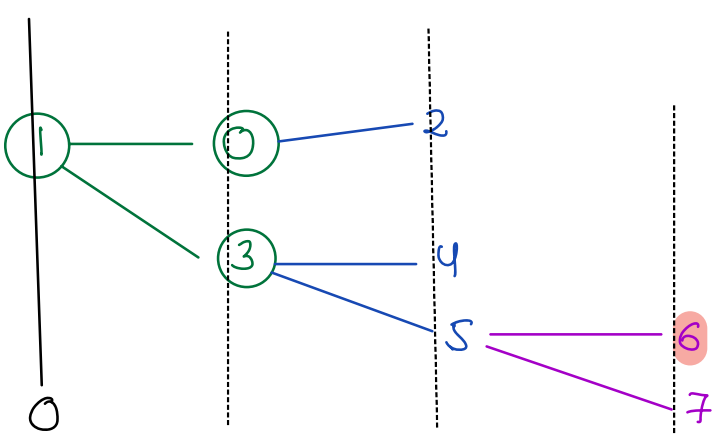
D A C B

Another approach to BFS



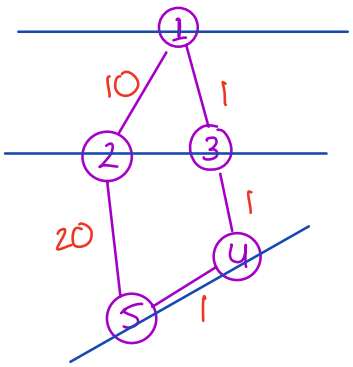
Find the min no. of edges to reach v starting from u in simple undirected graph





∵ BFS work level by level the first time we see a node is its minimum distance from source.

⟶ It will always work only if the edge weights are SAME



1 —— 5 shortest dist = 3

but we got 30

For BFS to return shortest distance edge weights should be same