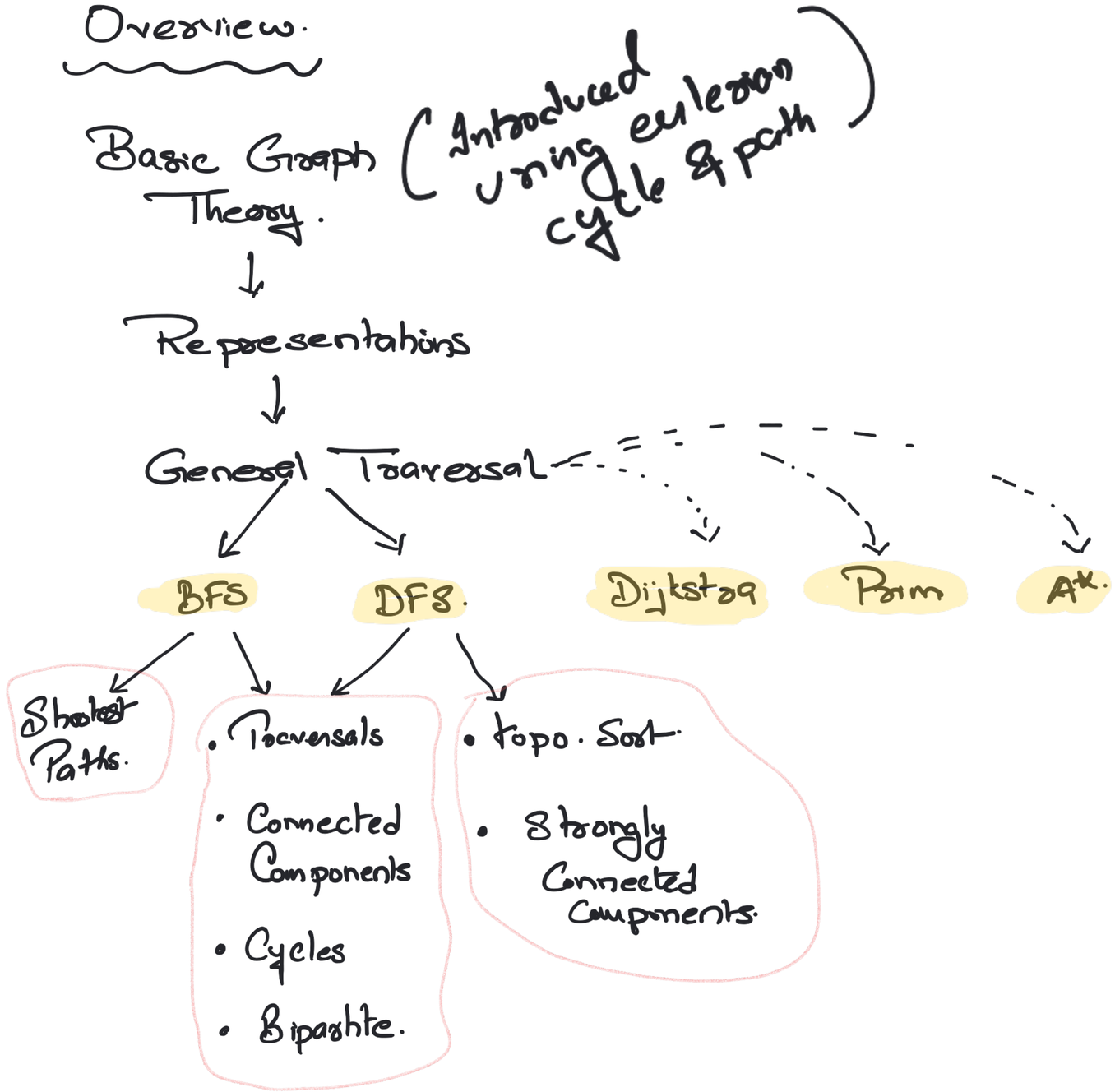
Date: 2020.09.10

Graphs.

Pavan - AMZN
Devices.

NSPAVAN @ GMAIL.COM.

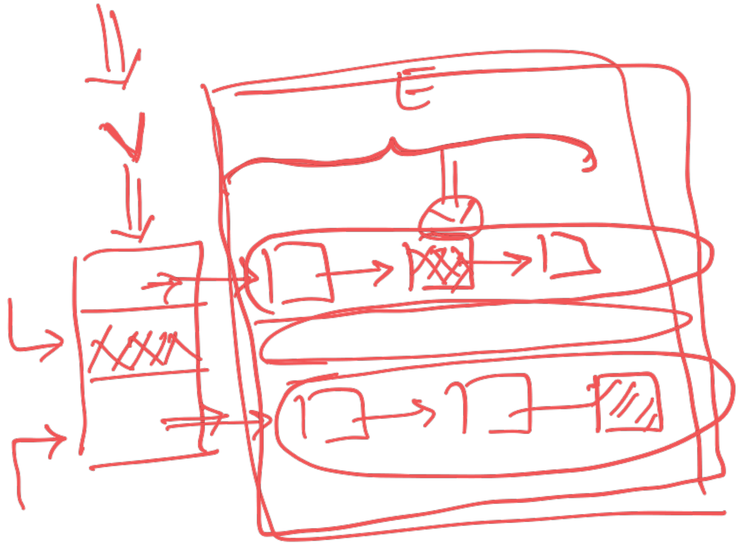

Overview.
Basic Graph Theory.
(Introduced using euleian cycle & path)
Representations
General Traversal
BFS
DFS
Dijkstra
Prim
A*.
Shortest Paths.
• Traversals
• Connected Components
• Cycles
• Bipartite.
• topo. Sort
• Strongly Connected Components.

Terminology.

- Vertex
- Edge
- Path
- Cycle
- connected graph.
- weights of edges.
- Degree of a vertex.

Representations:

- edge list., vertices.
- Adj list, Map
- Adj Matrix



for every vertex: V :
{ for every adj of V : A :
(V → A)

$V(1 + e)$

$(V + Ve) \rightarrow E$

$(V + V \sum_{i=0}^{n} e_i)$

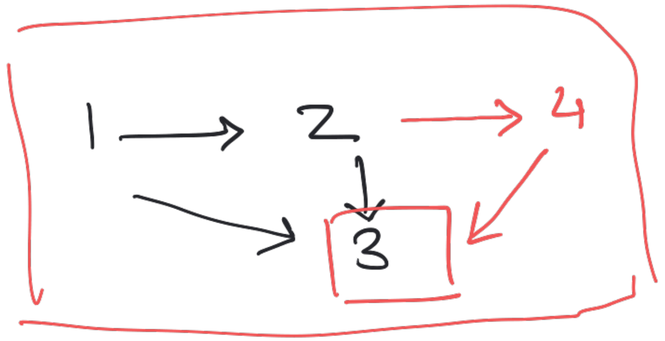
Universal Sink

[x → y]

[1, 2]

[1, 3]

[2, 3]

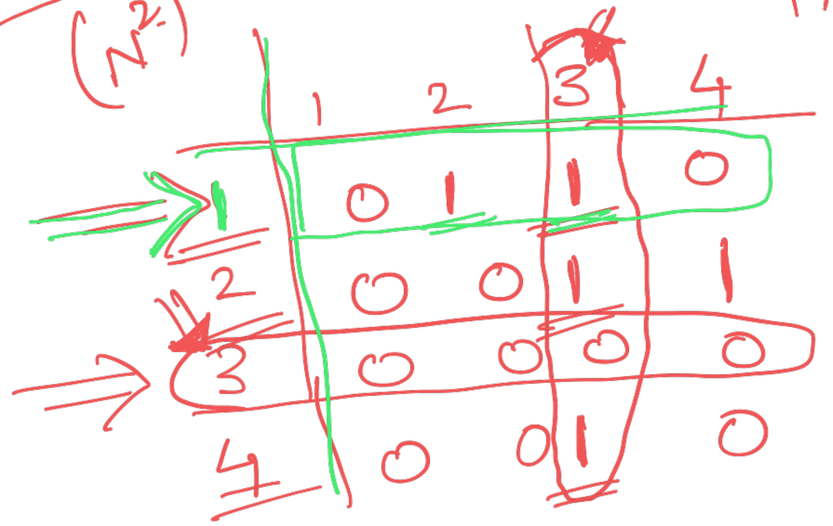


Indegree (N-1)

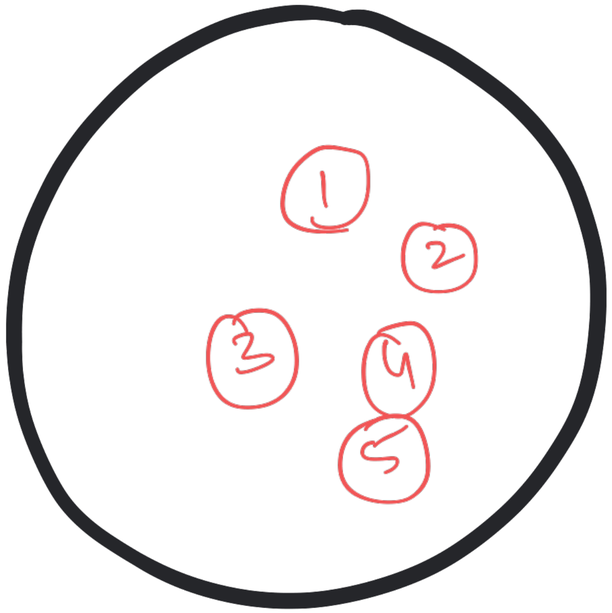
Outdegree (0).

Whats the indegree of the celebrity where there a "n" ppl?.

(N^2)

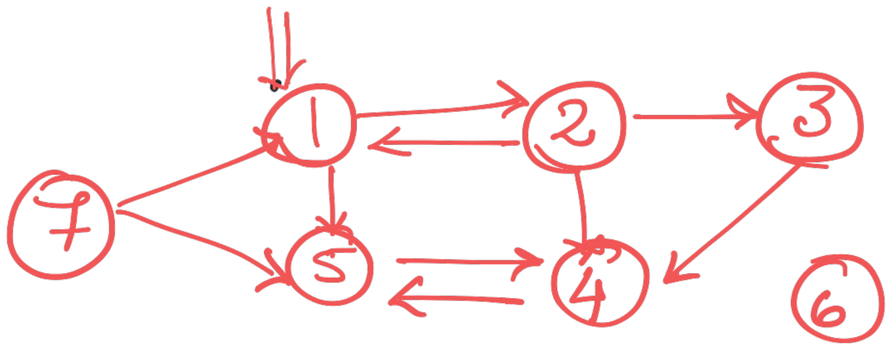


General Traversal.

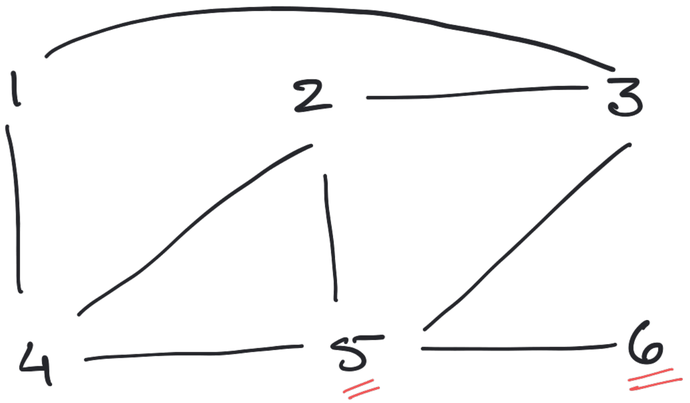


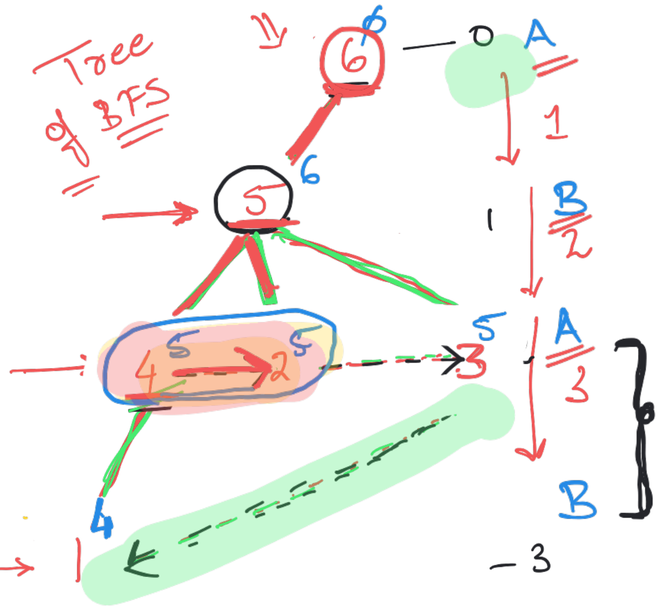
Captured

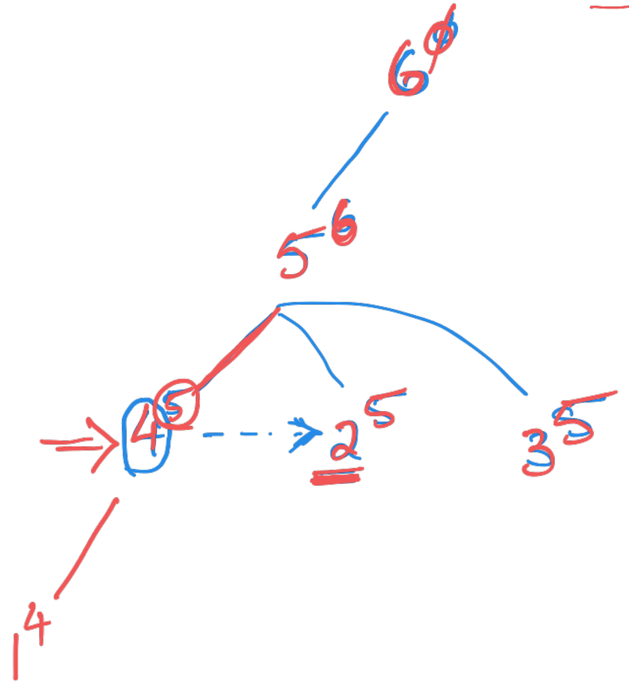
Yet to Capture.

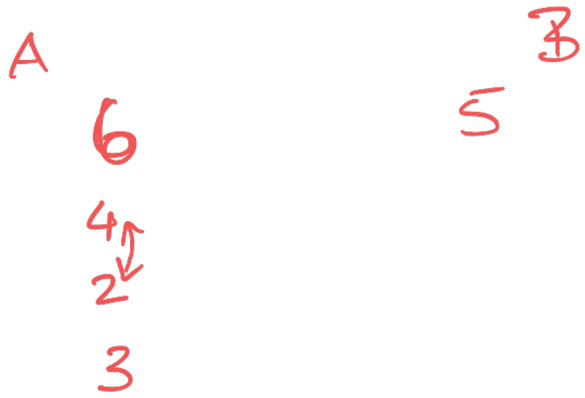


Breadth First Traversal.

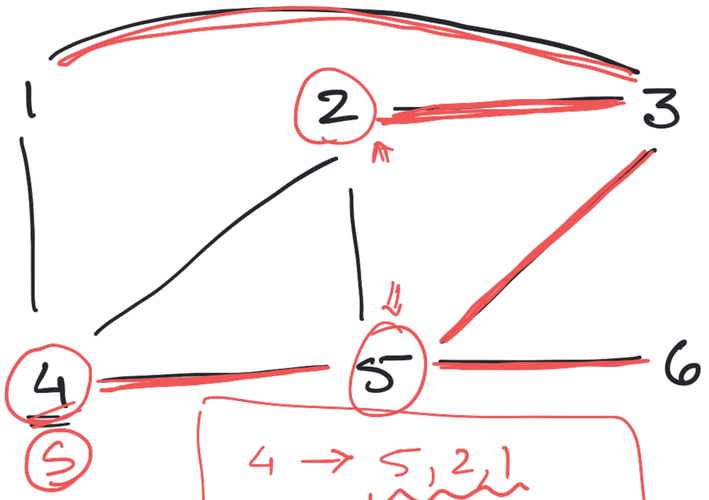








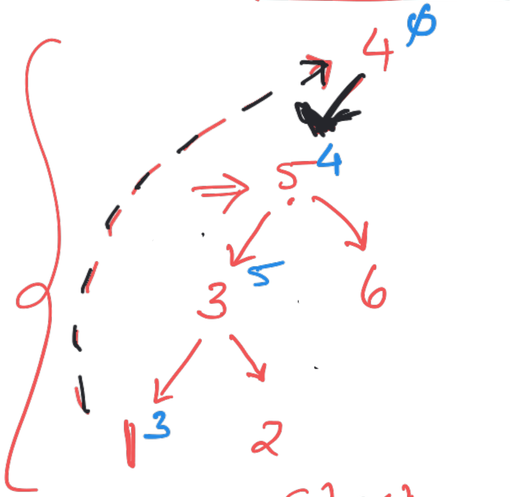
Depth first traversal



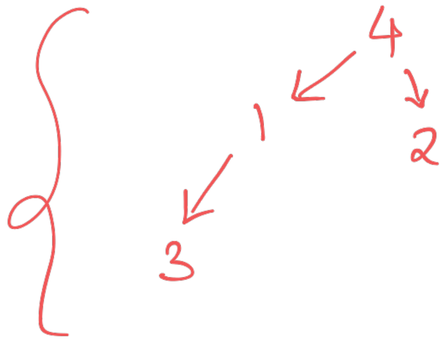
4 → 5, 2, 1
5 → 4, 3, 6
1 → 4, 3.
2 → 4, 3, 5
6 → ∅
3 → 1, 2, 5

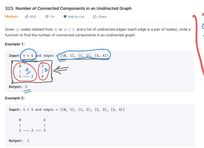
DFS tree

Recursion



Stack






```
countComponents(n, edges):
    adjList = createAdj(n, edges)
    return bfsCount(adjList)
```

```
bfsCount(adjList):
    visited = [], count = 0
    for (v = 0; v < n; v++):
        if (!visited[v]):
            bfs(v, adjList)
            count += 1
    return count
```

```
createAdj(n, edges):
    adjList = [] * n;
    for each edge in edges:
        adjList[edge.first].push(edge.second)
        adjList[edge.second].push(edge.first)
    return adjList.
```

driver dfs on a graph

```
dfsCounter(adjList):
    visited = [], count = 0.
    for (v = 0; v < n; v++):
        if (!visited[v]):
            dfs(v)
            cont += 1
    return Count.
```

from a node

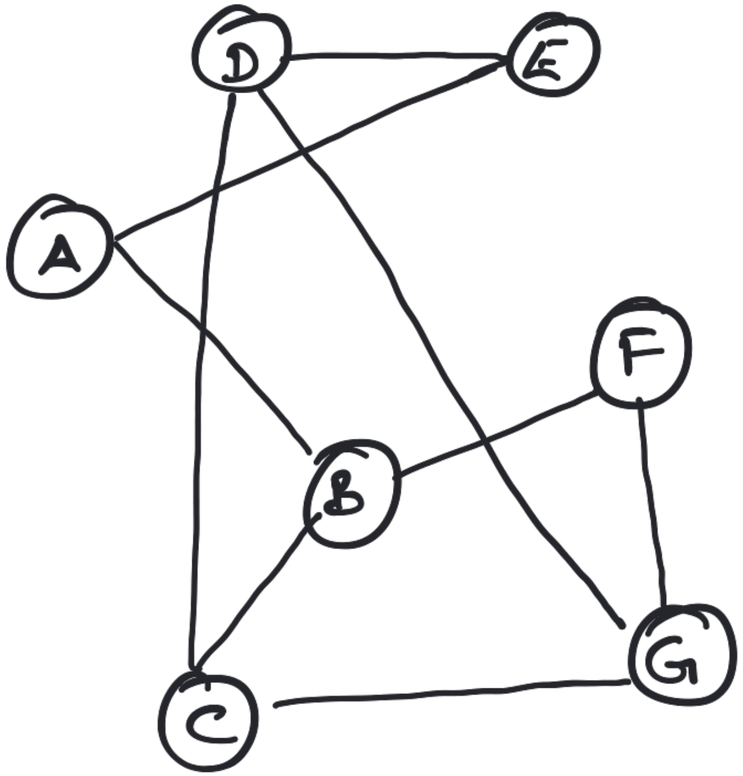
```
bfs(src, adj, visited):
    q.push(src);
    visited[src] = true;
    while (q.size() != 0):
        curr = q.pop();
        for neighbor in adj[curr]:
            if (!visited[neighbor]):
                q.push(neighbor);
                visited[neighbor] = true;
```

```
dfs(src, adj, visited):
    visited[src] = true;
    for neighbor in adj[src]:
        if (!visited[neighbor]):
            dfs(neighbor)
```

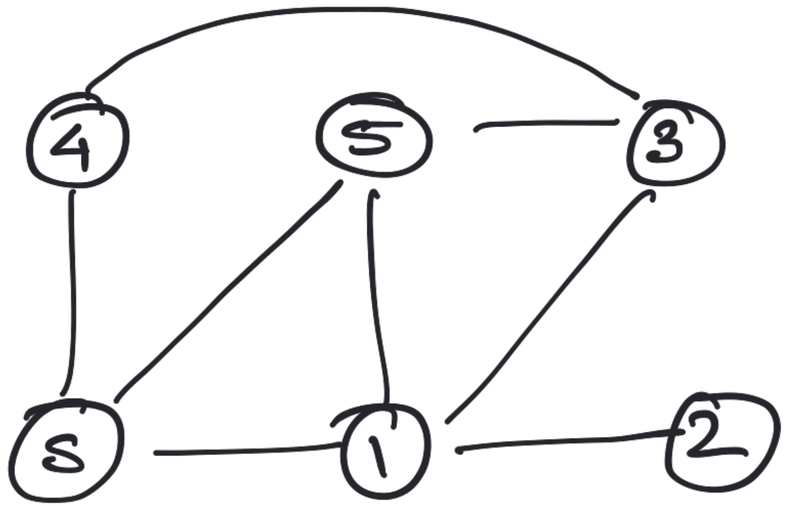
```
bfs(src, adj, visited):
    q.push(src);
    visited[src] = true;
    while (q.size() != 0):
        curr = q.pop();
        for neighbor in adj[curr]:
            if (!visited[neighbor]):
                q.push(neighbor);
                visited[neighbor] = true;
```

```
dfs(src, adj, visited):
    visited[src] = true;
    for neighbor in adj[src]:
        if (!visited[neighbor]):
            dfs(neighbor)
```

Lets explore Properties of BFS. on undirected graph.



Let explore Properties of DFS. on an undirected graph.



New Terms:

Tree Edge:

Back edges:

Forward edges ← directed.

Cross edges:

Questions:

1) Are there back-edges in BFS of an undirected graph?

261. Graph Valid Tree

Medium 913 31 Add to List Share

Given `n` nodes labeled from `0` to `n-1` and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges make up a valid tree.

Example 1:

```
Input: n = 5, and edges = [[0,1], [0,2], [0,3], [1,4]]
Output: true
```

Example 2:

```
Input: n = 5, and edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]
Output: false
```

Note: you can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0,1]` is the same as `[1,0]` and thus will not appear together in `edges`.

```
bool
↳ dfs(src, adj, visited):
    visited[src] = true;
    for neighbor in adj[src]:
        if (!visited[neighbor]):
            if (dfs(neighbor)):        parent[neighbor]=src
                return true;
        .
        .
        .
        else:
            if parent[src] != neighbor:
                # cycle
                return true;
    return false;
```

```
bool
↳ dfs(src, adj, visited):
    visited[src] = true;
    for neighbor in adj[src]:
        if (!visited[neighbor]):
            if (dfs(neighbor)):        parent[neighbor]=src
                return true;
        .
        .
        .
        else:
            if parent[src] != neighbor:
                # cycle
                return true;
    return false;
```

```
bfs(src, adj, visited):
    q.push(src);
    visited[src] = true;
    while (q.size() != 0):
        curr = q.pop();
        for neighbor in adj[curr]:
            if (!visited[neighbor]):
                q.push(neighbor);
                visited[neighbor]=true;
                parent[neighbor] = curr
    else:
        if parent[curr] != neighbor:
            # cycle.
            return true;
    return false.
```

```
bfs(src, adj, visited):
    q.push(src);          dist[src]= 0
    visited[src] = true;
    while (q.size() != 0):
        curr = q.pop();
        for neighbor in adj[curr]:
            if (!visited[neighbor]):
                q.push(neighbor);
                visited[neighbor]=true;
                parent[neighbor] = curr
                distance[neighbor]= distance[curr]+1
    else:
        if parent[curr] != neighbor:
            if distance[curr] == distance[neighbor]
                return false
    return ~~false~~true.
```

New Team: Bipartite.

785. Is Graph Bipartite?

Medium 1151 140 Add to List Share

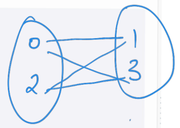
Given an undirected `graph`, return `true` if and only if it is bipartite.

Recall that a graph is *bipartite* if we can split it's set of nodes into two independent subsets A and B such that every edge in the graph has one node in A and another node in B.

The graph is given in the following form: `graph[i]` is a list of indexes `j` for which the edge between nodes `i` and `j` exists. Each node is an integer between `0` and `graph.length - 1`. There are no self edges or parallel edges: `graph[i]` does not contain `i`, and it doesn't contain any element twice.

Example 1:
Input: `[[1,3], [0,2], [1,3], [0,2]]`
Output: true
Explanation:
The graph looks like this:
```
0----1
|    |
|    |
3----2
```
We can divide the vertices into two groups: {0, 2} and {1, 3}.

N = 4
0 → 1,3
1 → 0,2
2 → 1,3
3 → 0,2



Example 2:
Input: `[[1,2,3], [0,2], [0,1,3], [0,2]]`
Output: false
Explanation:
The graph looks like this:
```
0----1
| \  |
|  \ |
3----2
```
We cannot find a way to divide the set of nodes into two independent subsets.

Note:

- `graph` will have length in range `[1, 100]`.
- `graph[i]` will contain integers in range `[0, graph.length - 1]`.
- `graph[i]` will not contain `i` or duplicate values.
- The graph is undirected: if any element `j` is in `graph[i]`, then `i` will be in `graph[j]`.

A, B

```
bool
↳ dfs(src, adj, visited):
    visited[src] = true;
    for neighbor in adj[src]:
        if (!visited[neighbor]):
            if (dfs(neighbor)):
                return false.
        else:
            if color[src] == color[neighbor]:
                return false
    return true;
```

if parent[src] != undefind
    color[src] = flip(color[parent[src]])
else color[src] = "A";

parent[neighbor] = src

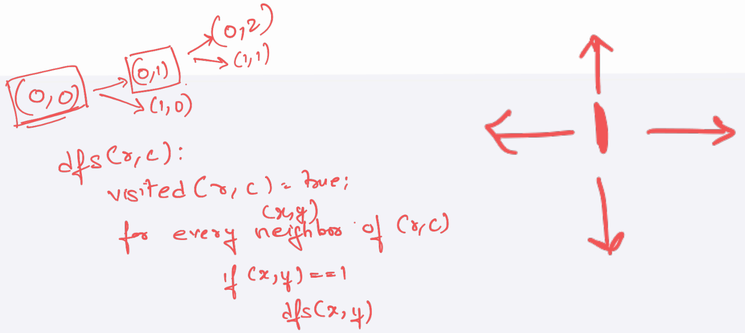
```
bfs(src, adj, visited):
    q.push(src);
    visited[src] = true;
    while (q.size() != 0):
        curr = q.pop();
        for neighbor in adj[curr]:
            if (!visited[neighbor]):
                q.push(neighbor);
                visited[neighbor] = true;
                parent[neighbor] = curr
            else:
                if parent[curr] != neighbor:
                    # cycle.
                    return true;
    return false.
```

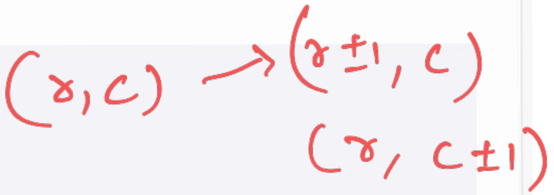
200. Number of Islands

Medium 4795 180 Add to List Share

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:






```
Input:
0 11110
1 11010
2 11000
3 00000

Output: 1
```

Example 2:

```
Input:
11000
11000
00100
00011

Output: 3
```

Description | Solution | Submissions | Discuss (586)

695. Max Area of Island

Medium | 1628 | 74 | Add to List | Share

Given a non-empty 2D array `grid` of 0's and 1's, an **island** is a group of `1`'s (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

Find the maximum area of an island in the given 2D array. (If there is no island, the maximum area is 0.)

Example 1:

```
[[0,0,1,0,0,0,0,1,0,0,0,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,1,1,0,1,0,0,0,0,0,0,0,0],
 [0,1,0,0,1,1,0,0,1,0,1,0,0],
 [0,1,0,0,1,1,0,0,1,1,1,0,0],
 [0,0,0,0,0,0,0,0,0,0,1,0,0],
 [0,0,0,0,0,0,0,1,1,1,0,0,0],
 [0,0,0,0,0,0,0,1,1,0,0,0,0]]
```

Given the above grid, return `6`. Note the answer is not 11, because the island must be connected 4-directionally.

Example 2:

```
[[0,0,0,0,0,0,0,0]]
```

Given the above grid, return `0`.

Note: The length of each dimension in the given `grid` does not exceed 50.

733. Flood Fill

Easy 847 159 Add to List Share

An `image` is represented by a 2-D array of integers, each integer representing the pixel value of the image (from 0 to 65535).

Given a coordinate `(sr, sc)` representing the starting pixel (row and column) of the flood fill, and a pixel value `newColor`, "flood fill" the image.

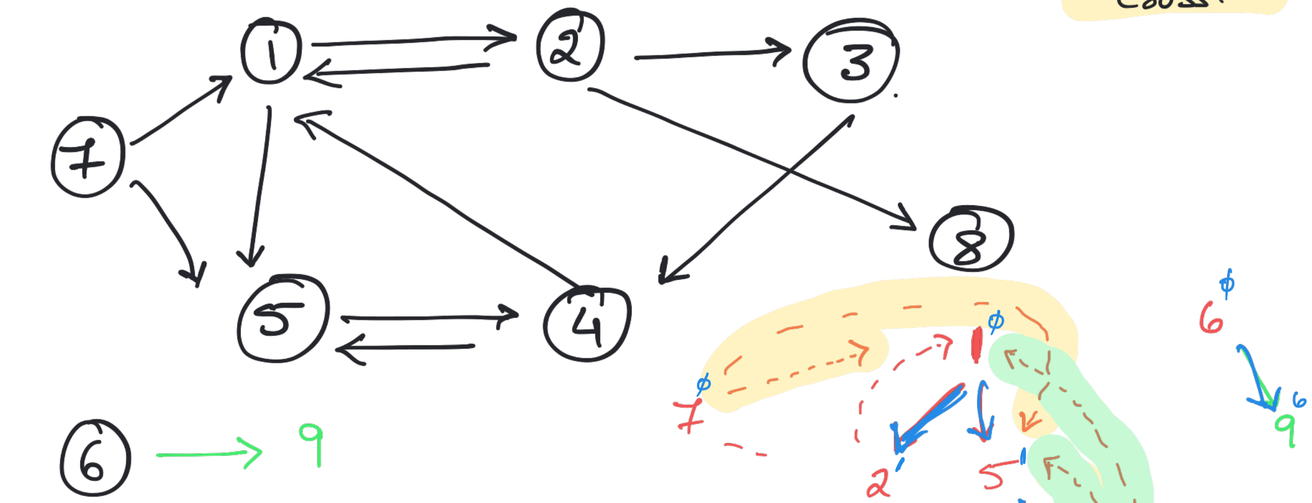
To perform a "flood fill", consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color as the starting pixel), and so on. Replace the color of all of the aforementioned pixels with the newColor.

At the end, return the modified image.

Example 1:

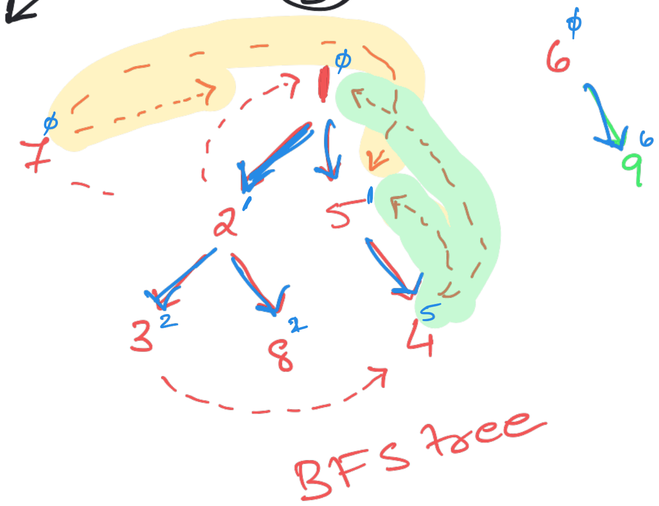
```
Input:
image = [[1,1,1],[1,1,0],[1,0,1]]
sr = 1, sc = 1, newColor = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation:
From the center of the image (with position (sr, sc) = (1, 1)), all pixels connected
by a path of the same color as the starting pixel are colored with the new color.
Note the bottom corner is not colored 2, because it is not 4-directionally connected
to the starting pixel.
```

Directed Graphs.



Back

Cross.



More Questions.

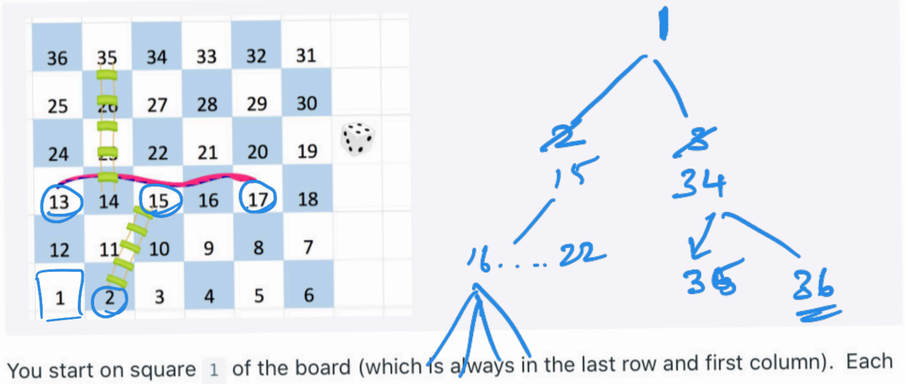
What does it take to find a cycle in such a graph ?

When would you use BFS in directed graphs ?

909. Snakes and Ladders

Medium 277 698 Add to List Share

On an N x N `board`, the numbers from `1` to `N*N` are written *boustrophedonically* **starting from the bottom left of the board**, and alternating direction each row. For example, for a 6 x 6 board, the numbers are written as follows:



You start on square `1` of the board (which is always in the last row and first column). Each move, starting from square `x`, consists of the following:

- You choose a destination square `S` with number `x+1`, `x+2`, `x+3`, `x+4`, `x+5`, or `x+6`, provided this number is `<= N*N`.
  - (This choice simulates the result of a standard 6-sided die roll: ie., there are always **at most 6 destinations, regardless of the size of the board**.)
- If `S` has a snake or ladder, you move to the destination of that snake or ladder. Otherwise, you move to `S`.

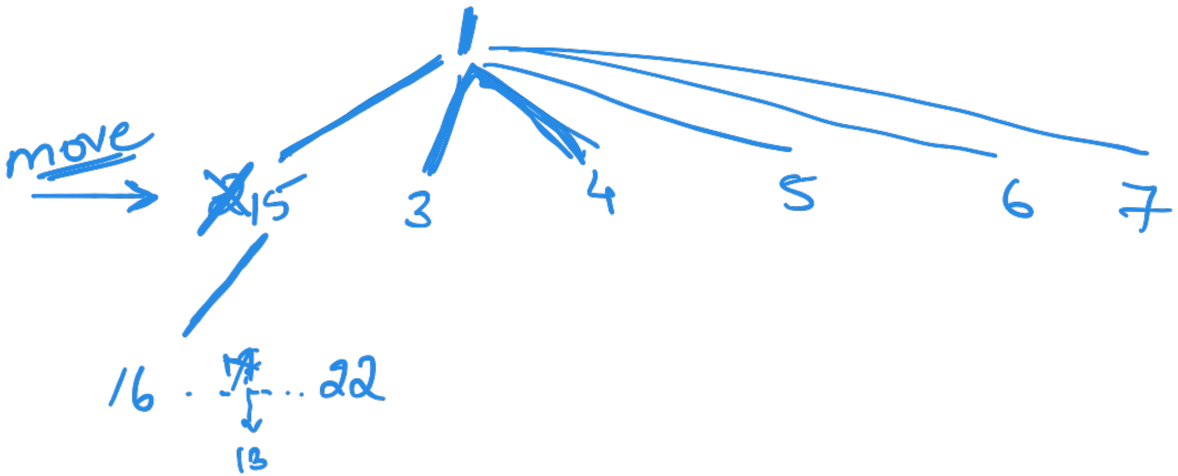
A board square on row `r` and column `c` has a "snake or ladder" if `board[r][c] != -1`. The destination of that snake or ladder is `board[r][c]`.

Note that you only take a snake or ladder at most once per move: if the destination to a snake or ladder is the start of another snake or ladder, you do **not** continue moving. (For example, if the board is `[[4,-1],[-1,3]]`, and on the first move your destination square is `2`, then you finish your first move at `3`, because you do **not** continue moving to `4`.)

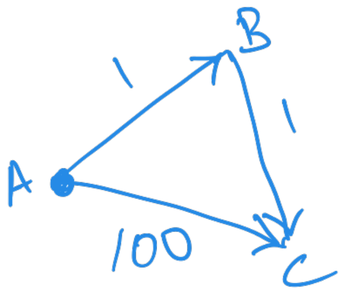
Return the least number of moves required to reach square `N*N`. If it is not possible, return `-1`.

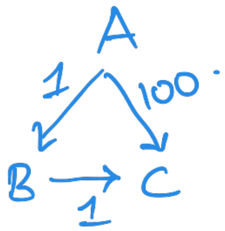
Example 1:

```
Input: [
[-1,-1,-1,-1,-1,-1],
[-1,-1,-1,-1,-1,-1],
[-1,-1,-1,-1,-1,-1],
[-1,35,-1,-1,13,-1],
[-1,-1,-1,-1,-1,-1],
[-1,15,-1,-1,-1,-1]]
Output: 4
Explanation:
At the beginning, you start at square 1 [at row 5, column 0].
You decide to move to square 2, and must take the ladder to square 15.
You then decide to move to square 17 (row 3, column 5), and must take
the snake to square 13.
You then decide to move to square 14, and must take the ladder to
square 35.
You then decide to move to square 36, ending the game.
It can be shown that you need at least 4 moves to reach the N*N-th
square, so the answer is 4.
```



What if there are weights on the edges.

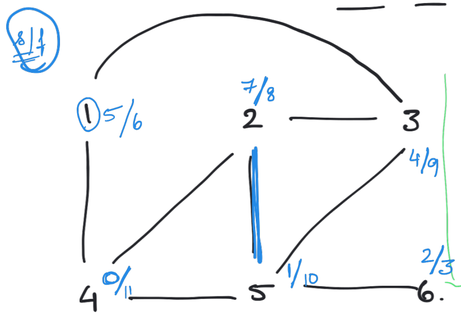




start
finished.

DFS
with additional Info
Book Keeping.

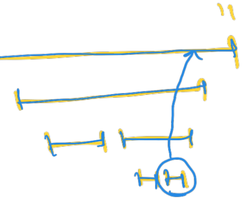
t = 0



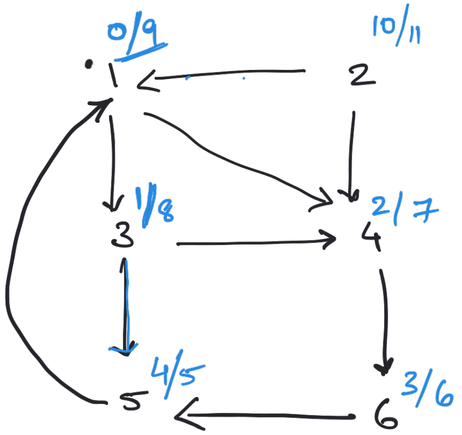

```
dfs (src, adj, visited):
    start [src] = t++
    visited [src] = true;
    for (neighbor) in adj [src]:
        if (!visited [neighbor]) :
            dfs (neighbor)
    finish [src] = t++
```

tree edge.

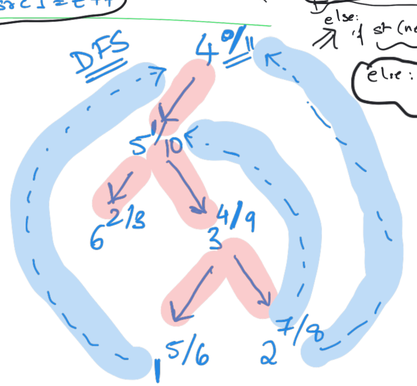
else:
- if no end time } back.
- else: if st (neighbor) > st (src) forward
- else: cross.

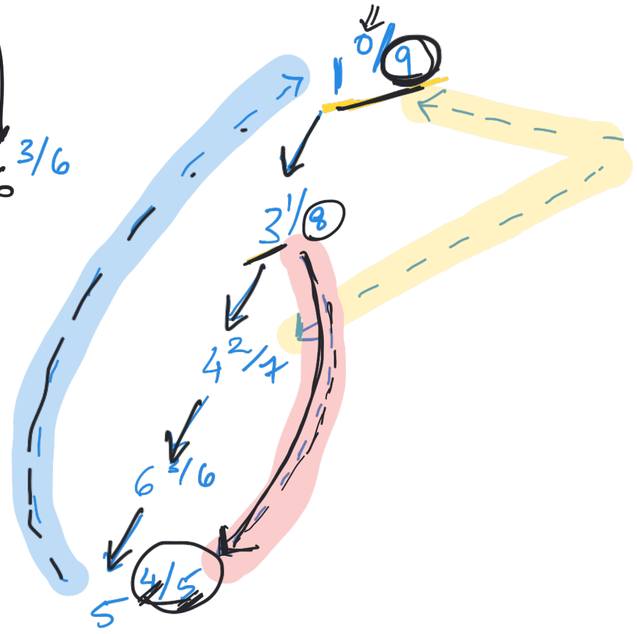


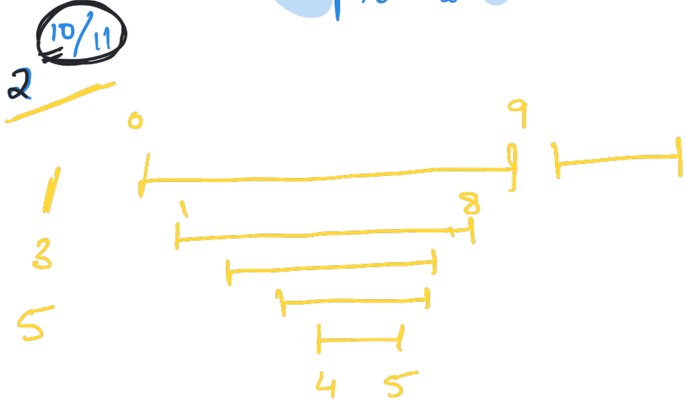
Undirected Graph.



Directed Graph.







207. Course Schedule

Medium 3178 162 Add to List Share

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, is it possible for you to finish all courses?

Example 1:

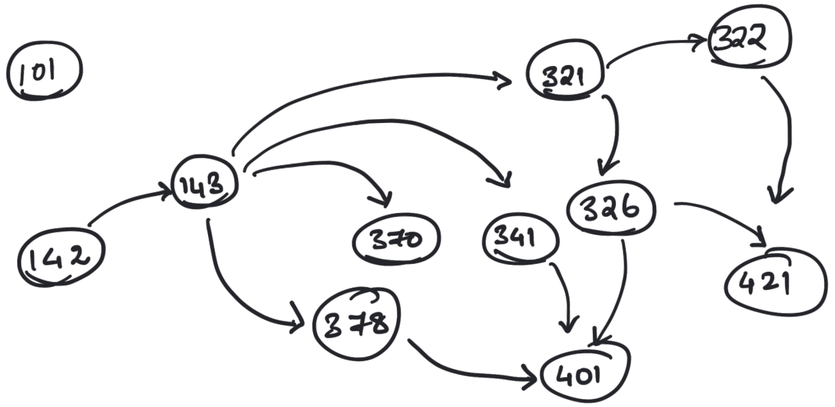
```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
             To take course 1 you should have finished course 0, and to take course 0 you should
             also have finished course 1. So it is impossible.
```

Constraints:

- The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
- You may assume that there are no duplicate edges in the input prerequisites.
- `1 <= numCourses <= 10^5`



210. Course Schedule II

Medium 1779 119 Add to List Share

There are a total of *n* courses you have to take, labeled from `0` to `n-1`.

Some courses may have prerequisites, for example to take course 0 you have to first take course 1, which is expressed as a pair: `[0,1]`

Given the total number of courses and a list of prerequisite **pairs**, return the ordering of courses you should take to finish all courses.

There may be multiple correct orders, you just need to return one of them. If it is impossible to finish all courses, return an empty array.

Example 1:

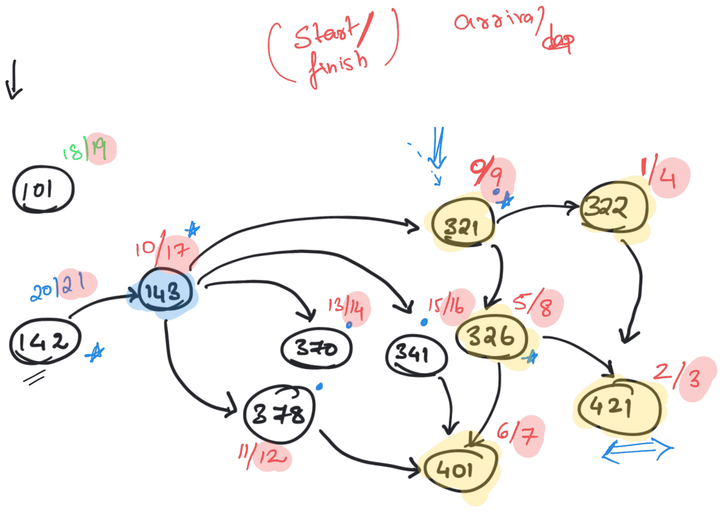
```
Input: 2, [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have finished
             course 0. So the correct course order is [0,1] .
```

Example 2:

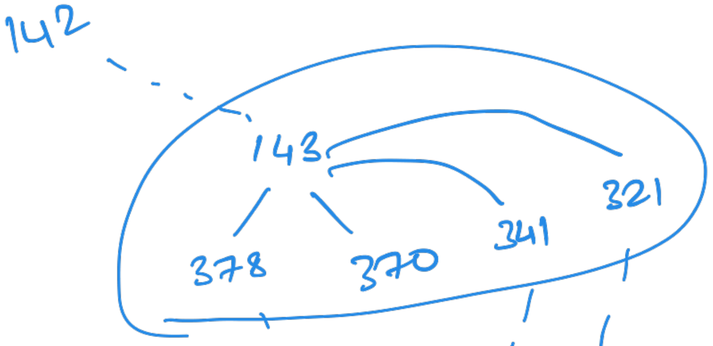
```
Input: 4, [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3] or [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have finished both
             courses 1 and 2. Both courses 1 and 2 should be taken after you finished course 0.
             So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3] .
```

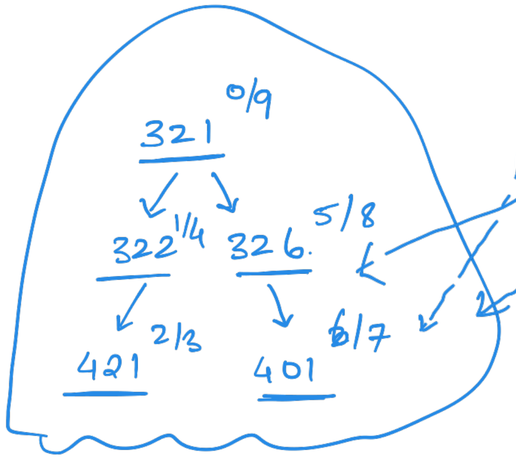
Note:

1. The input prerequisites is a graph represented by **a list of edges**, not adjacency matrices. Read more about how a graph is represented.
2. You may assume that there are no duplicate edges in the input prerequisites.



142 101 143 341 370 378 ↳321 326 401 322 421





1192. Critical Connections in a Network

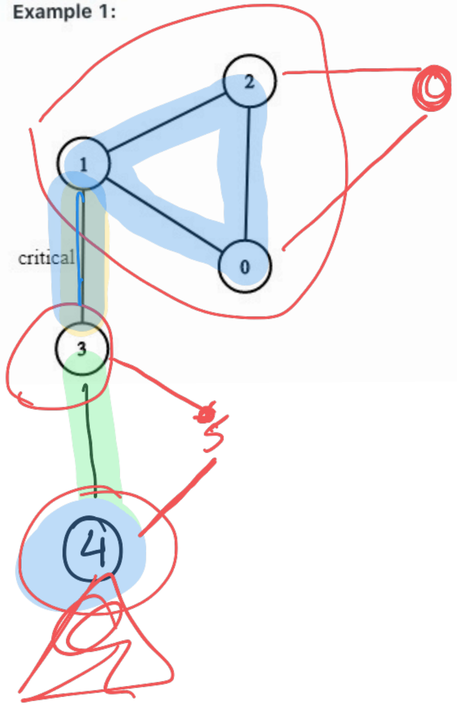
Hard 843 68 Add to List Share

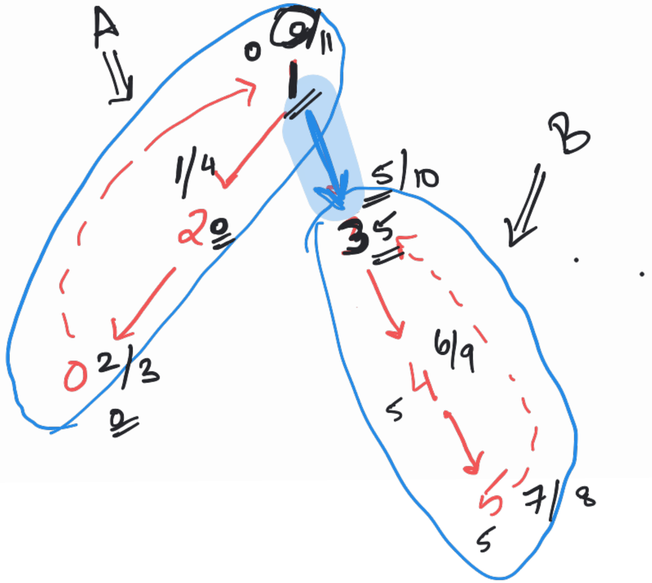
There are `n` servers numbered from `0` to `n-1` connected by undirected server-to-server `connections` forming a network where `connections[i] = [a, b]` represents a connection between servers `a` and `b`. Any server can reach any other server directly or indirectly through the network.

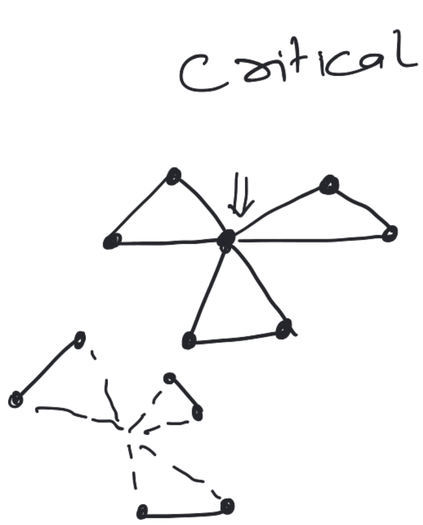
A *critical connection* is a connection that, if removed, will make some server unable to reach some other server.

Return all critical connections in the network in any order.

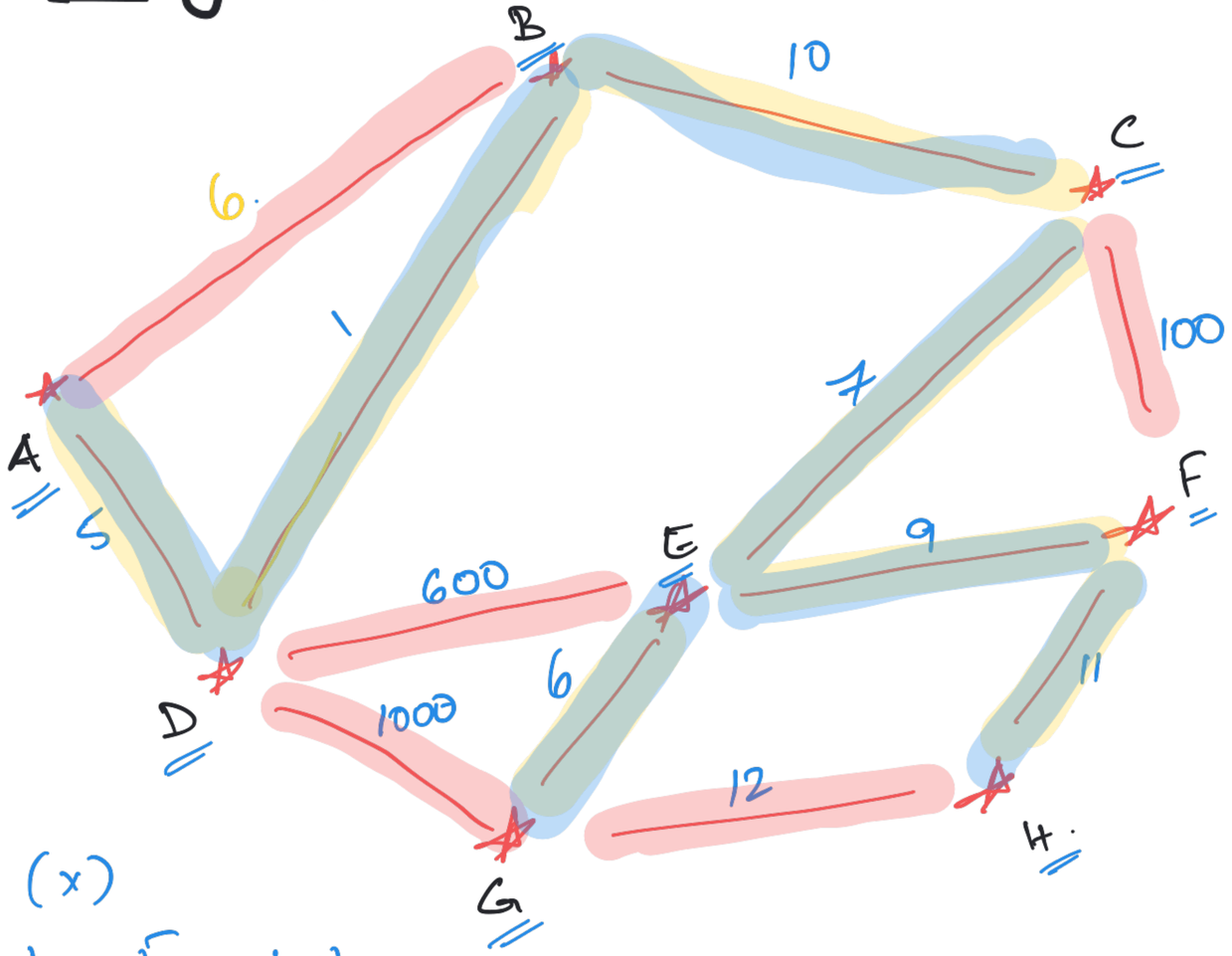
Example 1:







Spanning Tree.



for every edge (xy)

IsEdgeSafe (MST)

MST += xy

UNION (x, y)

FIND (x) != FIND (y)

(FIND (x)

!= FIND (y)

UNION (x, y))