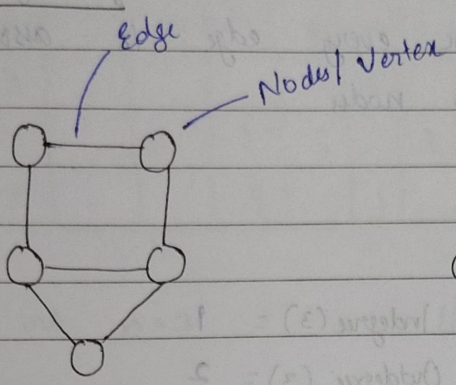
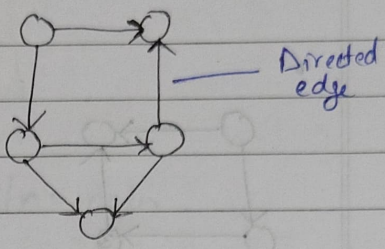
PAGE NO.
DATE / /

Graph

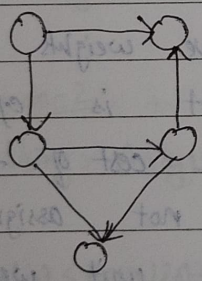


Undirected graph

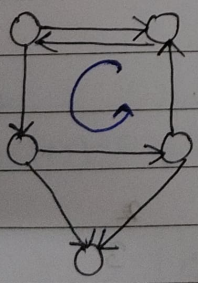


Directed graph

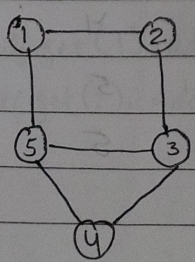
- If atleast one cycle present in the graph then it is called Undirected Cyclic graph



— Directed Acyclic Graph (DAG)



— Directed Cyclic Graph



Degree — Number of edges that go inside or outside that node.

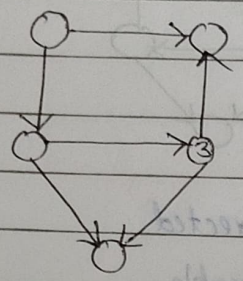
D(3) = 3

D(4) = 2

PAGE NO.
DATE / /

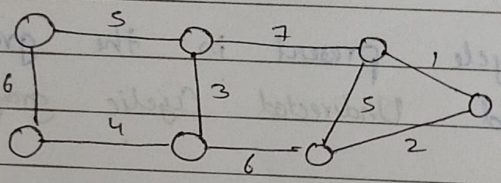
Property - Total degree of a graph is equal to twice the number of edges. This is because every edge is associated to two nodes.

•



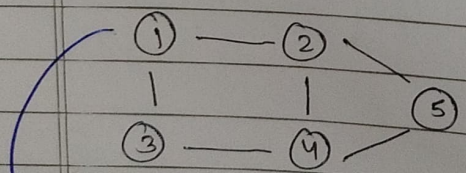
Indegree (3) = 1
Outdegree (3) = 2

•



Edge weight - Graph may have weights assigned on its edges. It is often referred to as the cost of the edge.
- If weights are not assigned then we assume the unit weight. i.e, 1.

• Graph Representation in Java



adj [n+1] [n+1]

1 based indexing

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

Approach - 1

	0	1	2	3	4	5
0						
1			1	1		
2		1		1	1	1
3					1	
4			1	1		1
5			1		1	

Space
$O(N \times N)$

int n = 3 // n = nodes

int adj[][] = new int[n+1][n+1];

adj[1][2] = 1
adj[2][1] = 1
} edge 1 - - - 2

adj[2][3] = 1
adj[3][2] = 1
} edge 2 - - - 3

Approach-2
Using ArrayList

List -> n+1

ArrayList<ArrayList<>> adj

0 -> ~~{}~~
1 -> {2, 3}
2 -> {1, 4, 5}
3 -> {1, 4}
4 -> {2, 3, 5}
5 -> {2, 4}

Space
$O(2m)$

adj.get(1).add(2)
adj.get(2).add(1)
} edge 1 - - - 2

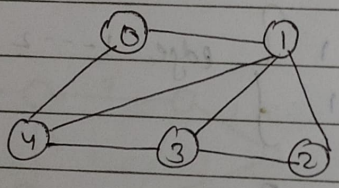
```
for(int i = 1 ; i<n ; i++){
    for(int j=0 ; j < adj.get(i).length ; j++){
        print(adj.get(i).get(j));
    }
}
```

Printing all the edges

- Print adjacency list

I/P- V = 5, E = 7

edges = [[0,1], [0,4], [4,1], [4,3], [1,3], [1,2], [3,2]]



O/P- [[1,4], [0,2,3,4], [1,3], [1,2,4], [0,1,3]]

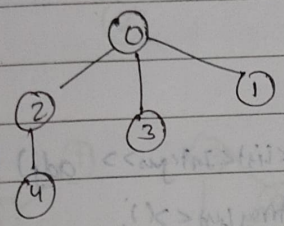
```
public List<List<Integer>> printGraph(int V, int edges[][]){
    List<List<Integer>> list = new ArrayList<>();
    for(int i=0 ; i<=V ; i++){
        list.add(new ArrayList<>());
    }
    for(int i=0 ; i < edges.length ; i++){
        list.get(edges[i][0]).add(edges[i][1]);
        list.get(edges[i][1]).add(edges[i][0]);
    }
    return list;
}
```

	0	1
0	0	1
1	0	4
2	4	1
3	4	3
4	1	3
5	1	2
6	3	2

PAGE NO.
DATE / /

- Breadth First Search (BFS) : Level Order Travel

Inp: adj = [[2,3,1], [0], [0,4], [0], [2]]



O/P - [0, 2, 3, 1, 4]

Adjacency list will be provided in question

```
ArrayList<Integer> list = new ArrayList<>();
Queue<Integer> q = new LinkedList<>();
boolean vis[] = new boolean[V];
q.offer(0);
vis[0] = true;
while(!q.isEmpty()){
    Integer temp = q.poll();
    list.add(temp);
    for(Integer x : adj.get(temp)){
        if(!vis[x]){
            vis[x] = true;
            q.offer(x);
        }
    }
}
return list;
```

while the queue is not empty, dequeue a vertex, add it to the result list, and enque its unvisited neighbors

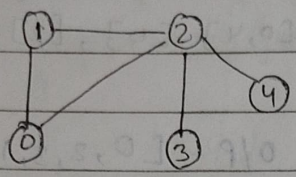
TC - O(N) + O(2E)
SC - O(3N)

this is for the inner loop, we are traversing nodes connecting to it means it will be equal to degree.
D = 2E

PAGE NO.
DATE / /

- Depth First Search (DFS) of Graph

I/P :



adj = [[1,2], [0,2], [0,1,3,4], [2], [2]]

```
public List<Integer> dfsOfGraph(List<List<Integer>> adj) {
    List<Integer> list = new ArrayList<>();
    boolean vis[] = new boolean [adj.size()];
    dfs(0, vis, list, adj);
    return list;
}
```

Recursion

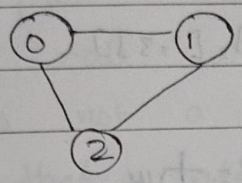
```
public void dfs(int val, boolean vis[], List<Integer> list,
                List<List<Integer>> adj) {
    vis[val] = true;
    list.add(val);
    for(int x : adj.get(val)) {
        if(!vis[x]) {
            dfs(x, vis, list, adj);
        }
    }
}
```

TC - O(N) + 2E

SC - O(N) + O(N) + O(N) ≅ O(N)

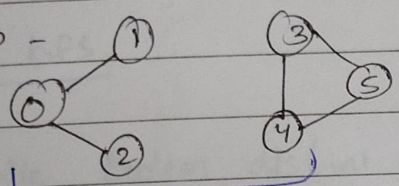
For list — for visited array — Recursion stack space

- Find if path exists in Graph

I/P - 

, source = 0 , destination = 2

O/P - true

I/P - 

, source = 0 , destination = 5

O/P - false

These are 2 different component of same graph

```
boolean vis[] = new boolean [n];
Queue<Integer> q = new LinkedList<>();
q.offer (source);
vis[source] = true;
while (!q.isEmpty()){
  int val = q.poll();
  for (int x : list.get(val)){
    if (!vis[x]){
      q.offer(x);
      vis[x] = true;
    }
  }
}
return vis[destination];
```

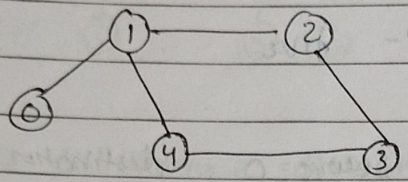
TC - O(V+E)
SC - O(N)

If both source and destination are in same component then we will get the destination by traversal (BFS/DFS)

PAGE NO.
DATE / /

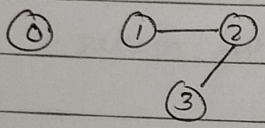
- Cycle detection in Undirected graph

I/P - adj: [[1], [0,2,4], [1,3], [2,4], [1,3]]



O/P - true

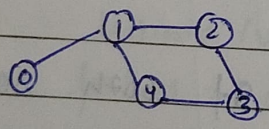
I/P - adj: [[], [2], [1,3], [2]]



O/P - false

From both ways we can solve (BFS / DFS)

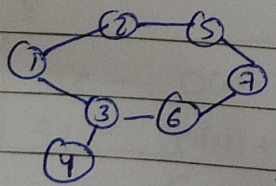
- BFS



(3,2)
(4,1)
(2,1)
(1,0)
(0,-1)

Queue

- DFS



The intution is we are carrying the node and the parent node. While traversing if there is a node which is already visited and that is not a parent of current node then this means that node was previously in the path

For Dfs

```
public boolean dfs(int src, int parent, boolean vis[], List<List<
                    Integer>> adj) {
    vis[src] = true
    for(int x : adj.get(src)){
        if(!vis[x]){
            if(dfs(x, src, vis, adj)){
                return true;
            }
        } else if(x != parent){
            return true;
        }
    }
    return false;
}
```

Main logic

```
public boolean isCycle( List<List<List<Integer>> adj ){
    int V = adj.size();
    boolean vis[] = new boolean[V];
    for(int i=0 ; i<V ; i++){
        if(!vis[i]){
            if(dfs(i, -1, vis, adj)){
                return true;
            }
        }
    }
    return false;
}
```

Main function

initial parent of node

• for BFS

```
public boolean bfs(List<List<Integer>> adj, int src, boolean vis[]){
    Queue<Pair> q = new LinkedList<>();
    q.offer(new Pair(src, -1));
    vis[src] = true;
    while(!q.isEmpty()){
        Pair p = q.poll();
        int val = p.val;
        int parnt = p.parnt;
        for(int x: adj.get(val)){
            if(!vis[x]){
                q.offer(new Pair(x, val));
                vis[x] = true;
            } else if(parnt != x){
                return true;
            }
        }
    }
    return false;
}

public boolean isCycle(List<List<Integer>> adj){
    int V = adj.size();
    boolean vis[] = new boolean[V];
    for(int i=0; i<V; i++){
        if(!vis[i]){
            boolean ans = bfs(adj, i, vis);
            if(ans){
                return true;
            }
        }
    }
    return false;
}
```

Parent node

Node

PAGE NO.
DATE / /

- Bipartite graph

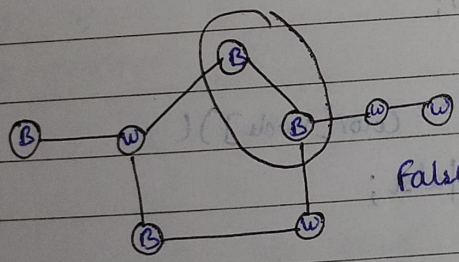
If we are able to colour a graph with two colours such that no adjacent nods have the same color, it is called bipartite graph.

- If graph is linear
- If cycle of graph contains even number of edges.

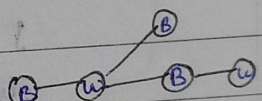
} In these cases graph will be Bipartite.

- If cycle of graph contains add number of edges

} Not a Bipartite graph

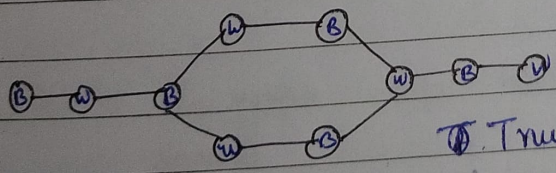


False



True

B - Black
W - White



True

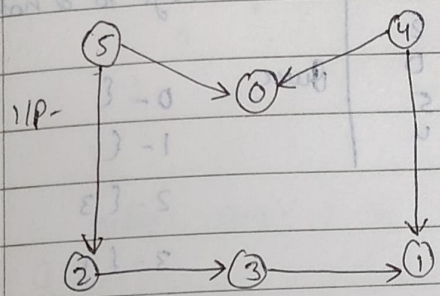
- Using BFS

```
public boolean helper(int start, int V, int color[],
        ArrayList<ArrayList<Integer>> adj){

    Queue<Integer> q = new LinkedList<>();
    q.offer(start);
    color[start] = 0;
    while(!q.isEmpty()){
        int node = q.poll();
        for(int x : adj.get(node)){
            if(color[x] == -1){
                color[x] = 1 - color[node];
                q.offer(x);
            }
            else if(color[x] == color[node]){
                return false;
            }
        }
    }

    return true;
}

public boolean isBipartite(ArrayList<ArrayList<Integer>> adj){
```

PAGE NO.
DATE / /

• Topological Sort Algorithm

Topological Sorting of a directed acyclic graph (DAG) is an linear ordering of its vertices such that for every directed edge $u \rightarrow v$, vertex u appears before vertex v in the ordering.



I/P -

O/P - 5, 4, 2, 3, 1, 0

u v

5 → 0
4 → 0
5 → 2
2 → 3
3 → 1
4 → 1

u appears before v

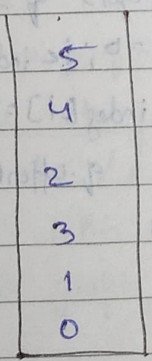
TC - O(V+E)
SC - O(N) + O(N)

The above result satisfies all the necessary conditions.

Note - A graph may have multiple topological sortings.

Using stack in dfs for topological

```
dfs() {
  vis[ind] = true;
  for(int x : adj.get(ind)){
    if(!vis[x]){
      dfs(x, st, vis, adj);
    }
  }
  st.push(ind);
}
```

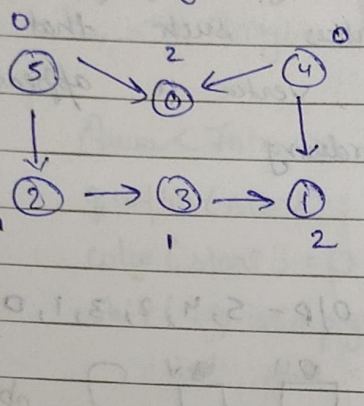


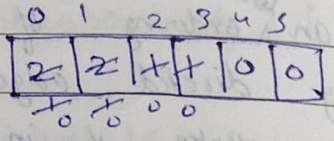
dfs(0) — ①
dfs(1) — ②
dfs(2) — ③
dfs(3)
dfs(4) — ④
dfs(5) — ⑤

PAGE NO.
DATE / /

- Kahns algorithm

Topological sort solved using modified BFS





No. of incoming edge to a node

Queue

1
3
2
0
5
4

0 - {
1 - {
2 - {3
3 - {1
4 - {0,1
5 - {0,2

TC - O(V+E)
SC - O(N)+O(N) ~ O(N)

```
static List<Integer> topo (List<List<Integer>> adj){

    int indeg[] = new int[V]
    for(int i=0; i<adj.size(); i++){
        for(int x: adj.get(i)){
            indeg[x]++;
        }
    }

    Queue<Integer> q = new LinkedList<>();
    for(int i=0; i<indeg.length, i++){
        if(indeg[i]==0){
            q.offer(i);
        }
    }
```

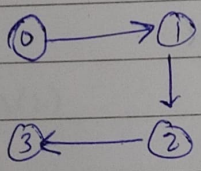
- Calculating in-degree for all nodes
- This helps us identify nodes with no dependency (indegree=0)

- All nodes with indegree 0 is added to queue
- These nodes have no dependency and can be processed first.

```
while(!q.isEmpty()){
    int node = q.poll();
    ans.add(node);
    for(int x : adj.get(node)){
        indeg[x]--;
        if(indeg[x]==0){
            q.offer(x);
        }
    }
}
return ans;
}
```

- Process the nodes in the queue using BFS
- Remove a node from the queue, add it to the result, and reduce the ~~in-depth~~ indegree of its neighbours
- If a neighbor's in-degree becomes 0, add it to the queue as its now ready to be processed.

- Cycle detection in Directed graph

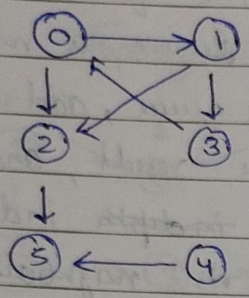


Intuition :- We will use Kahn's algorithm
- What happens in topological ~~sort~~ sort is that if it is DAG then only all the nodes will get into the queue (indeg[x]==0 → q.offer(~~node~~ x)) and while poping we are storing that into list.
- So if [list.size() != V] this means in it is not a DAG and there is a cycle in graph

* Problem based on Kahn's algorithm
- Course schedule I
- Course schedule II

PAGE NO.
DATE / /

• Find Eventual Safe State



Terminal node - If there are no outgoing edges

Safe node - If every possible path starting from that node leads to a terminal node.

5,6 → terminal node.

```
public boolean dfs(i, vis[], pathVis[], adj){

    vis[i] = true;
    pathVis[i] = true;
    for(int x : adj.get(i)){
        if(!vis[x]){
            if(dfs(x, vis, pathVis, adj)){
                return true;
            }
        } else if(pathVis[x]){
            return true;
        }
    }
    pathVis[i] = false;
    return false;
}
```

If the neighbor is already in the current path, a cycle exists

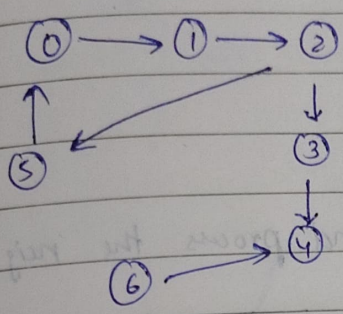
Backtrack and mark the Node as Not part of current path

Return false if no cycle detected.

```
main(){
    for(int i=0; i<V; i++){
        if(!vis[i]){ dfs(i, vis, pathVis, adj); }
    }
    for(int i=0; i<V; i++){
        if(!pathVis[i]){ List.add(i); }
    }
    return list
}
```

PAGE NO.
DATE / /

Intution



0 - { 1
1 - { 2
2 - { 3, ~~6~~, 5
3 - { 4
4 - {
5 - { 0
6 - { 4

vis:

F	F	F	F	F	F	F

pathVis -

F	F	F	F	F	F	F

dfs(0)

T	F	F	F	F	F	F
T	F	F	F	F	F	F

dfs(1)

T	T	F	F	F	F	F
T	T	F	F	F	F	F

dfs(2)

T	T	T	F	F	F	F
T	T	T	F	F	F	F

dfs(3)

T	T	T	T	F	F	F
T	T	T	T	F	F	F

dfs(4)

T	T	T	T	T	F	F
T	T	T	T	T	F	F

~~dfs(5)~~ No neighbor of 4, so back track (terminal node)

T	T	T	T	T	F	F
T	T	T	T	F	F	F

return false to node 3

Node 3 has no more neighbors, so backtrack

T	T	T	T	T	F	F
T	T	T	F	F	F	F

Backtracked to Node 2 and process the neighbor visit Node 5

T	T	T	T	T	T	F
T	T	T	F	F	T	F

dfs(s) ~~Visit~~ Visit Node 0

Node 0 is already visited and is in the current path (pathVis[0] = true)

Cycle detected, backtrack

PAGE NO.
DATE / /

• Alien Dictionary

Given sorted dictionary of an alien language having N words and K starting alphabets of a standard dictionary. Find the order of characters in the alien language.

I/P: N = 5, K = 4
dict = {"baa", "abcd", "abca", "cab", "cad"}

O/P: bdac

Comparing
baa
abcd
b → a

Comparing
abcd
abca
b → a
d

Comparing
abca
cab
b → a → c
d

Comparing
cab
cad
b → a → c
d
final directed graph

• 2 follow up question for interview

- when is the ordering not possible
  i) If every character matches and the largest word appears before the shortest word
  Eg → "abcd" appears before "abc".

  ii) If there exist a cyclic dependency between the characters.
  Eg → {"abc", "bat", "ade"}
  'a' < 'b' < 'a', which is not possible.

PAGE NO.
DATE / /

```
public String findOrder(String dict[], int K){
    int dict.length;
    List<List<Integer>> adj = new ArrayList<>();
    for(int i=0; i<K, i++){
        adj.add(new ArrayList<>());
    }
    for(int i=0; i<n-1, i++){
        String s1 = dict[i];
        String s2 = dict[i+1];
        int len = Math.min(s1.length(), s2.length());
        for(int j=0; j<len; j++){
            if(s1.charAt(j) != s2.charAt(j)){
                adj.get(s1.charAt(j)-'a').add(s2.charAt(j)-'a')
                break;
            }
        }
    }
    List<Integer> list = topo(adj, K);
    String ans = "";
    for(int i=0; i<list.size(); i++){
        ans += (char)(list.get(i) + (int)'a');
    }
    if(ans.length() != K){
        return "";
    }
    return ans;
}

public List<Integer> topo(adj, K)
    --- Kahns algorithm ---
    return list
}
```

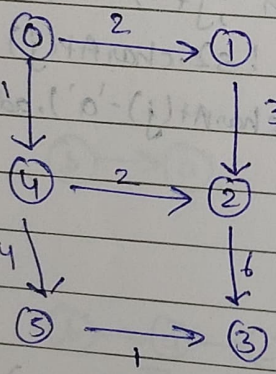
PAGE NO.
DATE / /

• Shortest path in Directed Acyclic Graph

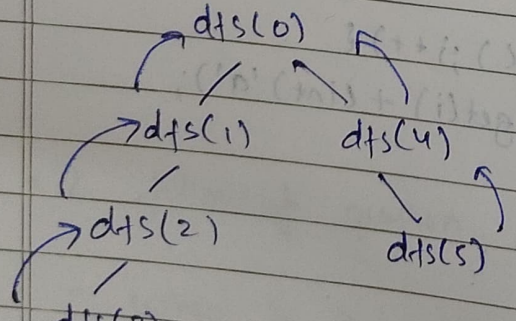
Shortest path from the source to all other nodes we assume the source vertex to be '0'
Weight of the edges is already provided.

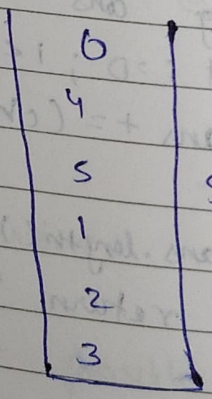
I/P : V = 6 , E = 7
edges = [[0,1,2], [0,4,1], [4,5,4], [4,2,2], [1,2,3], [2,3,6], [5,3,1]]

O/P : [0, 2, 3, 6, 1, 5]



0 - { 1, 4
1 - { 2
2 - { 3
3 - {
4 - { 2, 5
5 - { 3





Dist :

0	1	2	3	4	5
0	2	3	6	1	5

PAGE NO.
DATE / /

```
public int[] shortestPath(int V, int E, int[][] edges) {
    List<List<Pair>> adj = new ArrayList<>();
    for(int i=0; i<V ,i++){
        adj.add(new ArrayList<>());
    }
    for(int i=0; i<E ,i++){
        int u = edges[i][0];
        int v = edges[i][1];
        int w = edges[i][2];
        adj.get(u).add(new Pair(v,w));
    }
    boolean vis[] = new boolean[V];
    Stack<Integer> st = new Stack<>();
    for(int i=0; i<V; i++){
        if(!vis[i]){
            dfs(i, vis, st, adj);
        }
    }
    int dist[] = new int[V];
    for(int i=0; i<V; i++){
        dist[i] = (int)1e9;
    }
    dist[0] = 0;                    // Distance of source vertex
    while(!st.isEmpty()){
        int node = st.pop();
        for(int i=0; i<adj.get(node).size(); i++){
            int val = adj.get(node).get(i).val;
            int w = adj.get(node).get(i).weight;
            if(dist[node] + w < dist[val]){
                dist[val] = dist[node] + w;
            }
        }
    }
}
```

Adding edge (u → v) with weight w

Performing topological sorting using DFS

Set all distance to "infinity" initially

Process vertices in topological order to find shortest paths