~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Graph Revision (Theory + Problems)

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Agenda of the session :

1. FAQs about the session.
2. Introduction and Type of Graphs
3. Traversal Algorithms
    a. BFS
    b. DFS
4. Rotten Oranges * connected comps.
5. Topological Sort → Kahn's Algo
6. Dijkstra's Algorithm
7. Cycle in graphs
    a. For directed graph
    b. For undirected graph

~~~~~~~~~~~~~~~~~~~~~~~~~~~~

FAQs about the session

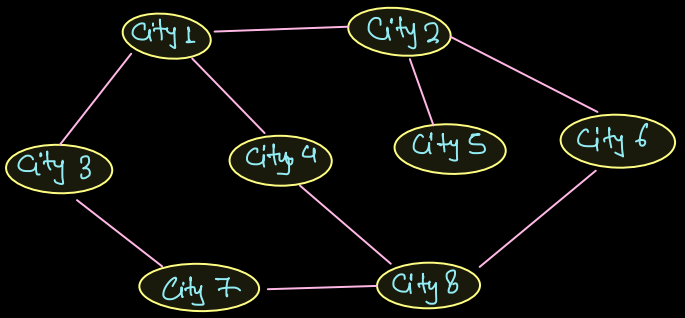
~~~~~~~~~~~~~~~~~~~~~~~~~~~~

- About me?
  Name: Shreesh Tripathi
  Work: SE and Instructor in SCALER
  Teaching Experience: teaching DSA for 3.5 years
  In Scaler: More than 350+ sessions

- Is this session worth it if I didn't attend Graph's session? → Revision Not useful.
- About the Pace of this session? → Pace fast
- How We Proceed in this session, How we revise the content of 3 sessions in 1.
    * Pre-written Notes.
    * Pre-written code [JAVA]
    * Proceed with fast pace.
- Can We Discuss Problems from any other topic?
    * Not any other topic.

* Revision session, not problem solving session.

Introduction and Type of Graphs



Vertex : 8 [city 1, 2, 3, ... 8]

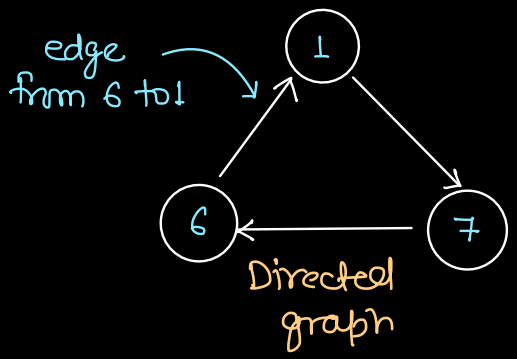
Edge : 9

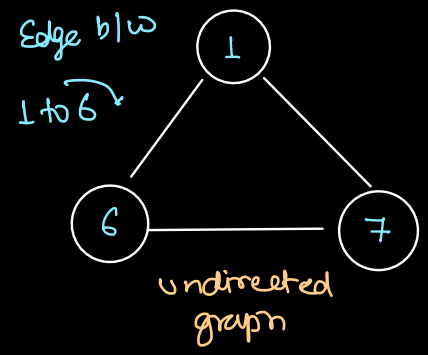
[1-2, 1-3, 1-4, 2-5, 2-6, 3-7, 7-8, 4-8, 6-8]

Graph : Collection of Vertex and Edges is graph.

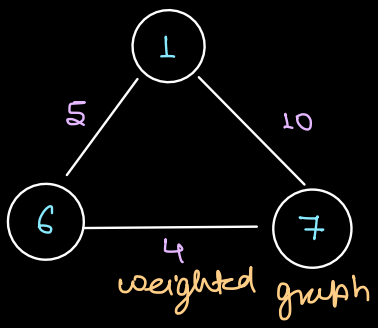
Type of Graph

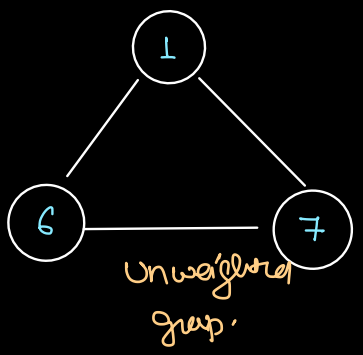
- Undirected Graph
- Directed Graph



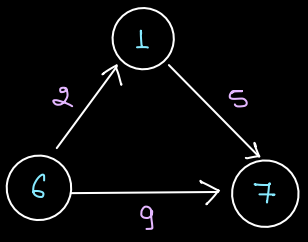


- Weighted Graph
- Unweighted Graph





- A combination of the above is also Also possible

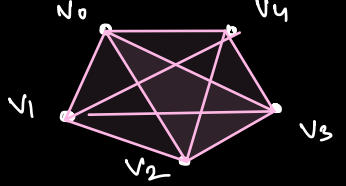


→ directed weighted graph

→ other combination is also possible.

- Complete Graph

Count of Edge in a complete graph:



→ Vertex is connected with each other.

count of edge in complete graph $= {}^{n}C_2 = \frac{n(n-1)}{2}$

How to Implement Graph Data Structure?

- Adjacency Matrix
- Adjacency List

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

Drawback :

→ Wastage of memory.

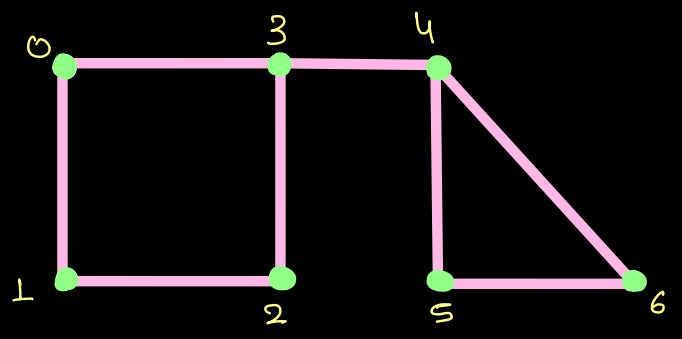
→ n^2

Vertex = 7 , Edges = 8

Edge →

u	v
0	3
0	1
2	3
3	4
1	2
4	5
4	6
5	6

Undirected graph

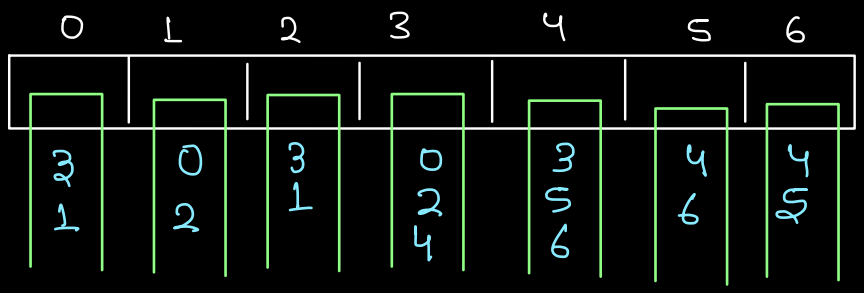


Undirected graph

Vertex = 7 , Edges = 8

Edge →

v_1	v_2
0	3
0	1
2	3
3	4
1	2
4	5
4	6
5	6



HashMap < key, value > graph

key → name of vertex

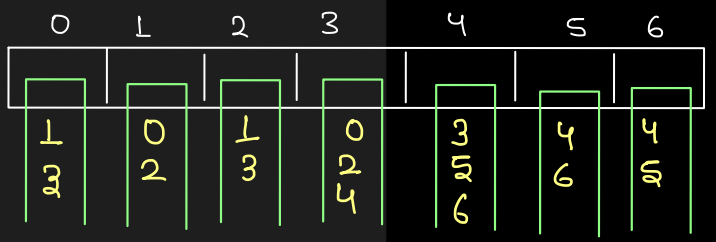
value → AL<nbrs>

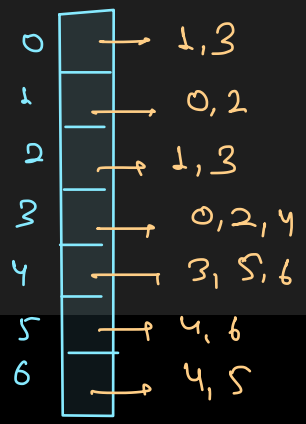
ArrayList < AL< nbrs >> graph

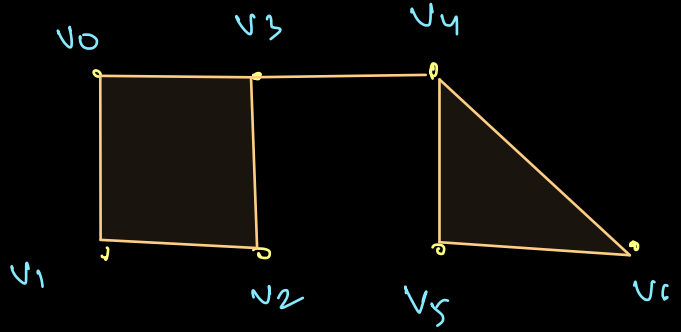
```java
import java.util.*;

class Main {

    public static void addEdge(ArrayList<ArrayList<Integer>> graph,
                               int u, int v) {
        graph.get(u).add(v);
        graph.get(v).add(u);
    }

    public static void display(ArrayList<ArrayList<Integer>> graph) {
        for(int v = 0; v < graph.size(); v++) {
            System.out.print(v + " ->[");
            // graph.get(v) -> neighbours of vth vertex
            for(int nbr : graph.get(v)) {
                System.out.print(nbr + ", ");
            }
            System.out.println("]");
        }
    }

    public static void demo() {
        int vtx = 7;
        int edges = 8;

        ArrayList<ArrayList<Integer>> graph = new ArrayList<>();
        for(int v = 0; v < vtx; v++) {
            graph.add(new ArrayList<>());
        }

        // addEdge(graph, u, v);
        addEdge(graph, 0, 1);
        addEdge(graph, 0, 3);
        addEdge(graph, 1, 2);
        addEdge(graph, 2, 3);
        addEdge(graph, 3, 4);
        addEdge(graph, 4, 5);
        addEdge(graph, 4, 6);
        addEdge(graph, 5, 6);

        display(graph);
    }

    public static void main(String args[]) {
        demo();
    }
}
```

undirected graph, directed → u→v No v→u

0→[1, 3,]
1→[0, 2,]
2→[1, 3,]
3→[0, 2, 4,]
4→[3, 5, 6,]
5→[4, 6,]
6→[4, 5,]







~~~~~~~~~~~~~~~~~~~~

Traversal Algorithms

~~~~~~~~~~~~~~~~~~~~

I. DFS (Depth-First Search) → Recursive Approach, move in Depth first

II. BFS(Breadth-First Search) → Iterative Approach, Explore graph by radial movement.

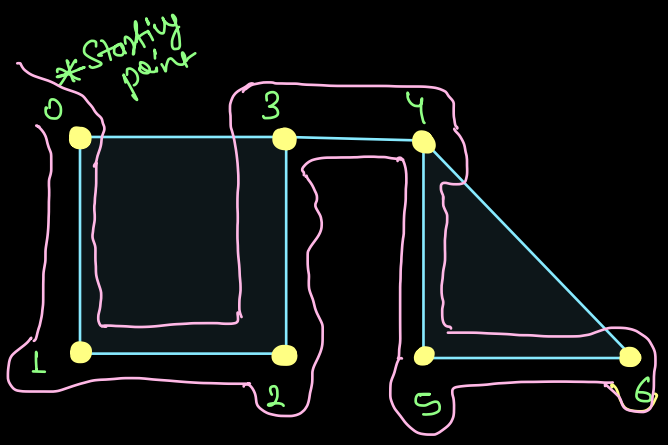
~~~~~~~~~~~~~~~~~~~~~~

DFS: Depth-First Search

~~~~~~~~~~~~~~~~~~~~~~

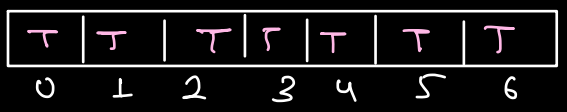
Steps of Depth First Search :

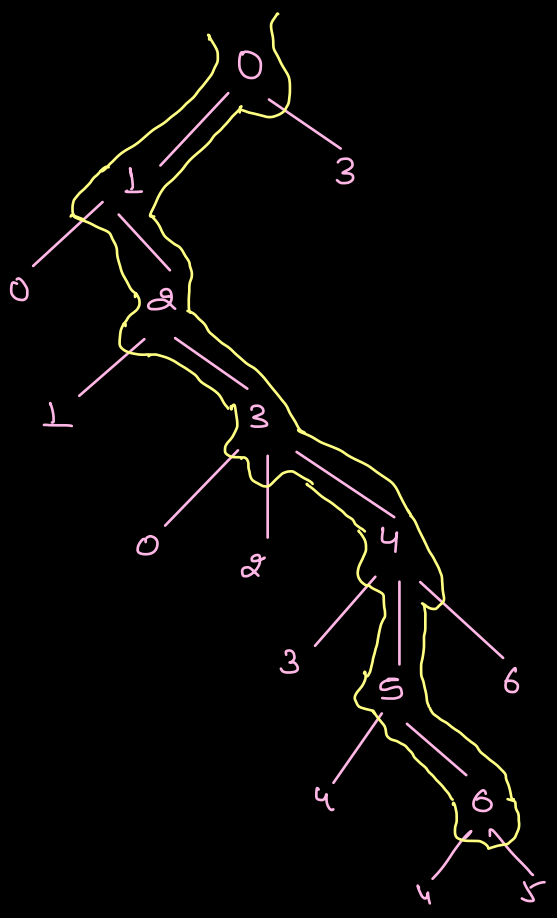
1. Start at the Selected Node
   Before starting, mark that node in the visited array
2. Start Recursive traversal
3. Iterate on all neighbors
4. If not visited, mark it and move toward it
5. Repeat Until All Nodes Are Visited




```java
public static void DFS(ArrayList<ArrayList<Integer>> graph) {
    // total vertex
    int n = graph.size();
    // prepare visited array
    boolean[] vis = new boolean[n];
    // For Now, assume source point
    int src = 0;
    // mark yourself and then make call to helper of DFS function
    vis[src] = true;
    dfsHelper(graph, src, vis);
}

public static void dfsHelper(ArrayList<ArrayList<Integer>> graph,
    int src, boolean[] vis) {
    // 1. Print
    System.out.print(src + " ");
    // 2. move toward unvisited neighbour
    for(int nbr : graph.get(src)) {
        // unvisited neighbour
        if(vis[nbr] == false) {
            // mark that neighbour
            vis[nbr] = true;
            // move toward that neighbour
            dfsHelper(graph, nbr, vis);
        }
    }
}
```

→ path → tracker

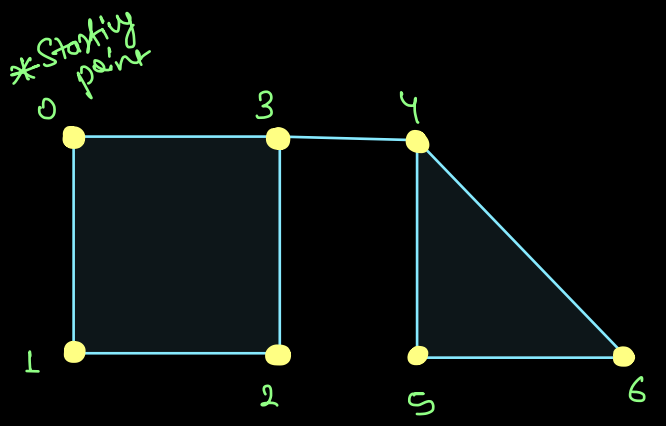


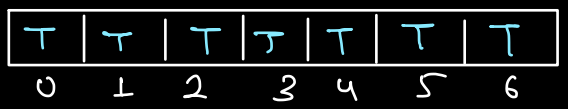


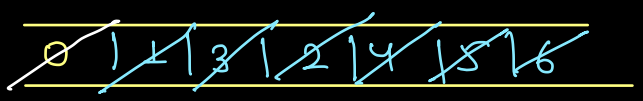
BFS: Breadth-First Search

Steps in the BFS Algorithm:

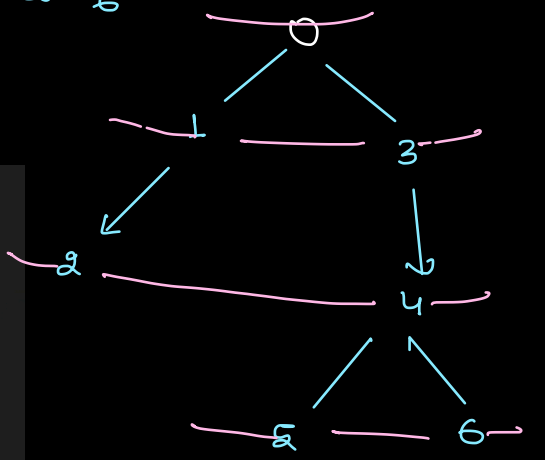
1. Make a Queue and add a source point to it
2. Make a visited array and mark the source point in it
3. Process the Queue until it is empty
    a. Remove
    b. Work → print
    c. Add unvisited neighbor
    While adding it => mark the nbr







o/p: 0 1 3 2 4 5 6



Code (in JAVA)

```java
public static void BFS(ArrayList<ArrayList<Integer>> graph, int src) {
    // number of vertex
    int n = graph.size();
    // creation of boolean array for representation of visited array
    boolean[] vis = new boolean[n];
    // Queue for BFS
    Queue<Integer> qu = new ArrayDeque<>();
    // Add source point in queue and mark it in visited
    qu.add(src);
    vis[src] = true;
    // que traversal
    while(qu.size() > 0) {
        // remove
        int rem = qu.remove();
        // print
        System.out.print(rem + " ");
        // add unvisited neighbours
        for(int nbr : graph.get(rem)) {
            if(vis[nbr] == false) {
                vis[nbr] = true;
                qu.add(nbr);
            }
        }
    }
}
```

Application of BFS

→ Shortest path in terms of Edge

→ Rotten Oranges.

→ Connected Component
  → No of islands. 1
  → No of island 2

→ Alice and bricks.

~~~~~~~~~~~~~~

Rotten Oranges

~~~~~~~~~~~~~~

You are given an m x n grid where each cell can have one of three values:

0 -> representing an empty cell,

1 -> representing a fresh orange, or

2 -> representing a rotten orange.

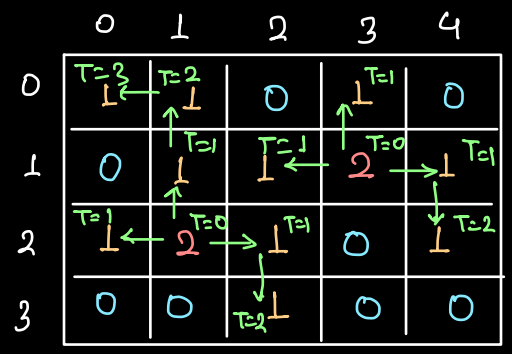
Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange.

If this is impossible, return -1.

Approach: Multi Source BFS

Radially a rotted orge is Rotten fresh oreges.



T=0 → (1,3), (2,1)

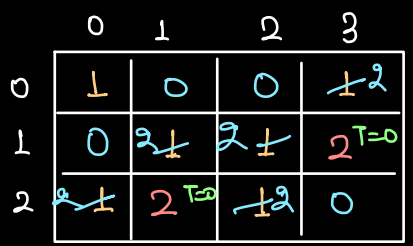
T=1 → (0,3), (1,2), (1,4), (1,1), (2,0), (2,2)

T=2 → (2,4), (0,1), (3,2)

T=3 → (0,0)

T=4 → ✗

Ans: 3 minutes



Starting point

(1,3,0), (2,1,0) | (0,3,1), (1,2,1) | (1,1,1) (2,0,1) (2,2,1)

T=0 [1,3 → 0 ; 2,1 → 0]

T=1 [0,3 → 1 ; 1,2 → 1 ; 1,1 → 1 ; 2,0 → 1 ; 2,2 → 1]

All orgs are not rotted → -1

Remove
work
add unvisited nbr
→ marks

```
class {
    int x;
    int y;
    int t;
}
```

Queue<Pair> que = new ArrayDeque<>();

Code :

```java
public static class Pair {
    int i;
    int j;
    int t;
    public Pair(int i, int j, int t) {
        this.i = i;
        this.j = j;
        this.t = t;
    }
}

public static int[] xdir = {-1, 0, 1, 0};
public static int[] ydir = {0, -1, 0, 1};

public static int rottenOranges(int[][] arr) {
    /*
        Make a queue and add all oranges which are already rotted
        if they are already rotted, time for them is T=0
        make sure while adding rotted oranges in queue
        take the count of total oranges as well
    */
    int n = arr.length;
    int m = arr[0].length;

    Queue<Pair> qu = new ArrayDeque<>();

    int count = 0;
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < m; j++) {
            if(arr[i][j] == 2) {
                // (i,j) location is already rotted
                qu.add(new Pair(i,j,0));
            }

            // count the total orange as well
            // arr[i][j] == 1 || arr[i][j] == 2 => arr[i][j] != 0
            if(arr[i][j] != 0) {
                // rotted + fresh
                count++;
            }
        }
    }

    // Apply BFS Algorithm on que
    // multiple starting point available for BFS
    int time = 0;
    while(qu.size() > 0) {
        // remove
        Pair rem = qu.remove();
        // one orange rotted, decrease the count from total orange
        count--;
        // work  -> maximise the time
        time = rem.t;
        // add unvisited neighbour
        // iteration on 4 direction -> top, left, down, right
        for(int d = 0; d < 4; d++)  {
            int r = rem.i + xdir[d];
            int c = rem.j + ydir[d];

            if(r >= 0 && r < n && c >= 0 && c < m && arr[r][c] == 1) {
                // marking
                arr[r][c] = 2;
                // adding in que
                qu.add(new Pair(r,c, rem.t + 1));
            }
        }
    }
    if(count == 0) {
        return time;
    } else {
        return -1;
    }
}
```

Row and colm cont

→ Apply BFS Algo

→ to calculate total orgen

→ If Rotted add with t=0

→ If it is oraye add it in count

helping in making sure that location is valid

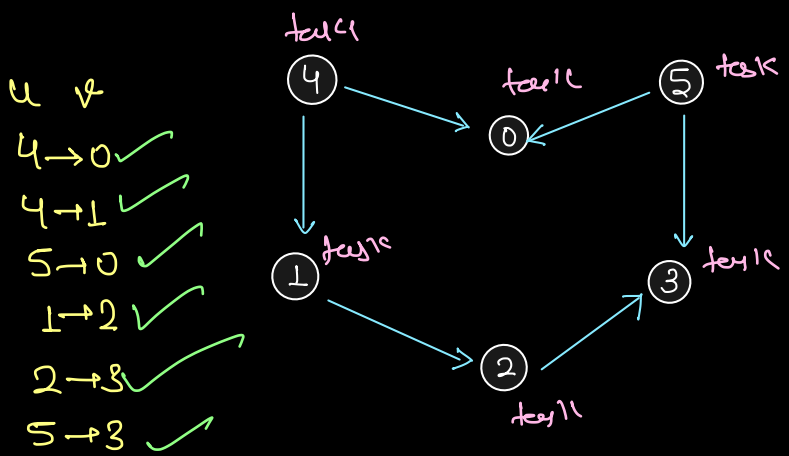
helping in making sure that added orgye is fresh.

~~~~~~~~~~~~~~~~

Topological Sort

~~~~~~~~~~~~~~~~

Linear Order of the graph such that for every directed edge u to v, vertex u comes before v in order. It is Applicable for (Directed Acyclic Graph -> DAG)



NOTE: for single Graph, more than 1 topological order is possible.

3 1 0 2 5 4 ✗

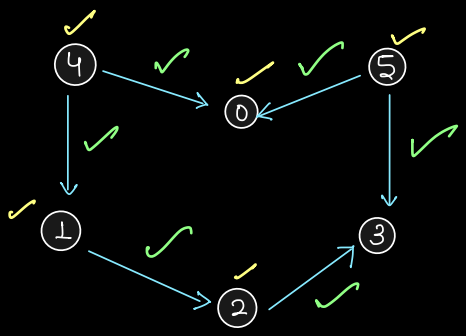
4 0 5 3 1 2 ✗

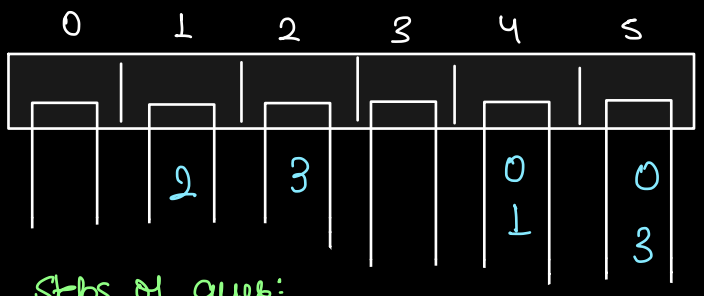
4 5 0 2 1 3 ✗

4 5 0 1 2 3 ✓

5 4 0 1 2 3 ✓

Kahn's Algorithm:



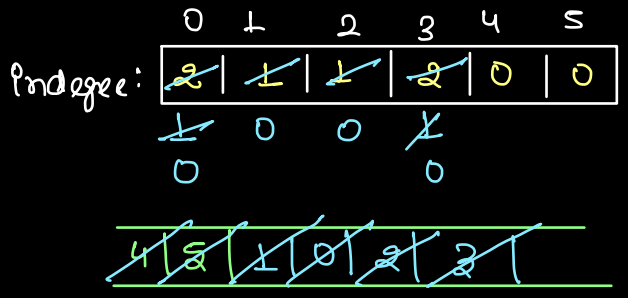


Steps of ques:

→ Remove front elemet of ques
→ print removed elemt → topological order
→ Decrease indegree of nbr of removed elem while decreasing, if indegree becomes 0 add it in ques.

Kahn's Algo:

① Create an indegree array.
② Add vertex with 0 indegree in ques.
③ Traverse and solve ques.

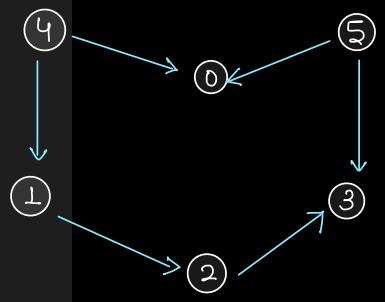


O/p: 4 5 1 0 2 3

⇒ Topological Sort

```java
// topological Sort  -> Kahn's Algorithm
public static void topologicalSort(ArrayList<ArrayList<Integer>> graph) {
    int n = graph.size();
    // 1. prepare indegree array
    int[] indegree = new int[n];
    for(int v = 0; v < n; v++) {
        // and edge avialable from v->nbr
        // indgeree of nbr increased by 1
        for(int nbr : graph.get(v)) {
            indegree[nbr]++;
        }
    }

    // 2. add 0 indegree vertex in queue and process it
    Queue<Integer> qu = new ArrayDeque<>();
    for(int v = 0; v < n; v++) {
        if(indegree[v] == 0) {
            qu.add(v);
        }
    }

    // 3. Queue Processing
    while(qu.size() > 0) {
        // 3.1 Remove front element from queue
        int rem = qu.remove();
        // 3.2 Print Removed vertex (topological order)
        System.out.print(rem + " ");
        // 3.3 Iterate on nbr of removed vertex and decrease
        // indegree of nbr, if indegree becomes 0, add it in que
        for(int nbr : graph.get(rem)) {
            indegree[nbr]--;
            if(indegree[nbr] == 0) {
                qu.add(nbr);
            }
        }
    }
}
```

n=6



	0	1	2	3	4	5
Indegree:	~~2~~ 1	~~1~~ 0	~~1~~ 0	~~2~~ 1	0	0
	1			0		
	0					

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

4 5 1 0 2 3 ⇒ topological Sort

```java
public static void addEdge(ArrayList<ArrayList<Integer>> graph,
    int u, int v) {
    graph.get(u).add(v); // single edge -> directed graph
}

public static void demo() {
    int n = 6;
    ArrayList<ArrayList<Integer>> graph = new ArrayList<>();
    for(int v = 0; v < n; v++) {
        graph.add(new ArrayList<>());
    }

    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 5, 0);
    addEdge(graph, 1, 2);
    addEdge(graph, 2, 3);
    addEdge(graph, 5, 3);
    topologicalSort(graph);
}
```

graph properties

u → v

u is depending on v

order of execution:

Reverse of topological Sort

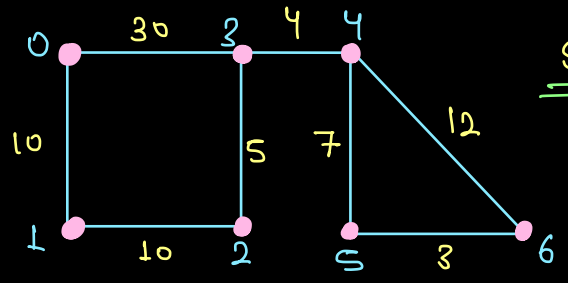
Dijkstra's Algorithm

Single Source shortest path in terms of weight.

Wig BFS ← Shortest path from 0-3 (edge)
0–3 → weight = 30

Dijkstra's ← Shortest path from 0-3 (weight)
0–1–2–3 → weight = 25

Example:



Src = 0

Shortest path from '0'

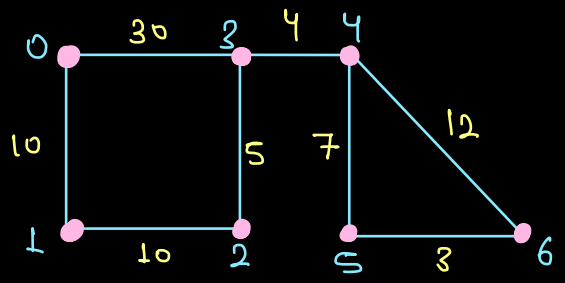
0	1	2	3	4	5	6
0	10	20	25	29	36	39

Steps of Dijkstra's Algorithm:

- Make a Priority Queue
- Add Pair(have vtx, wsf) of Source Point in it
- Make a visited array
- Process the Priority Queue until it is empty
  - Remove the pair
  - If the pair is already marked -> continue
  - Otherwise, mark it
  - Iterate on all nbrs of removed vtx,
    If not visited, make a pair and add them in PQ

wsf => weight so far

mark Diff b/w BFS & Dijkstra



Source = 0

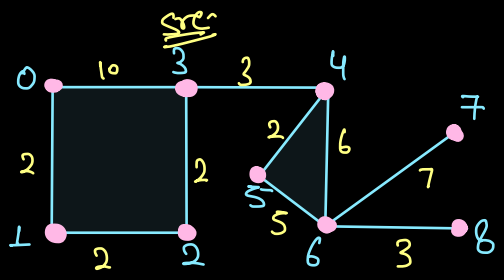
0	1	2	3	4	5	6
0	10	20	25	29	36	39

* Rem
* work
* Add unvisited nbr

Priority Queue: MinPQ → Shortest path

0–0
1–10
3–30 * Already removed → Already visited through better wsf
2–20
3–25
4–29
5–36, 6–39
6–44 * Already marked → extra

Priority Queue →



src = 3

nbr_wsf

~~3-0~~ ~~6-10~~
~~0-10~~ ~~7-16~~
~~2-2~~ ~~8-12~~
~~4-3~~
~~1-4~~
~~5-5~~
~~6-9~~
~~0-6~~

vis: T T T T T T T T T

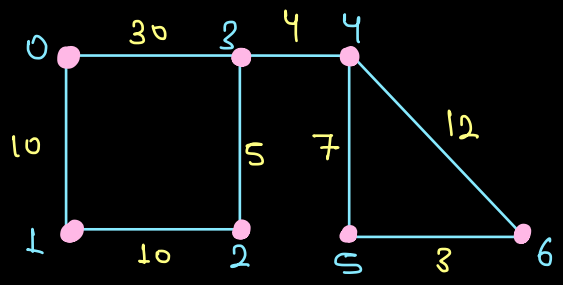
Arr:	6	4	2	0	3	5	9	16	12
	0	1	2	3	4	5	6	7	8

class Pair {
    int vtx;
    int wsf;
}

NOTE: Single source multiple destination min PQ
shortest path Algo ⇒ Dijkstra's Algo.

```java
// Dijkstras Algorithm
public static class DPair {
    int vtx;
    int wsf;
    String psf;

    public DPair(int vtx, int wsf, String psf) {
        this.vtx = vtx;
        this.wsf = wsf;
        this.psf = psf;
    }
}

public static void dijkstrasAlgo(ArrayList<ArrayList<Pair>> graph, int src) {
    int n = graph.size();
    // 1. Priority Queue
    PriorityQueue<DPair> pq = new PriorityQueue<>(new Comparator<DPair>(){
        public int compare(DPair a, DPair b) {
            return a.wsf - b.wsf;
        }
    });
    pq.add(new DPair(src, 0, src + ""));
    boolean[] vis = new boolean[n];
    // 2. Process PQ
    while(pq.size() > 0) {
        // 2.1 remove
        DPair rem = pq.remove();
        // 2.2 mark, if already marked continue
        if(vis[rem.vtx] == true) {
            // if already marked, skip this iteration
            continue;
        }
        // if not marked, mark it
        vis[rem.vtx] = true;
        // 2.3 work
        System.out.println("[" + rem.vtx + "] -> " + rem.psf + " @ " + rem.wsf);
        // 2.4 Add unvisited neighbour -> no need to mark
        for(Pair p : graph.get(rem.vtx)) {
            int nbr = p.nbr;

            if(vis[nbr] == false) {
                pq.add(new DPair(nbr, rem.wsf + p.wt, rem.psf + "-" + nbr));
            }
        }
    }
}
```

custom Comparator

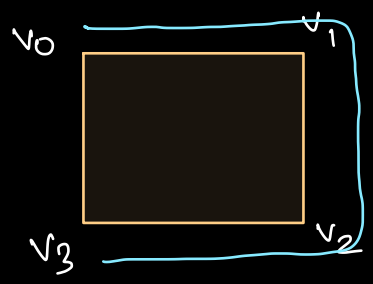


~~~~~~~~~~~~~~

Cycle in graphs

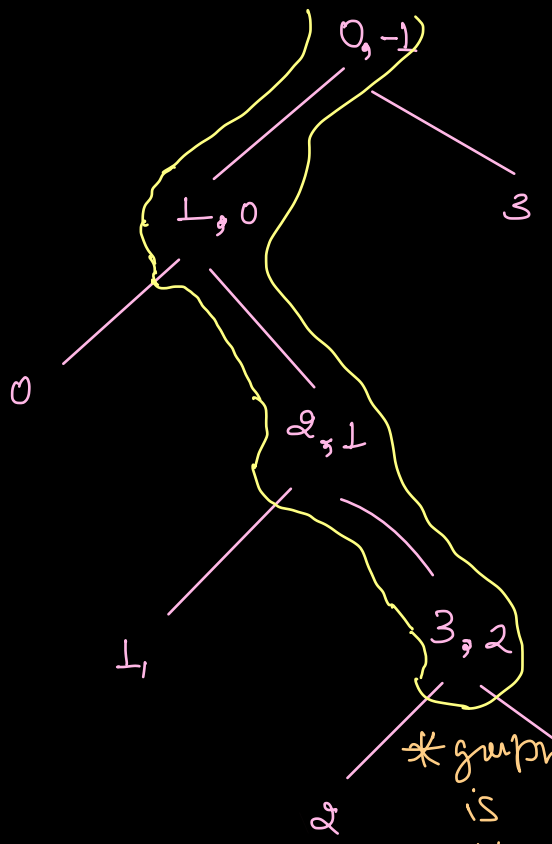
~~~~~~~~~~~~~~

For undirected graph -> Using DFS



DFS

vrtx, parent



0 is already visited and also not parent of 3

* graph is cyclic

0 → Different path

→ Cyclic graph

if vertex is already visited but it is not my parent vertex → then it is cycle.

Code:

```java
public class Solution {
    public boolean dfs(int u, int par, ArrayList<ArrayList<Integer>> adj, boolean[] vis) {
        vis[u] = true; // Marking the current vertex as visited.
        // Iterating over all the adjacent vertices.
        for (int v : adj.get(u)) {
            if (v == par) continue; // If the adjacent vertex is the parent, skip it.
            // If the current vertex is visited, we return true; else,
            // we call the function recursively to detect the cycle.
            if (vis[v] == true) return true;
            if (dfs(v, u, adj, vis) == true) return true;
        }
        return false; // No cycle found in the current path.
    }

    public boolean isCycle(int V, ArrayList<ArrayList<Integer>> adj) {
        boolean[] visited = new boolean[V]; // Tracking visited vertices.
        // Checking each vertex for a cycle.
        for (int i = 0; i < V; i++) {
            if (visited[i] == false) { // If vertex is not visited, we call dfs to detect cycle.
                boolean cycle = dfs(i, -1, adj, visited);
                if (cycle) return true; // If a cycle is detected, return true.
            }
        }
        return false; // No cycle found in the graph.
    }
}
```

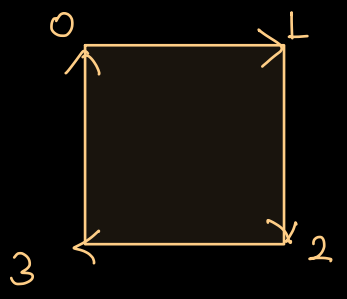
For directed graph -> Using Kahn's Algorithm [using kahn's Algo]

Topological sort

While processing queue of topological sort problem.

Count of processed vertex $\neq$ total vertex

$\therefore$ cycle is available in graph $\Rightarrow$ Directed graph

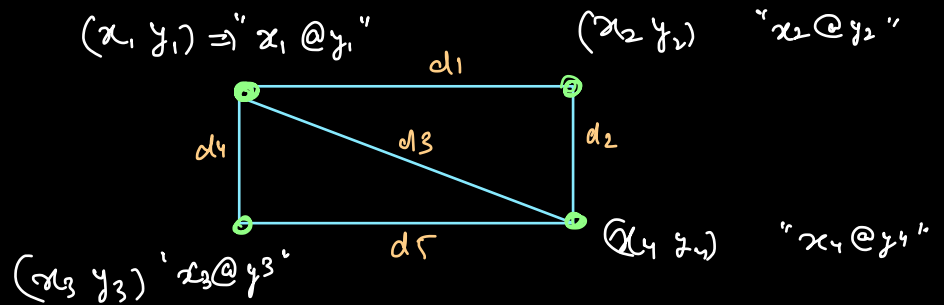


$\rightarrow$ Because of Dependency, we can't even start execution.

$\rightarrow$ Indegree never be 0 for all vertex

$\Rightarrow$ Cycle is there

$\rightarrow$ Directed graph is cyclic



```java
// cycle detection in directed graph using indegree
public static void isCyclic(ArrayList<ArrayList<Integer>> graph) {
    int n = graph.size();
    // 1. prepare indegree array
    int[] indegree = new int[n];
    for(int v = 0; v < n; v++) {
        // and edge avialable from v->nbr
        // indgeree of nbr increased by 1
        for(int nbr : graph.get(v)) {
            indegree[nbr]++;
        }
    }

    // 2. add 0 indegree vertex in queue and process it
    Queue<Integer> qu = new ArrayDeque<>();
    for(int v = 0; v < n; v++) {
        if(indegree[v] == 0) {
            qu.add(v);
        }
    }
```

```java
    // 3. Queue Processing
    int count = 0;
    while(qu.size() > 0) {
        // 3.1 Remove front element from queue
        int rem = qu.remove();
        // 3.2 increase count variable and remove vertex
        count++;
        // 3.3 Iterate on nbr of removed vertex and decrease
        // indegree of nbr, if indegree becomes 0, add it in que
        for(int nbr : graph.get(rem)) {
            indegree[nbr]--;
            if(indegree[nbr] == 0) {
                qu.add(nbr);
            }
        }
    }
    if(count != n) {
        System.out.println("Cyclic Graph");
    } else {
        System.out.println("Acyclic Graph");
    }
}
```