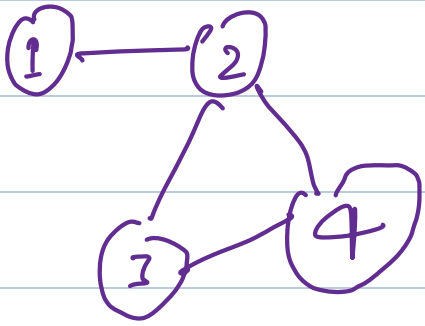
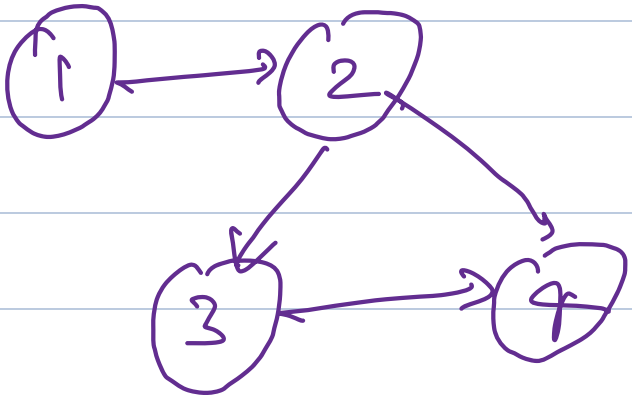
Graphs

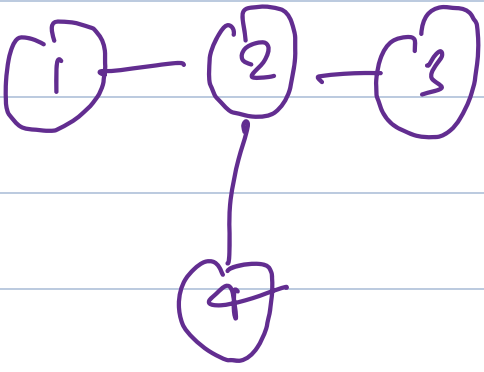
* Directed
* Undirected

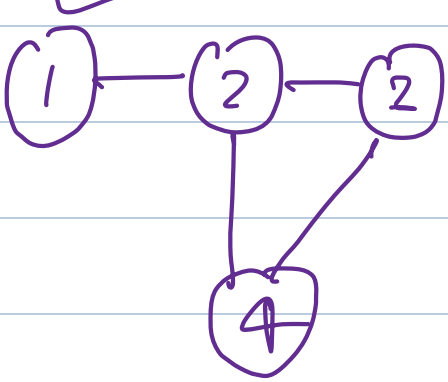




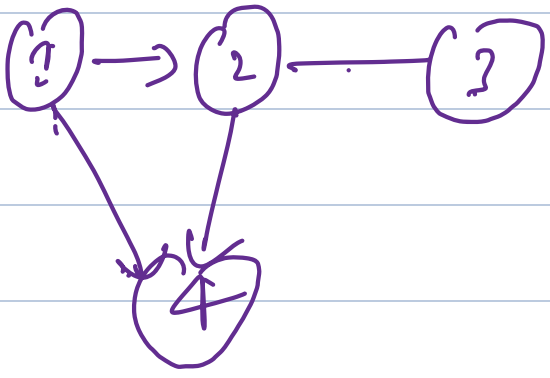
* Cycles in Graph
  - Start from a node
  - and exist a path that end on the same node

* Undirected cyclic graph





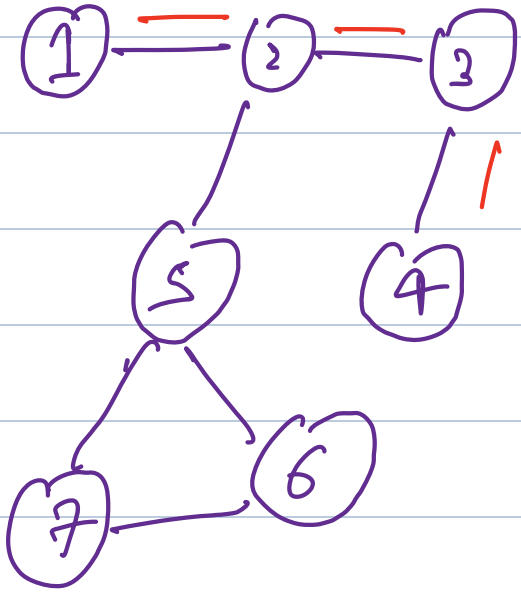
A Directed a cyclic graph



(DAG)

A Path

Contains list of vertices and edges



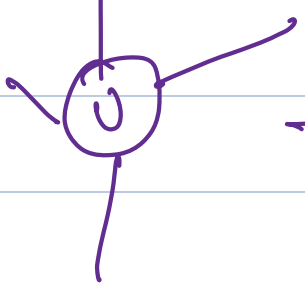
1 → 2 → 3

Path

unique set of vertices and edges

* Degree of a vertex

no of edges that goes or emerge



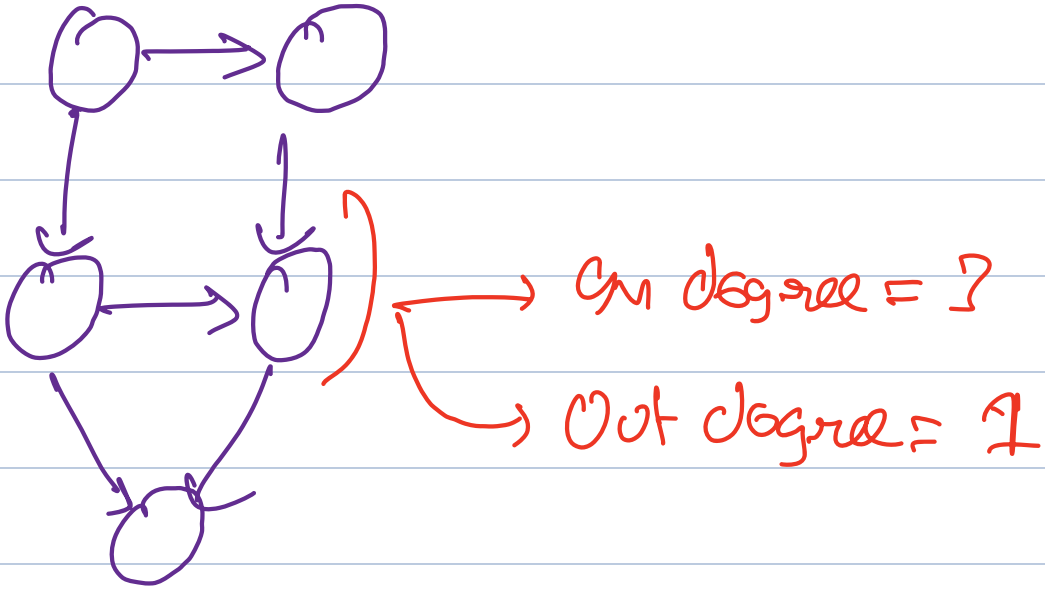
→ 4 degree

Total Degree of an undirected graph

= 2 * Edges

Directed graph degree

- In degree
- Out degree



* Edge weight

weight ≡ any numerical value

weights not provided → assume → 1 unit

* Graph Representation

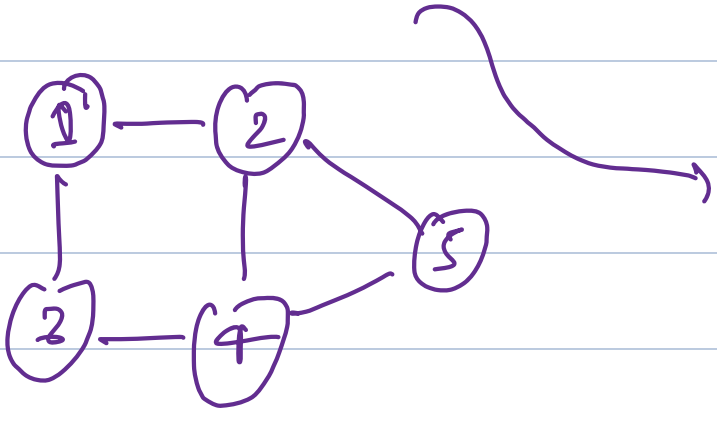
input ≡ n . m

m lines that represent edges

m0 node1 – node2
m1
m2
⋮
m(m-1) node$_j$ – node$_i$

*Example

n=5 m=6



5 4
5 2
2 4
2 1
1 3
1 2

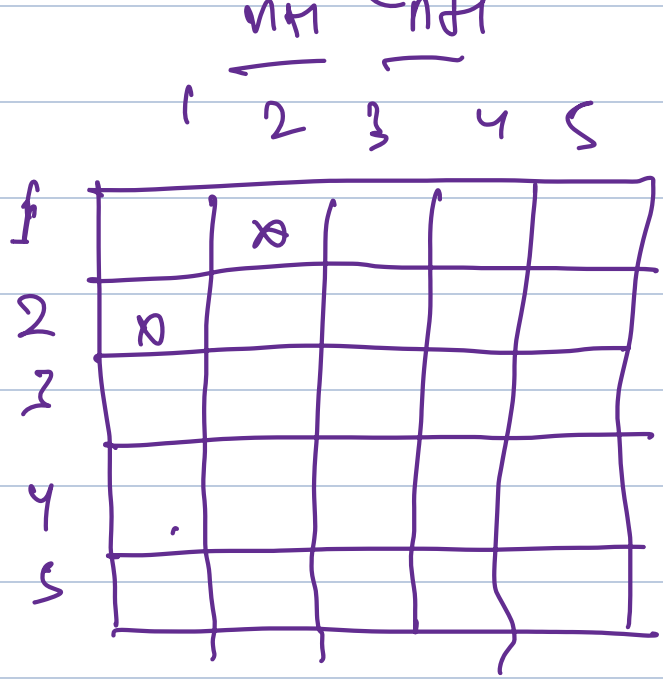
* How to store

① Matrix way

② List way

→ adjacency matrix

mat = adj[7][7] → no of nodes



for an undirected graph

adj[u][v] — There's an edge b/w

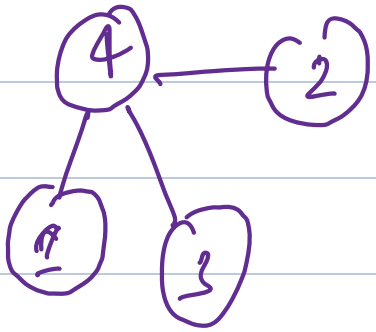
u → v and v → u

for an directed graph

adj[u][v] — edge b/w u → v

* Size of adjacency matrix ≡ N × N → $O(n^2)$

* Adjacency list



0 -> []
1 -> []
2 -> []
3 -> []
4 -> []
5 -> []

4 -> [1, 3, 2]

adjList = [n+1] list Space ≡ O(2 × E)

(for undirected

(u) ⟶ (v)

adj[u].add(v)

adj[v].add(u)

Space ≡ O(E)

for directed graph

* Representing of weights

(u) ⟷ (v)

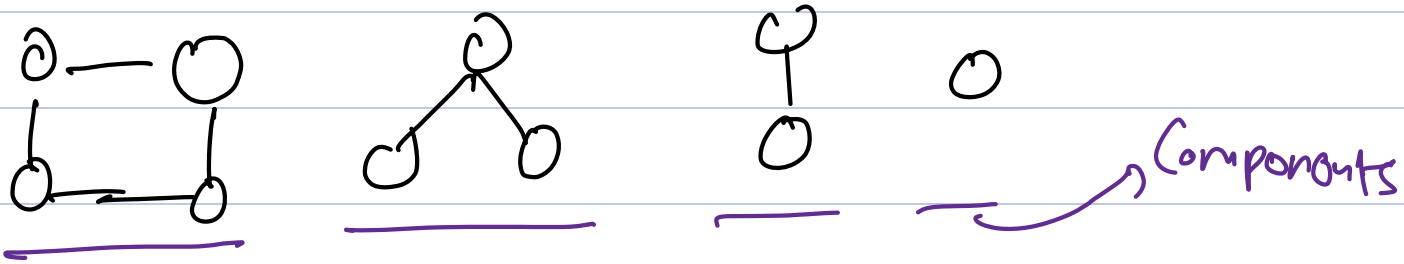
adjacency matrix adj[u][v] = weight

adjacency list

adj[u].add((v, weight))

↳ store pairs

* Connected components



- every vertex in graph is connected to another
- or there exist a path b/w any two pair of vertices

* Always maintain a visited Array in traversal algorithms

for every vertex v in V:

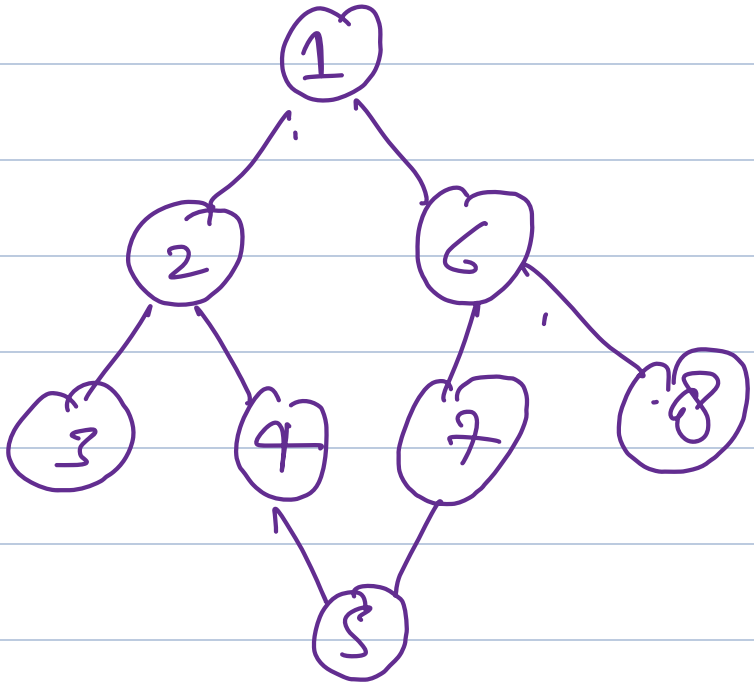
if v is not visited:

traverse(v)

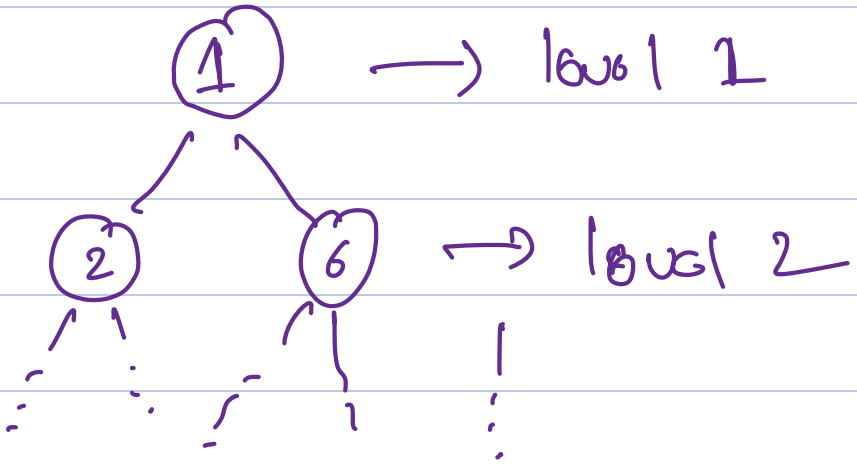
TRAVERSAL TECHNIQUES

- **BFS**

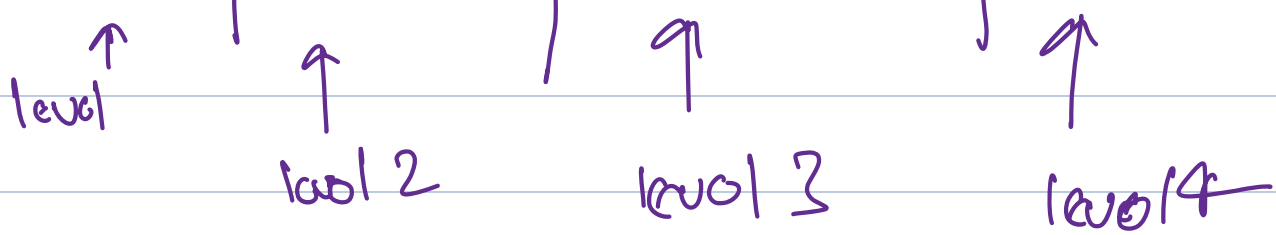
N = 8



Breadth ≡ level order traversal



⇒ 1 | 2 6 | 3 4 7 8 | 5



* Change the starting node and the traversal changes

* Implementation

- Take a Queue
- Add the first node to the Queue
- created a visited Array of size [N+1]
- visited [startNode] = true

---

initial Configuration

Steps → 1 → Take out the node from q

2 → visit it

3 → add all the neighbours to the queue and mark them visited

SC = O(N)

TC = node goes once into BFS queue and for every node the inside loop will run for every degree

$$O(N) + O(2E)$$

- DFS

Explore the Depths and memorize where you came from

↳ looks like recursion and recursion stack will remember all things

* Implementation

↳ create visited Array

→ mark the starting as visited

→ Call a recursive DFS on the start

DFS(v)

→ visit[startNode] = True

→ visit it

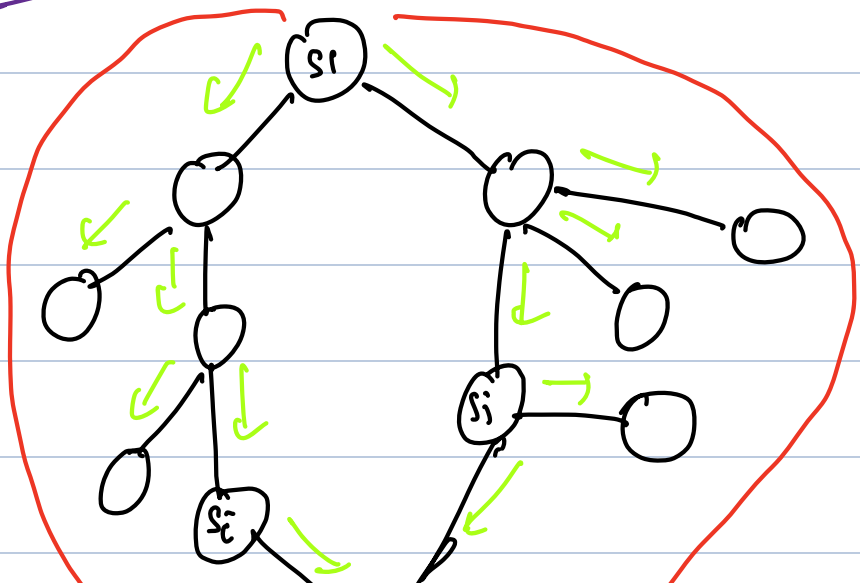
→ for every neighbor of v:

if neighbor is not visited:

DFS(neighbor)

(*) Detect a cycle in an undirected graph

BFS Intuition



you started from a node and you encounter S_n which is already visited, this can only happen if there is exist another path from where you started. This will form a cycle

* conditions —)

neighbour = S_n

currentNode = S_c

if S_n is not visited :

visit it

else if S_n is visited and S_n is a parent of S_c :

ignore

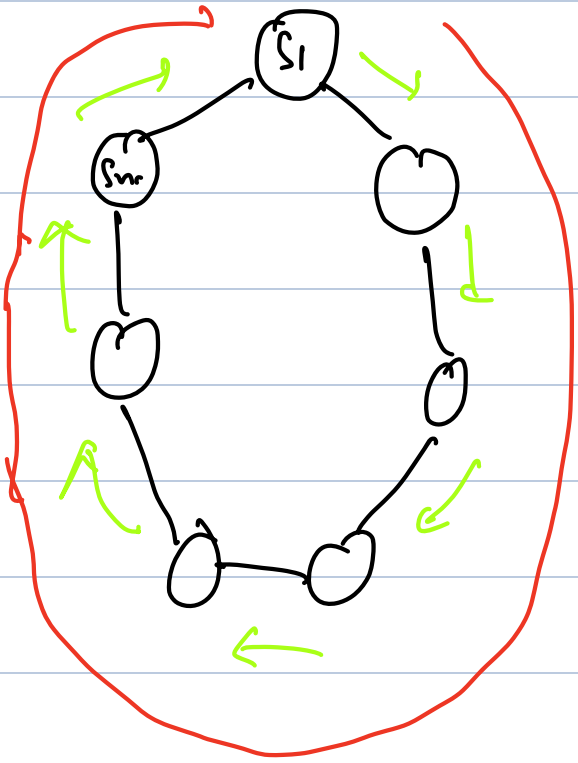
else S_n is visited and S_n is not the parent of S_c :

cycle detected

* Maintain a parent as well in Queue

* DFS

Intuition



you start at a node
you got to depth
and you encounter a visited node that is not where you came from

S_1 — start node

currentNode — S_n

S_i → parent Node

```
DFS ( currentNode , grid parent ):

    visit [currentNode] = True

    for neighbor in grid [currentNode]:
```

if neighbour is not visited:

visit it

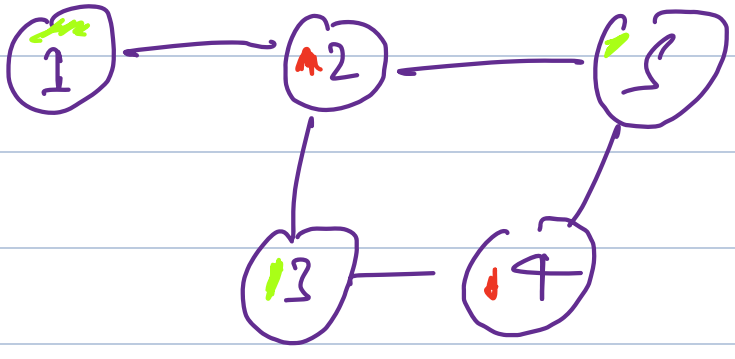
elif neighbour is visited and

neighbour != parent:

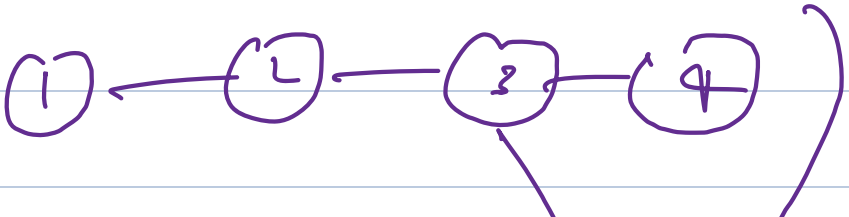
cycle detected.

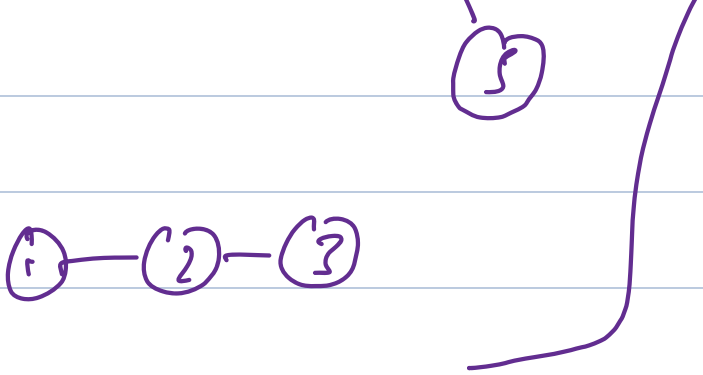
* Bipartite Graph

Can be coloured with 2 colors such that no adjacent vertex are of same color



* linear graphs are always bipartite





(*) Any graph with even cycle length can also be partite (BI)

odd cycle length is not a bi partite

↳ can never

Check for bi partite using BFS

(1) Color (visited) = $[v_1, v_2 \ldots\ldots v_n]$

(-1) ⟶ not colored yet

(2) color → 0

color → 1

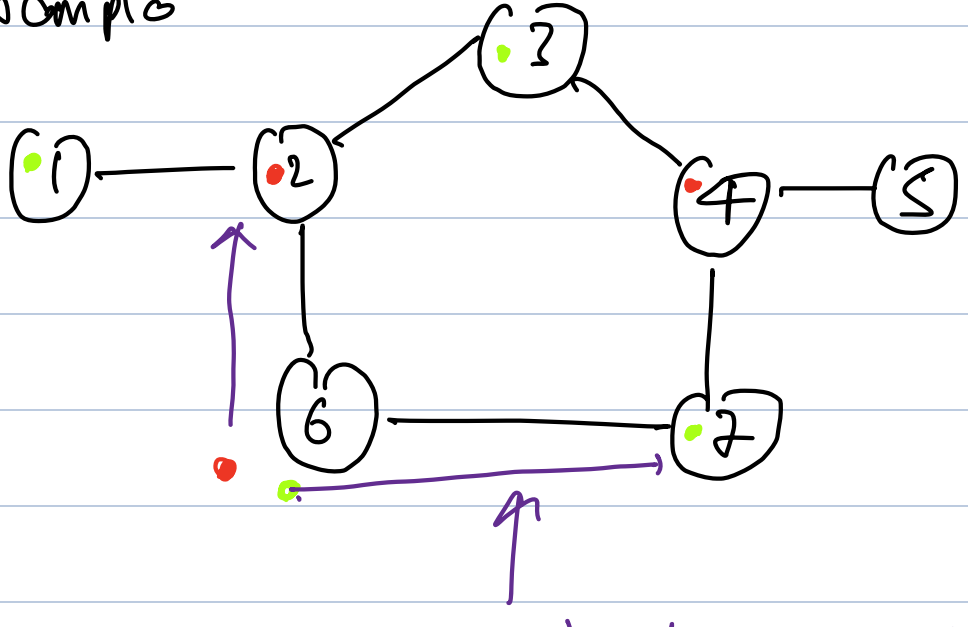
(3) Start with any node

→ put it into queue

→ color it with any color 0 or 1

```
len(queue) > 0:
  node = queue.popleft()

  for every neighbour of node:

    if color[neighbour] not colored:
        color[neighbor] = !node_color
        queue.add(neighbour)

  else if neighbour is colored:
      if color of neighbour is correct:
          continue

  else → cannot be bipartite
```

Example



odd length cycle

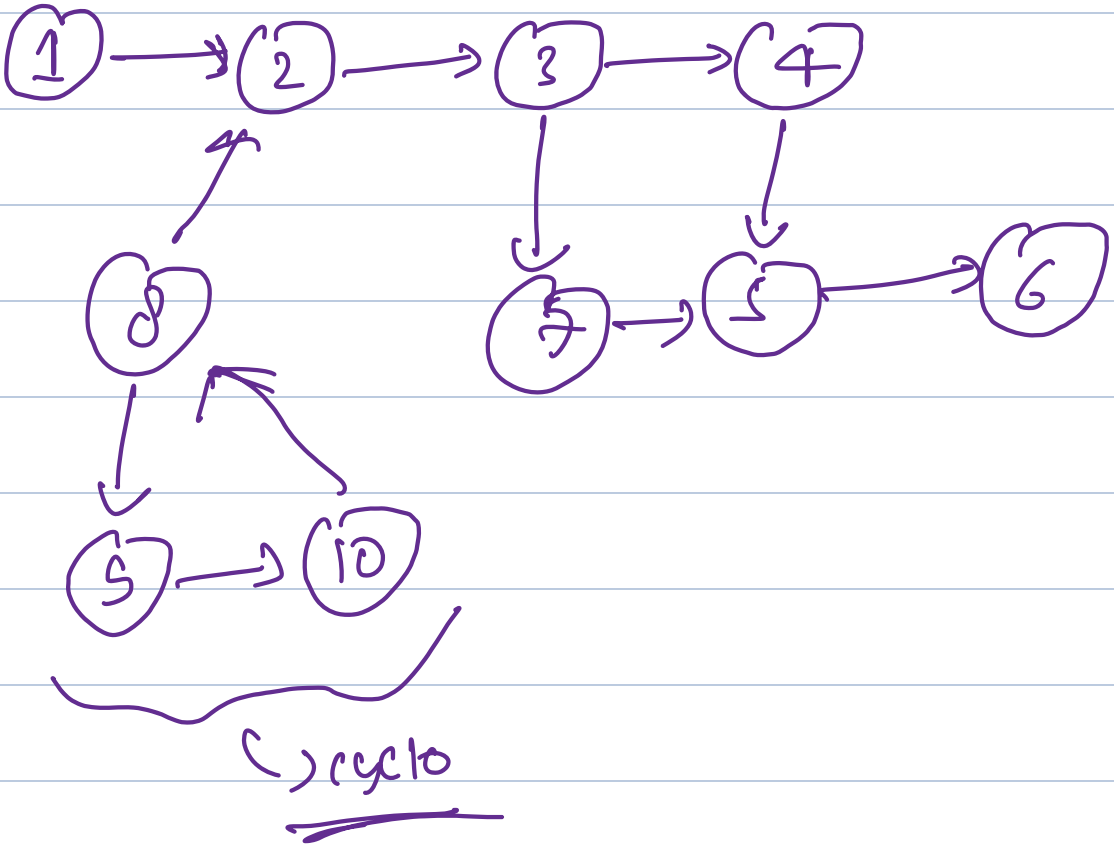
Note Graph can be disconnected

try all vertex

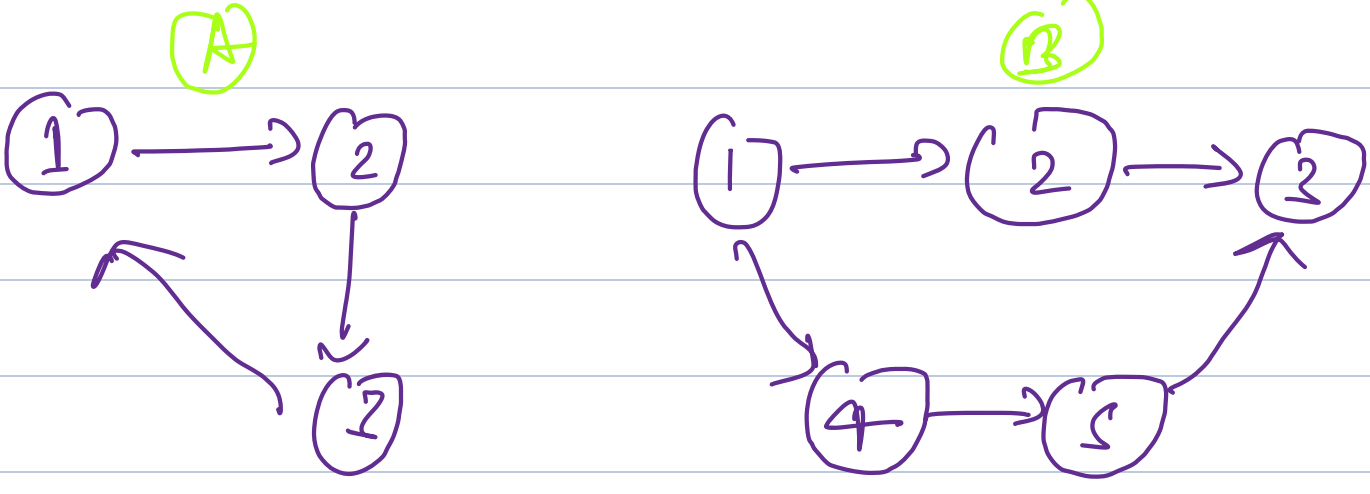
* Using DFS

```
DFS ( vertex , initial_color ):

    Color the vertex = initial_color

    for every neighbor of vertex:

        if color of neighbor == initial_color:

            not a bipartite

        elif neighbor is not colored:

            if DFS ( neighbor , ! initial_color )
                == false
                return not a bipartite
```

Detect a cycle in a Directed graph



↳ cycle



cycle

↳ not a cycle

understand the diff in intuition of DFS

* On graph (A) if you encounter a node that is visited then its a cycle

* On a graph (B) the same logic does not work

* DFS

Maintain a path visited array

visited = [ ]

path-visited = [ ]

Intuition

when you visit a vertex in DFS keep

visited [vertex] = True

path-visited [vertex] = True

but when you return from this vertex in DFS call you mark path-visited [vertex] = False

* If you find a vertex that is visited and you find it's on the same path from which you started then it's a cycle.

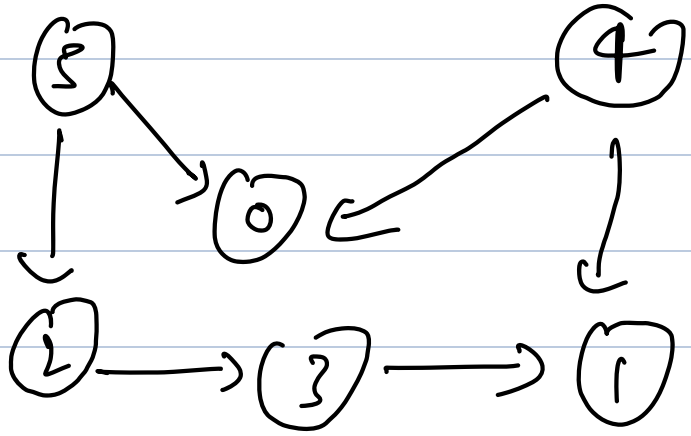
* if a vertex is visited and it's not on the same path, that means some else path exist from where you started and cannot form a cycle.

```
class Solution:

    #Function to detect cycle in a directed graph.
    def isCyclic(self, V : int , adj : List[List[int]]) -> bool :
        # code here
        visited = [False]*V
        path_visited = [False]*V

        for i in range(V):
            if visited[i] is False:
                if self.DFS(i, adj, visited, path_visited) is True:
                    return True
        return False
    def DFS(self, vertex, adj, visited, path_visited):
        visited[vertex] = True
        path_visited[vertex] = True
        for neighbor in adj[vertex]:
            if visited[neighbor] is False:
                if self.DFS(neighbor, adj, visited, path_visited) is True:
                    return True
            elif visited[neighbor] is True and path_visited[neighbor] is True:
                return True
        path_visited[vertex] = False
        return False
```

⋆ Topological Sorting

only exists on DAG → Directed Acyclic graph

→ Any linear ordering of vertices such that if there is any edge b/w u and v then u appears before v in that ordering.



5 4 2 3 1 0

4 5 2 3 1 0

} → Topologically sorted ordering

* Using DFS

Traverse every node using DFS, but when you return from a DFS call, put it into a stack.

The resultant stack is one of the linear orderings

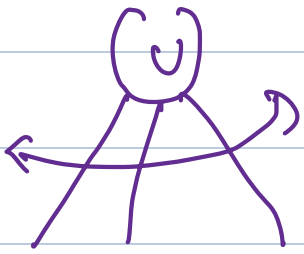
```
class Solution:

    #Function to return list containing vertices in Topological order.
    def topoSort(self, V, adj):
        # Code here
        visited = [False] * V
        stack = []
        for vertex in range(V):
            if visited[vertex] is False:
                self.DFS(vertex, adj, visited, stack)

        stack.reverse()
        return stack
    def DFS(self, u, adj, visited, stack):
        visited[u] = True
        for v in adj[u]:
            if visited[v] is False:
                self.DFS(v, adj, visited, stack)
        stack.append(u)
```

(2) Using BFS also known as

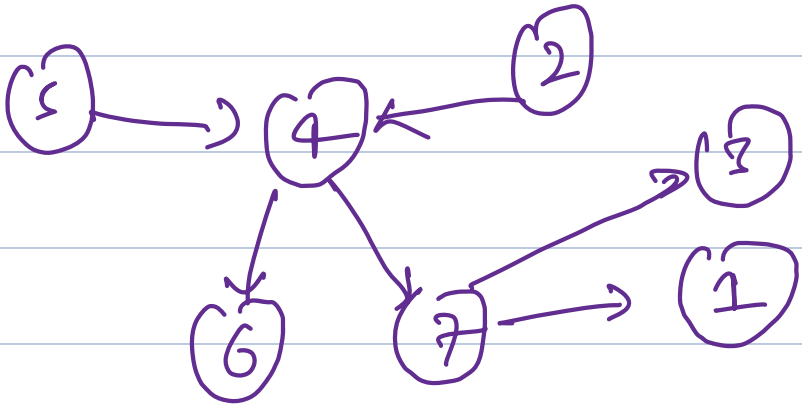
Kahn's Algorithm

Intuition

BFS traverse level wise (or breadth wise)



we have to keep those nodes first which have no incoming dependency (or in degree)



(5) has no in degree or incoming edge and have should appear first (intuition)

So take all the vertex which have in degree zero in queue and start with them

≡ Starting point → [5 2] or [2 5]

Now, traverse the adjacent nodes

maintain a in-degree array of each vertex

[1 0 1 2 0 1 1]

1 2 3 4 5 6 7

* For each vertex that is taken out of queue, add it to answer as its indegree is zero and will come before every one else

* while traversing a node's neighbor (v) if it's indegree count becomes zero, add it to the queue only then.

```
from collections import deque
class Solution:
    # Kahn's Algorithm
    #Function to return list containing vertices in Topological order.
    def topoSort(self, V, adj):
        # Code here
        in_degree = [0]*V
        queue = deque()
        # construct the in_degree list
        for i in range(V):
            for vertex in adj[i]:
                in_degree[vertex] += 1

        # Get nodes with 0 in_degree
        for i in range(V):
            if in_degree[i] is 0:
                queue.append(i)

        # create empty answer's list
        answer = []
        while len(queue) > 0:
            u = queue.popleft()
            answer.append(u)
            for v in adj[u]:
                in_degree[v] -= 1
                if in_degree[v] is 0:
                    queue.append(v)
        return answer
```

* patterns

Any where you see if a ordering of tasks is needed where one task depends upon another → apply Topo sort

* Shortest path in DAG using Topo sort

(1) Figure out the topo ordering in a stack

(2) Initialize a distance array

(3) pop a vertex from stack and update its neighbours in adj. list → distance[]

```python
class Solution:
    def shortestPath(self, n : int, m : int, edges : List[List[int]]) -> List[int]:
        # convert the given edges and wights to adjacency list
        adj = []
        for i in range(n):
            adj.append([])
        for i in range(m):
            u, v, wt = edges[i]
            adj[u].append([v, wt])

        # Get the topological ordering
        visited = [False] * n
        topo_order = []
        self.topo(0, adj, visited, topo_order)

        # Create a distance array and mark source vertex
        distance = [-1] * n
        distance[0] = 0

        # Tranverse through the topological ordering and start relaxing their weights
        while len(topo_order) > 0:
            vertex = topo_order.pop()
            for neighbor in adj[vertex]:
                v, wt = neighbor
                if distance[v] == -1 or distance[v] > distance[vertex] + wt:
                    distance[v] = distance[vertex] + wt
        return distance

    def topo(self, u, adj, visited, topo_order):
        visited[u] = True
        for v in adj[u]:
            vertex, wt = v
            if visited[vertex] is False:
                self.topo(vertex, adj, visited, topo_order)
        topo_order.append(u)
```