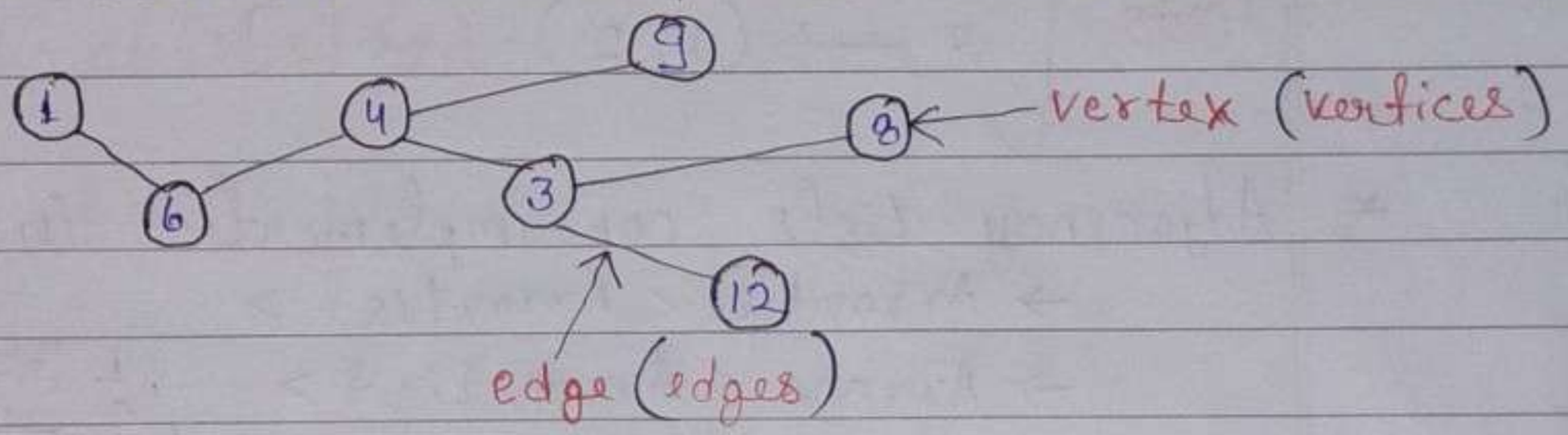
Enroll
Page No: 492
Date: / /

GRAPHS

* A Graph is a non-linear data structure.
* Graph is a network of nodes.
* Graph consisting of vertices (also called nodes) & edges.

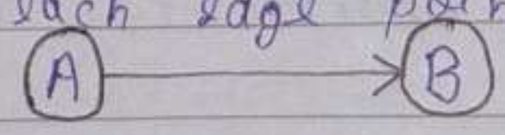
Vertices: The fundamental units of a graph where the data is stored.

Edges: The connections between pairs of vertices. A edge may be directed (one-way) or undirected (two-way).

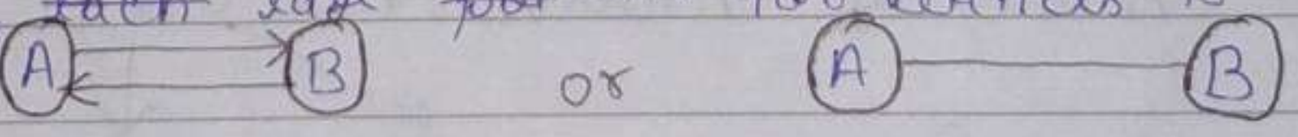


★ Types Of Graphs (Based on Edge)

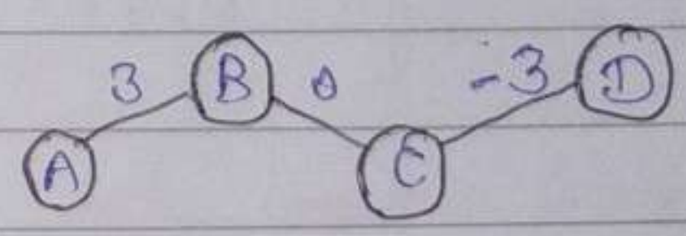
i) Uni-Directional or Directed Edge - edges have direction, each edge points from one vertex to another.



ii) Bi-Directional or UnDirected Edge - edges have no direction, ~~each~~ edge ~~per~~ between two vertices is bidirectional.



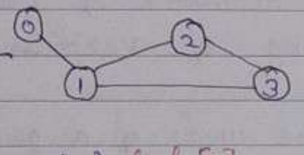
iii) Weighted Graph - each edge has an associated weight, it can be positive or negative.



iv) UnWeighted Graph - Graph where there is no weight or have equal of all edges.

★ GRAPH REPRESENTATION (storing a graph)

i) Adjacency Lists : list of lists.



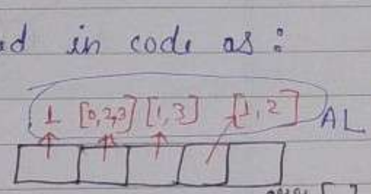
storing as adjacency list

V=4 | undirected
E=4 | unweighted

List of lists
- 0 → (1) list [0]
- 1 → (0, 2, 3) list [1]
- 2 → (1, 3) list [2]
- 3 → (1, 2) list [3]

Finding neighbour → Adjacency list → O(K)

* Adjacency Lists can implemented in code as :
  - → ArrayList < ArrayList >
  - → Array < ArrayList >



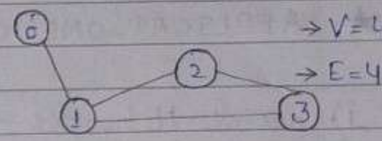
  - → HashMap < int , lists >
    - ↓ vertex
    - ↓ values

* Adjacency lists are optimized & doesn't store extra information.

* How data are stored inside Adjacency List :-
  Array of ArrayList < Edge >
  - Source (जहाँ से जाना है)
  - Destination (जहाँ तक जाना है)
  - Weight (अगर weight है तो store करा लो or weight नहीं है तो या तो store मत कराओ या equal num store करा दो)

⇒ We will store weight two times in bidirectional.

ii) Adjacency Matrix :



→ V=4
→ E=4

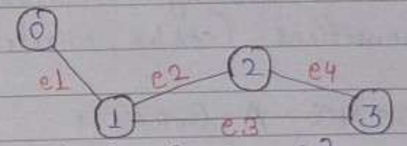
* V × V Matrix

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

* If (i, j) is 0 then no edge exist between i & j.
* If (i, j) is 1 then there is an edge exist between i & j.

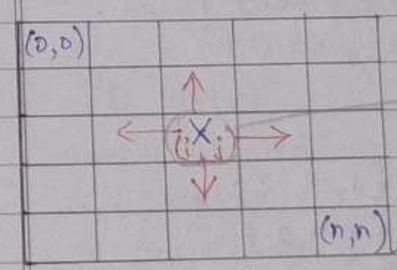
* There is disadvantage using Adjacency Matrix, the time complexity becomes Linear → O(V).

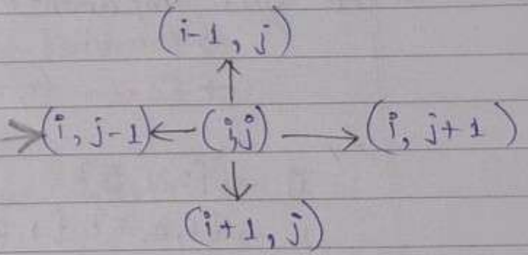
iii) Edge List :



* Edges = {{0, 1}, {1, 2}, {1, 3}, {2, 3}}
  (e1, e2, e3, e4)
* We can also store weight i.e., ⇒ [{0, 1, -1}, {1, 2, 5}]---
  (-1 : weight, 5 : weight)

iv) Implicit Graph :



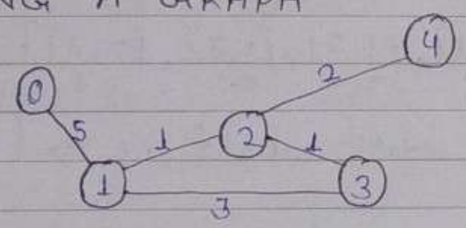


* We have to imagine 2D-Matrix into implicit Graph.
* Implicit graphs are not explicitly represented.

★ APPLICATIONS OF GRAPHS

i) Social Networks – Nodes represents people & edges represent friendships or interactions.
ii) Web Search – Web pages are nodes & hyperlinks between them are edges.
iii) Maps or GPS – Locations are nodes & roads are edges with weights (distances or times).
iv) Recommendation System (Amazon, Netflix) – Users & products are nodes & edges represent interactions (e.g., a user buying a product or rating a movie).
v) Routers or Routing – Routers & switches are nodes & connections (cables, wireless links) are edges.

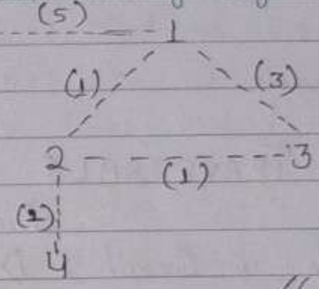
★ CREATING A GRAPH



* We will implement above graph using Adjacency list.
  → Arraylist <Edge> graph[]
  → Edge = (src, dest, weight)

i.e.,
0 → {0, 1, 5}
1 → {1, 0, 5} {1, 2, 1} {1, 3, 3}
2 → {2, 1, 1} {2, 3, 1} {2, 4, 2}
3 → {3, 1, 3} {3, 2, 1}
4 → {4, 2, 2}

⇒
```
static class Edge {
    int src;
    int dest;
    int wt;
    public Edge (int s, int d, int w) {
        this.src = s;
        this.dest = d;
        this.wt = w;
    }
}
public static void main (String args[]) {
/*
```




```
*/                                  //null → empty arraylist
int V=5;
Arraylist <Edge>[] graph = new Arraylist[V];
for (int i=0; i<V; i++) {
    graph[i] = new Arraylist<>();
}
//0 vertex
graph[0].add (new Edge (0,1,5));
//1 vertex
graph[1].add (new Edge (1, 0, 5));
graph[1].add (new Edge (1, 2, 1));
graph[1].add (new Edge (1, 3, 3));
//2 vertex
graph[2].add (new Edge (2, 1, 1));
graph[2].add (new Edge (2, 3, 1));
graph[2].add (new Edge (2, 4, 2));
```

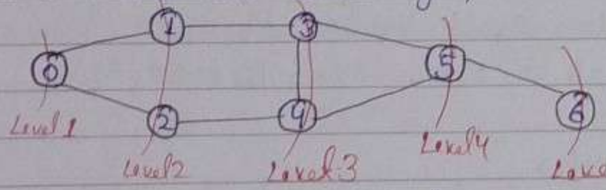
```
//3 vertex
graph[3].add(new Edge(3,1,3));
graph[3].add(new Edge(3,2,1));
//4 vertex
graph[4].add(new Edge(4,2,2));
//2's neighbors
for(int i=0; i<graph[2].size(); i++){
    Edge e = graph[2].get(i); //src, dest, wt
    System.out.println(e.dest);
}
}
```

Output: 1
3
4

★ GRAPH TRAVERSALS

There are two types of traversals in Graph :- i) Breadth First Search (BFS)
ii) Depth First Search (DFS)

I. BFS → We can starting point (src) by own.



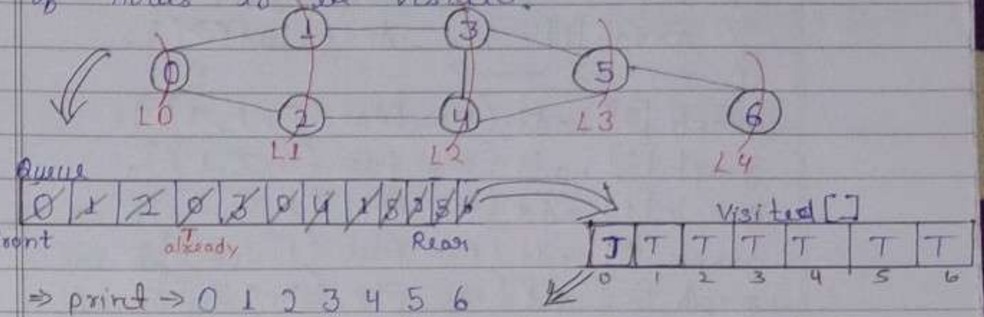
//O(V+E)

#Go to immediate neighbors first.

⇒ "Breadth-First Search is a graph traversal algorithm that explores the nodes of a graph level by level starting from a given source node & visiting all its neighbors before moving to the next level of nodes." ~~If nodes queue data structure to keep too~~

If 0 is also a neighbour of 1 in graph & it should also be added in Queue again. (But it will be T in queue so this will not change ans)

★ It is achieved by using a queue to keep track of nodes to be visited.



⇒ print → 0 1 2 3 4 5 6

★ We have to check that specific node is visited or not so we make an array of boolean types that are as false at starting.

★ Graphs are in cycle so we take an extra array & Binary Tree didn't have cycles.

★ हमें Visited को Print तभी करवाना है जब वो पहले Unvisited हों तभी Visited करके Print करवाना है वरना Visited अगर पहले से है तो Print मत करवाना।

⇒
```
static class Edge {
    int src;
    int dest;
    int wt;
    public Edge(int s, int d, int w) {
        this.src = s;
        this.dest = d;
        this.wt = w;
    }
}
```

//Next page

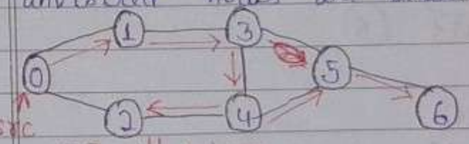
```
static void createGraph(ArrayList<Edge> graph[]){
    for(int i=0; i<graph.length; i++){
        graph[i] = new ArrayList<>();
    }
    graph[0].add(new Edge(0,1,1));
    graph[0].add(new Edge(0,2,1));
    graph[1].add(new Edge(1,0,1));
    graph[1].add(new Edge(1,3,1));
    graph[2].add(new Edge(2,0,1));
    graph[2].add(new Edge(2,4,1));
    graph[3].add(new Edge(3,1,1));
    graph[3].add(new Edge(3,4,1));
    graph[3].add(new Edge(3,5,1));
    graph[4].add(new Edge(4,2,1));
    graph[4].add(new Edge(4,3,1));
    graph[4].add(new Edge(4,5,1));
    graph[5].add(new Edge(5,3,1));
    graph[5].add(new Edge(5,4,1));
    graph[5].add(new Edge(5,6,1));
    graph[6].add(new Edge(6,5,1));
}

public static void bfs(ArrayList<Edge>[] graph){
    Queue<Integer> q = new LinkedList<>();
    boolean vis[] = new boolean[graph.length];
    q.add(0); //source = 0
    while(!q.isEmpty()){
        int curr = q.remove();
        if(!vis[curr]) //visit curr
            System.out.print(curr+" ");
            vis[curr] = true;
            for(int i=0; i<graph[curr].size(); i++){
                Edge e = graph[curr].get(i);
                q.add(e.dest);
```



```
public static void main(String args[]){
/*      1---3
       /    |\
      0     1 5--6
       \    |/
        2---4
*/
    int V=7;
    ArrayList<Edge> graph[] = new ArrayList[V];
    createGraph(graph);
    bfs(graph);
}
```

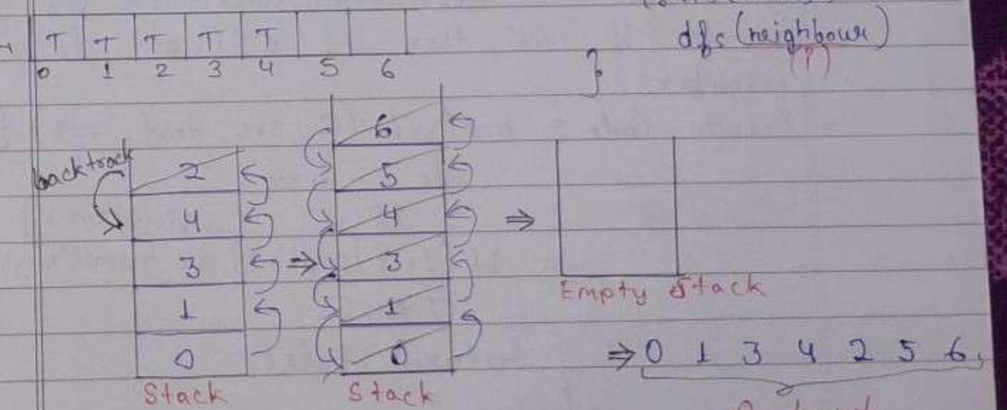
II. DFS → Keep going to the 1st neighbour.

"Depth First Search explores each node by moving to an adjacent unvisited node, diving deep into the graph, & only backtracks when no further unvisited nodes are available in the current path."



Recursive function

```
S1: dfs(graph, curr){
S2:     curr → visit → print vis[curr]=T
S3:     for(int i=0; i<to.k)
            (!vis[curr])
                dfs(neighbour)
        }
```

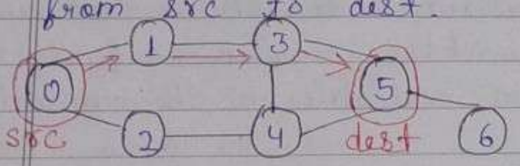


```
//O(V+E)
⇒ public static void dfs(ArrayList<Edge>[] graph, int
                          curr, boolean vis[]){
    //visit
    System.out.print(curr + " ");
    vis[curr] = true;
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(!vis[e.dist]){
            dfs(graph, e.dest, vis);
        }
    }
}
```

Input: dfs(graph, 0, new boolean[V]);
Output: 0 1 3 4 2 5 6

★ Has Path ?

- For given src & dest, tell if a path exists from src to dest.



* src = 0,
* dest = 5

* We will use Dfs.
* Basically, src will ask to its neighbors that you know the destination (dest) ; if 'yes' then go & if 'no' then it will ask to its neighbors.
* Pseudo Code : HasPath (G, src, dest, vis) {

```
    src == dest
        TRUE → visit[curr]
if(!vis[neighbor] && haspath(neighbor, dest))
    TRUE
    return false.
}
```

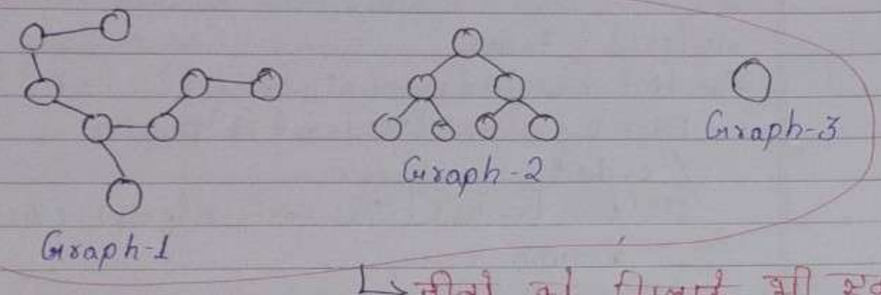


```
⇒ public static boolean hasPath(ArrayList<Edge>[] graph, int src, int dest, boolean vis[]){
    if(src == dest){
        return true;
    }
    vis[src] = true;
    for(int i=0; i<graph[src].size(); i++){
        Edge e = graph[src].get(i);
        //e.dest = neighbour
        if(!vis[e.dest] && hasPath(graph, e.dest, dest, vis)){
            return true;
        }
    }
    return false;
}
```

Input: Syso(hasPath(graph, 0, 5, new boolean[V]));
Output: true

GRAPHS - II

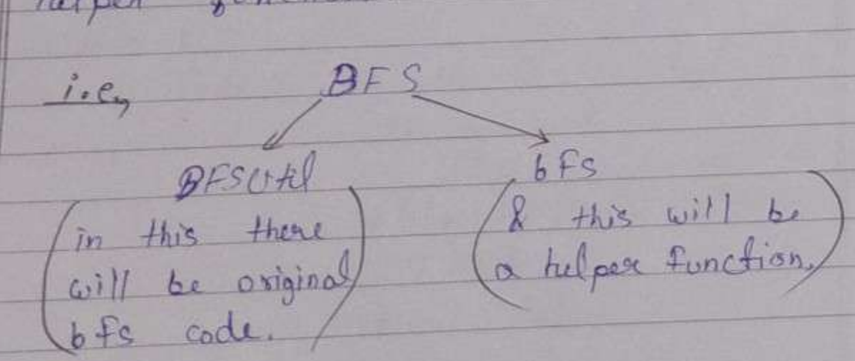
★ CONNECTED COMPONENTS



↳ तीनों को मिलाके भी एक Single graph बनेगा।
(Graph)

* Above Graph (G1, G2, G3) are not connected, but they all three are ~~combined so it's~~ together are a single graph but we cannot traverse by DFS & BFS.
* We cannot use DFS & BFS directly becoz the G1, G2, G3 are not connected with eachother.
* We can use DFS or BFS with a extra helper function.

i.e,



⇒ //BFS METHOD

```
public static void bfs(ArrayList<Edge>[] graph){
    boolean vis[] = new boolean[graph.length];
    for(int i=0; i<graph.length; i++){
        if(!vis[i]){
            bfsUtil(graph, vis);
        }
    }
}
public static void bfsUtil(ArrayList<Edge>[] graph,
                           boolean vis[]){
    Queue<Integer> q = new LinkedList<>();
    q.add(0); //source = 0
    while(!q.isEmpty()){
        int curr = q.remove();
        if(!vis[curr]){ //visit curr
            System.out.print(curr+" ");
            vis[curr] = true;
            for(int i=0; i<graph[curr].size(); i++){
                Edge e = graph[curr].get(i);
                q.add(e.dest);
            }
        }
    }
}
```

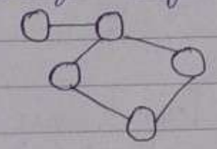
⇒ //DFS METHOD

```
public static void dfs(ArrayList<Edge>[] graph){
    boolean vis[] = new boolean[graph.length];
    for(int i=0; i<graph.length; i++){
        dfsUtil(graph, i, vis);
    }
}
```

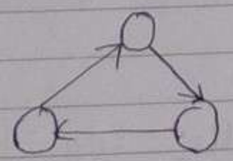
```
public static void dfsUtil(ArrayList<Edge>[] graph,
                           int curr, boolean vis[]){
    System.out.print(curr + " ");
    vis[curr] = true;
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(!vis[e.dest]){
            dfsUtil(graph, e.dest, vis);
        }
    }
}
```

★ CYCLE IN GRAPHS

* In Undirected Graphs, we can detect cycle using algorithms :- i) DFS
  ii) BFS
  iii) DSU (Disjoint Set Union)

Ex→ 

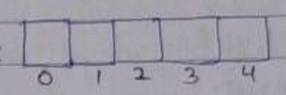
* In Directed Graphs, we can detect cycle using algorithms :- i) DFS
  ii) BFS
  iii) Topological Sort (Kahn's Algorithm)

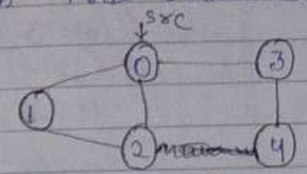
Ex→ 

Cycle detection in a graph depend on whether the graph is directed or undirected.

★ CYCLE DETECTION IN UNDIRECTED GRAPH (We will use DFS)

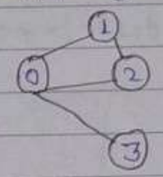
* We will make boolean type function that if given graph is Cycle so return true otherwise false.

Approach: vis[] : 



* ~~किसी भी~~ a node के पास तीन तरीके के neighbor हो ~~सकते हैं~~ :-

अगर किसी Node के neighbors पहले से ही visited हो तो ऐसा हो सकता है कि cycle exist हो।



Case1 : vis[neighbor] ✓ parent ✗ → TRUE

Case2 : vis[neighbor] ✓ parent ✓ → continue; (it is not necessary there will be cycle)

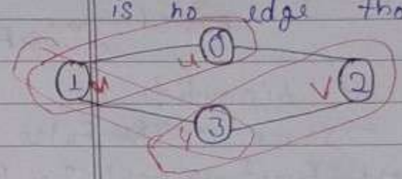
Case3 : vis[neighbor] ✗ parent ✗ → normal DFS call (visit) → cycle → TRUE

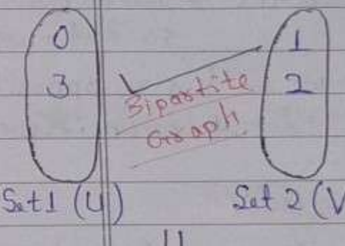
⇒ // no need for weight (wt) variable.

```
public static boolean detectCycle(ArrayList<Edge>[] graph){
    boolean vis[] = new boolean[graph.length];
    for(int i=0; i<graph.length; i++){
        if(!vis[i]){
            if(detectCycleUtil(graph, vis, i, -1)){
                return true; //cycle exists in one of the parts
            }
        }
    }
    return false;
}
```

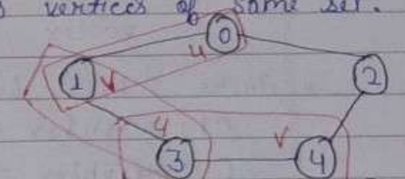
```
public static boolean detectCycleUtil (ArrayList<Edge>[] graph, boolean vis[], int curr, int par) {
    vis[curr] = true;
    for (int i=0; i<graph[curr].size(); i++) {
        Edge e = graph[curr].get(i);
        //case-3
        if (!vis[e.dest]) {
            if (detectCycleUtil(graph, vis, e.dest, curr)) {
                return true;
            }
        }
        //case-1
        else if (vis[e.dest] && e.dest != par) {
            return true;
        }
    }
    //case 2 -> do nothing -> continue,
    return false;
}
public static void main (String args[]) {
    /*      0 - - - 3
          /   |     |
        1     1     1
          \   |     |
            \ 1     4
              2
    */
    int V=5;
    ArrayList<Edge> graph[] = new ArrayList[V];
    create Graph (graph);
    System.out.println (detectCycle (graph));
}
```

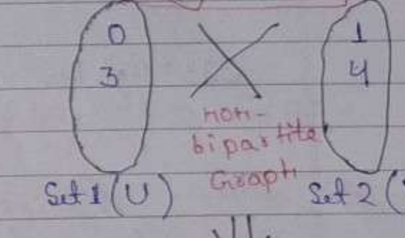

★ BIPARTITE GRAPH – A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U & V such that every edge (u,v) either connects a vertex from U to V or a vertex from U to V. In other words, for every edge (u, v), either u belongs to U & v to V, or u belongs to V & v to U. We can also say that there is no edge that connects vertices of same set.





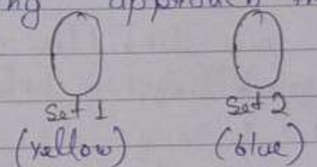
⇓ Return TRUE



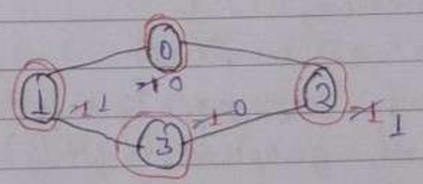


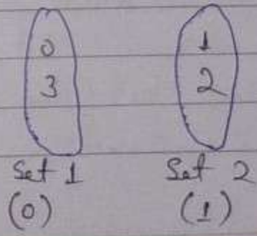
⇓ Return FALSE

We will use BFS approach & also coloring approach in graphs.

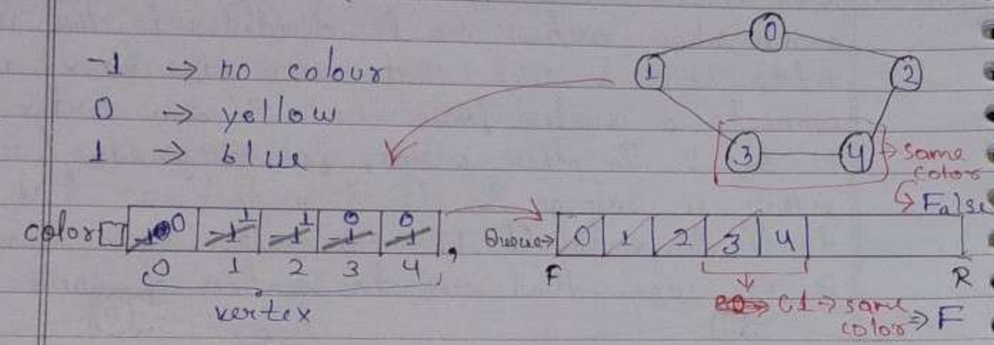


Ex →





* BFS Based (Coloring)



* There will be 3 case of bipartite i.e.;
  C1: neighbor → coloring → same → ~~TRU~~ False
  C2: neighbor → coloring → different → maybe T or F so continue;
  C3: neighbor → coloring X → we will give opposite color & push in queue

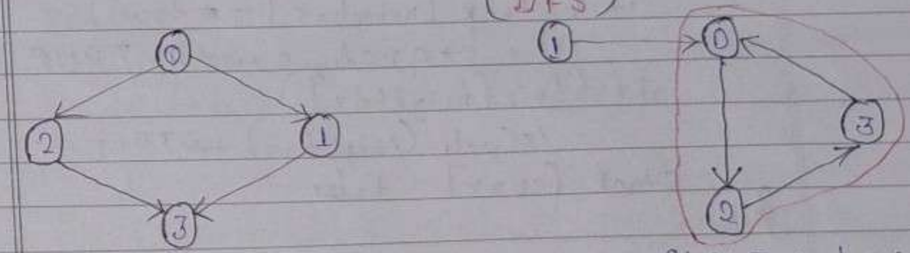
⇒ // Time Complexity → O(V)

```
public static boolean isBipartite(ArrayList<Edge>[] graph){
    int col[] = new int[graph.length];
    for(int i=0; i<col.length; i++){
        col[i] = -1; //no color
    }
    Queue<Integer> q = new LinkedList<>();
    for(int i=0; i<graph.length; i++){
        if(col[i] == -1){
            q.add(i);
            col[i] = 0;
            while(!q.isEmpty()){
                int curr = q.remove();
                for(int j=0; j<graph[curr].size(); j++){
                    Edge e = graph[curr].get(j);
```



```
                    if(col[e.dest] == -1){
                        int nextCol = col[curr] == 0 ? 1 : 0;
                        col[e.dest] = nextCol;
                        q.add(e.dest);
                    } else if(col[e.dest] == col[curr]){
                        return false;
                    }
                }
            }
        }
    }
    return true;
}
```

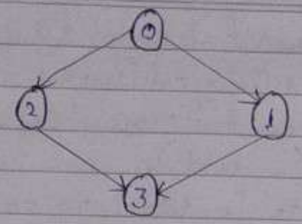
★ CYCLE DETECTION IN DIRECTED GRAPH (DFS)



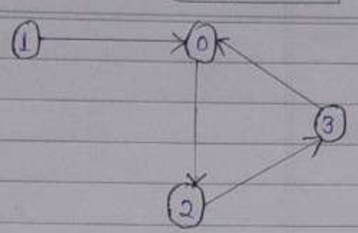
* Why Undirected DFS approach fails?
→ In directed graphs, backtracking may re-visit nodes due to the direction of edges, even if no cycle exists.

Ex: Node 1 → 0 → 2 → 3 has no cycle, but applying the undirected graph method might incorrectly flag a cycle.

Approach:



NO CYCLE



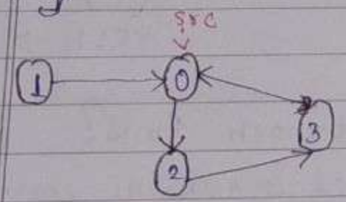
CYCLE

* We will use DFS with some modification.

Stack (boolean) ⇓ work as vis[]. → Recursion Stack (explicit) ⇓ यह हमें बनाना है।

Pseudo Code:

```
dfs (curr) {
  vis[curr] = true
  stack [curr] = true
  for (all neighbors)
    if (stack [neigbor] == true)
          └→ cycle exist → TRUE
    if (!vis [neighbor])
        isCycle (neighbor) → TRUE
  stack [curr] = false
}
```



T		T	T	vis[]
0	1	2	3	

##Now 3 will check की उसकी कोई neighbor stack में पहले से है या नहीं | * if YES → Cycle * if No → No cycle

3	T	T
2	T	T
1		T
0	T	

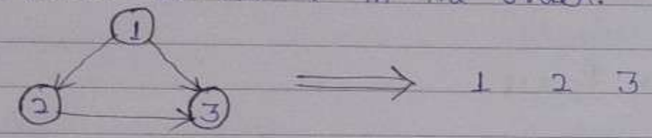
Stack ⇒ TRUE


```
⇒ public static boolean isCycle(ArrayList<Edge>[] graph){
    boolean vis[] = new boolean[graph.length];
    boolean stack[] = new boolean[graph.length];
    for(int i=0; i<graph.length; i++){
      if(!vis[i]){
        if(isCycleUtil(graph, i, vis, stack)){
          return true;
        }
      }
    }
    return false;
  }
  public static boolean isCycleUtil(ArrayList<Edge>[] graph, int curr, boolean vis[], boolean stack[]){
    vis[curr] = true;
    stack[curr] = true;
    for(int i=0; i<graph[curr].size(); i++){
      Edge e = graph[curr].get(i);
      if(stack[e.dest]){ //cycle
        return true;
      }
      if(!vis[e.dest] && isCycleUtil(graph, e.dest, vis, stack)){
        return true;
      }
    }
    stack[curr] = false;
    return false;
  }
  public static void main(String args[]){
    int V = 4;
    ArrayList<Edge> graph[] = new ArrayList[V];
    createGraph(graph);
    Sys o(isCycle(graph));
  }
```

★ TOPOLOGICAL SORTING -

* Topological sorting is used only for DAGs.
* Directed Acyclic Graph (DAG) is directed graph with no cycles.

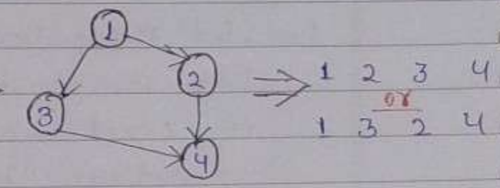
⇒ "Topological sorting is a linear order of vertices such that every directed edge $u \to v$, the vertex u comes before v in the order."

Ex→ 

⟹ 1 2 3

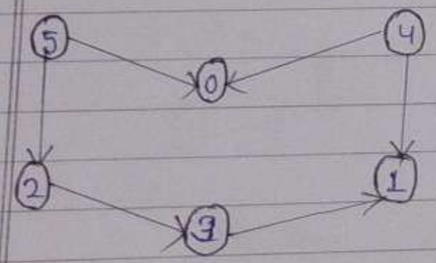
* Topological sorting can be in any order.
→ Dependency

Ex→ Action 1 - boil water
Action 2 - add masala
Action 3 - add maggie
Action 4 - serve maggie



⇒ 1 2 3 4
or
1 3 2 4

* We will use DFS for Topological sorting.



⇒ Possible ans
① 5 4 2 3 1 0
or
② 4 5 2 3 1 0

Approach: modified DFS
⇓
We will use Stack → to track which node come first.

```
⇒ public static void topSort (ArrayList<Edge>[] graph) {
    boolean vis[] = new boolean [graph.length];
    Stack<Integer> s = new Stack<>();
    for (int i=0; i<graph.length; i++) {
      if (!vis[i]){
        topSortUtil (graph, i, vis, s);
      }
    }
    while (!s.isEmpty()) {
      System.out.print (s.pop()+ " ");
    }
  }
public static void topSortUtil (ArrayList<Edge>[] graph, int curr, boolean vis[], Stack<Integer> s){
    vis [curr] = true;
    for (int i=0; i<graph [curr].size(); i++) {
      Edge e = graph [curr].get (i);
      if (!vis [e.dest]) {
        topSortUtil (graph, e.dest, vis, s);
      }
    }
    s.push (curr);
  }
```

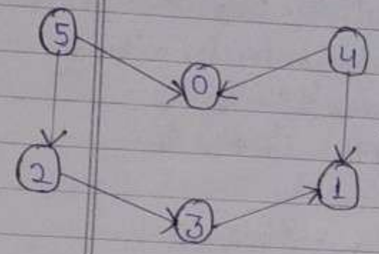
QUES: WHY DFS & not BFS for finding cycle in graphs?
Ans. In DFS, we take one vertex at a time & check if it has cycle. As soon as a cycle is found we can check other vertices. In BFS, we need to keep track of many vertex edges simultaneously & more often than not at the end you find out if it has cycle.

Page No: 515
Date: / /

GRAPHS - III

★ TOPOLOGICAL SORT USING BFS

→ Kahn's Algorithm



→ vertices.

	0	1	2	3	4	5
* in-degree (incoming edges) :	2	2	1	1	0	0
* out-degree (outgoing edges) :	0	0	1	1	2	2

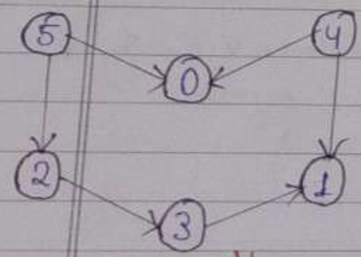
#Fact: A Directed Acyclic Graph has at least one vertex with in-degree 0 (starting node) & one vertex with out-degree 0. (ending node)

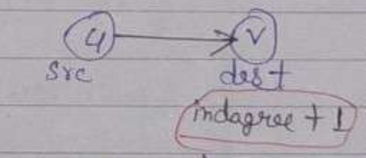
* We will use "in-degree" Approach.

* Topological sort is done on Dependency Graph.

* We will make an "in-degree" array to degrees of each nodes of graphs.

i.e. edges
src dest





indegree []:

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

* BFS (loop on Queue)

जिसकी in-degree 0 है पहले store करा लेंगे।

Queue: 4 5 0 2 3 1 (FIFO)

Output: 4 5 0 2 3 1

nodes के neighbors की ~~out करके~~ indegree -1 करेंगे फिर Queue में डालेंगे।

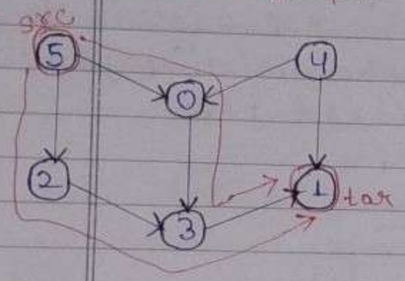
Page No: 516
Date: / /

```
⇒ public static void calcIndeg (ArrayList<Edge> graph[], int indeg[]){
    for(int i=0; i<graph.length; i++){
        int v= i;
        for(int j=0; j<graph[v].size(); j++){
            Edge e = graph[v].get(j);
            indeg[e.dest]++;
        }
    }
}

public static void topSort(ArrayList<Edge> graph[]){
    int indeg[] = new int[graph.length];
    calcIndeg(graph, indeg);
    Queue<Integer> q = new LinkedList<>();
    for(int i=0; i<indeg.length; i++){
        if(indeg[i] == 0){
            q.add(i);
        }
    }
    //bfs
    while(!q.isEmpty()){
        int curr = q.remove();
        System.out.print(curr+" ");//topological sort print
        for(int i=0; i<graph[curr].size(); i++){
            Edge e = graph[curr].get(i);
            indeg[e.dest]--;
            if(indeg[e.dest] == 0){
                q.add(e.dest);
            }
        }
    }
    System.out.println();
}
```

Page No: 517
Date: / /

★ ALL PATHS FROM SOURCE TO TARGET

– Directed graph



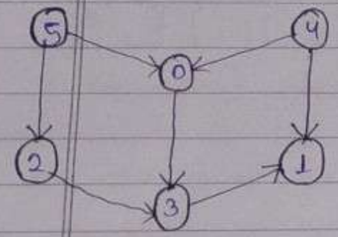
* Source = 5
* Target = 1

⇓

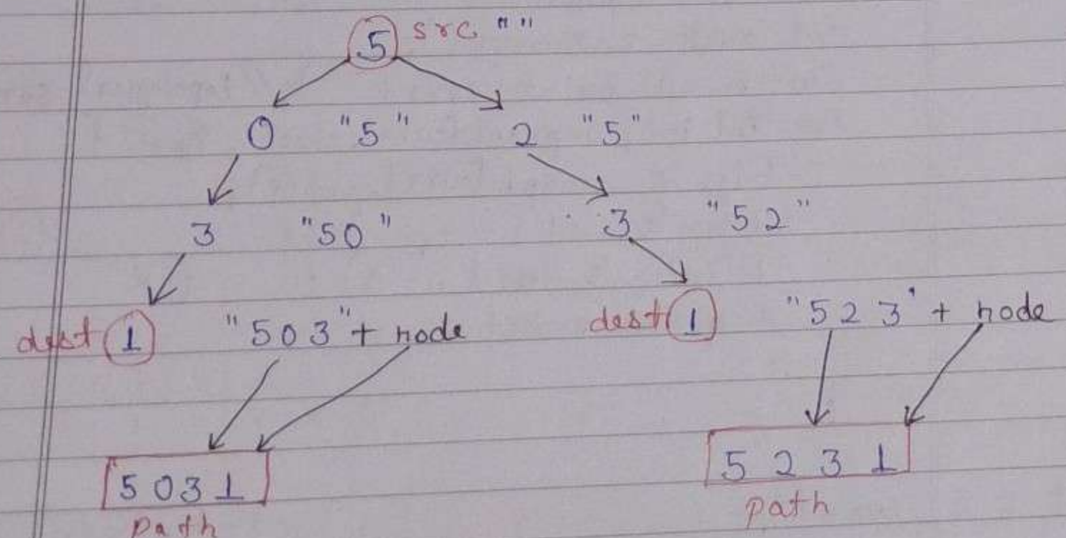
* I^st Path → 5 2 3 1
* II^nd Path → 5 0 3 1

* We will use DFS.

Approach: S = 5, d = 1




```
⇒ dfs (graph, src, dest, path){
    if (src == dest)
        print (path)
    for (neighbor)
        dfs (neighbor, dest,
             path + src)
}
```

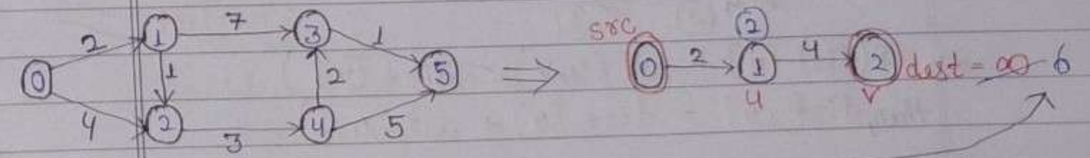


Page No: 518
Date: / /

```
⇒ // Time Complexity - O(V^V) - Exponential Complexity
public static void printAllPath (ArrayList<Edge>
                                 graph[], int src, int dest, String path){
    if (src == dest) {
        System.out.println (path + dest);
        return;
    }
    for (int i=0; i<graph[src].size(); i++) {
        Edge e = graph[src].get(i);
        printAllPath (graph, e.dest, dest, path + src);
    }
}
public static void main (String args[]) {
    int V = 6;
    ArrayList<Edge> graph[] = new ArrayList[V];
    create Graph(graph);
    int src = 5; dest = 1;
    printAllPath (graph, src, dest, "");
}
```

★ DIJKSTRA'S ALGORITHM

– Shortest paths from source to all vertices. (weighted graph)



by this logic

→ dest[u] + wt(u,v) < dest[v]
= dest[v] = dest[u] + wt(u,v)

We cannot use Dijkstra's Algorithm for negative edges Graph.

Page No: 519
Date: / /

⇒ "Dijkstra's Algorithm is graph search algorithm used to find the shortest path between nodes in a weighted graph."

S1: Set the distance of the source node to 0 & all others nodes to ∞. Mark all nodes as unvisited.

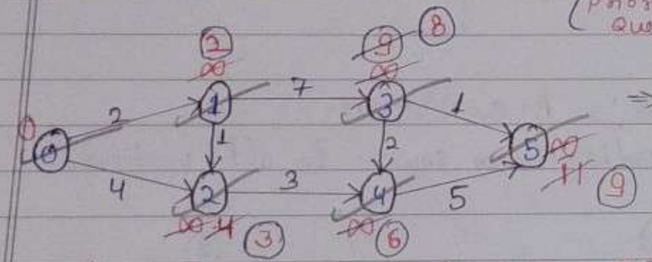
S2: Visit Nodes: Start from source ~~node~~, mark it as visited, & explore its neighbors. For each neighbor, calculate the total distance from the source to that neighbor via current node. If it's smaller than the known distance, update it.

S3: Repeat until all nodes are visited.

S4: When all nodes have been visited, the algorithm end.

S5: Shortest Path to each ~~known~~ nodes is known.

Approach: We will use BFS & min Heap (priority queue) to store.



⇒ हमेशा ये Check करना है की 'Visited X shortest ✓

⇒ if(dist[u] + wt(u,v) < dist[v])
then, dist[v] = dist[u] + wt(u,v)

hence, 0 → 0 ⟹ (0)
0 → 1 ⟹ (2)
0 → 2 ⟹ (3)
0 → 3 ⟹ (8)
0 → 4 ⟹ (6)
0 → 5 ⟹ (9)

Page No: 520
Date: / /

// Time Complexity - V + ElogV

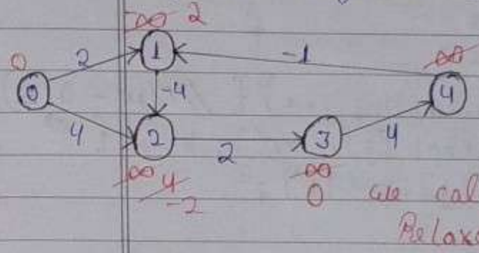
⇒
```
static class Edge {
    int src;
    int dest;
    int wt;
    public Edge(int s, int d, int w)
        this.src = s;
        this.dest = d;
        this.wt = w;
}
static void createGraph(ArrayList<Edge> graph[]) {
    for(int i=0; i<graph.length; i++)
        graph[i] = new ArrayList<>();
    graph[0].add(new Edge(0, 1, 2));
    graph[0].add(new Edge(0, 2, 4));
    graph[1].add(new Edge(1, 3, 7));
    graph[1].add(new Edge(1, 2, 1));
    graph[2].add(new Edge(2, 4, 3));
    graph[3].add(new Edge(3, 5, 1));
    graph[4].add(new Edge(4, 3, 2));
    graph[4].add(new Edge(4, 5, 5));
}
static class Pair implements Comparable<Pair> {
    int n;
    int path;
    public Pair(int n, int path)
        this.n = n;
        this.path = path;
    @Override
    public int compareTo(Pair p2) {
        return this.path - p2.path; //path based sorting
    }                                for my pairs
}
```

```
public static void dijkstra(ArrayList<Edge> graph[], int src) {
    int dist[] = new int[graph.length]; // dist[i] -> src to i
    for (int i=0; i<graph.length; i++) {
        if (i != src) {
            dist[i] = Integer.MAX_VALUE; // +∞
        }
    }
    boolean vis[] = new boolean[graph.length];
    PriorityQueue<Pair> pq = new PriorityQueue<>();
    pq.add(new Pair(src, 0));
    while (!pq.isEmpty()) {
        Pair curr = pq.remove();
        if (!vis[curr.n]) {
            vis[curr.n] = true;
            for (int i=0; i<graph[curr.n].size(); i++) {
                Edge e = graph[curr.n].get(i);
                int u = e.src;
                int v = e.dest;
                int wt = e.wt;
                if (dist[u] + wt < dist[v]) {
                    dist[v] = dist[u] + wt;
                    pq.add(new Pair(v, dist[v]));
                }
            }
        }
    }
}
public static void main(String args[]) {
    int V = 6;
    ArrayList<Edge> graph[] = new ArrayList[V];
    createGraph(graph);   int src = 0;
    dijkstra(graph, src);
}
```


GRAPH - IV

★ BELLMAN FORD ALGORITHM

- Shortest paths from the source to all vertices (valid for both positive & negative weighted edges).
* Time complexity of Bellman Ford is more than Dijkstra Algorithm.



* Perform below operation V-1 times.

⇒ for all edges (u, v)
  if dist[u] + wt(u,v) < dist[v]
    dist[v] = dist[u] + wt(u,v)

we call it Relaxation step.

→ nodes

	0	1	2	3	4
Initialization	0	∞	∞	∞	∞
1	0	2	-2	0	4
2	0	2	-2	0	4
iteration 3	0	2	-2	0	4
4	0	2	-2	0	4

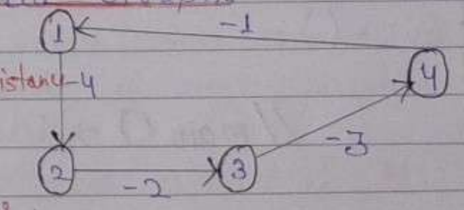
```
for (int i=0 to V-1)
    edge (u → v)
    Relaxation
```

यहले ही iteration में हमें Answer मिल गया। पर ऐसा हमेशा ऐसा नहीं होगा।

Limitation: Bellman Ford Algorithm doesn't work for Negative Weighted Graph.

↓

Shortest Path or Distance का कोई Sense नहीं बनता है ऐसे Graphs में।

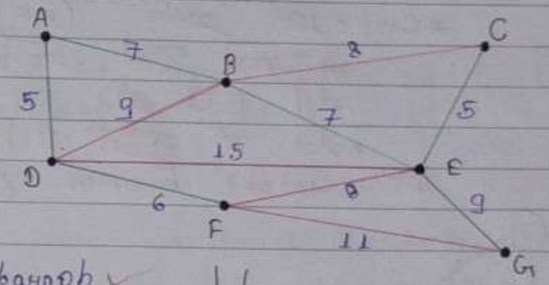


// Time Complexity - O(V*E)

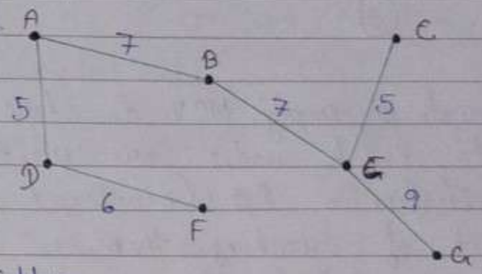
⇒
```
public static void bellmanFord (ArrayList<Edge> graph[], int src){
    int dist[] = new int [graph.length];
    for (int i=0; i< dist.length; i++) {
        if (i != src) {
            dist [i] = Integer.MAX_VALUE;
        }
    }

    int V = graph.length;
    //Algorithm starts
    for (int i=0; i<V-1; i++){
      for (int j=0; j< graph.length; j++) { //Edges - O(E)
        for (int k=0; k < graph [j].size(); k++) {
            Edge e = graph [j].get [k];
            //u, v, weight
            int u = e.src;
            int v = e.dest;
            int wt = e.wt;
            //Relaxation
            if (dist [u] != Integer.MAX_VALUE && dist[u] +
                    wt < dist [v]) {
                dist [v] = dist [u] + wt;
            }
        }
    }
}

for (int i=0; i<dist.length; i++) {
    System.out.print (dist [i] + " ");
}
Sys o ();
}

//main () → bellmanFord (graph, 0);
```


★ MINIMUM SPANNING TREE (MST)

→ A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles & with the minimum possible total edge weight.



MST of
- Subgraph ✓
- Cycle ✗
- Vertices connected
- Tot. weight (min)

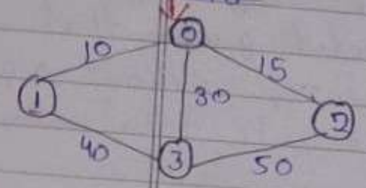


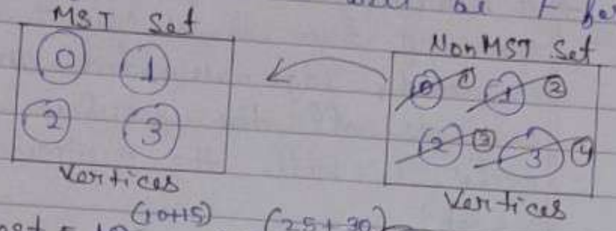
→ Prim's Algorithm

MST Set

* Prim's algorithm is greedy & has a straightforward way to create a minimum spanning tree.
* The MST found by Prim's Algorithm is the collection of edges in a graph, that connects all vertices, with minimum sum of edges weights.
* Prim's Algorithm uses concept of MST Set.

At initial, → all nodes will be in notMST set & MST set will be empty & vis[] will be F for all.



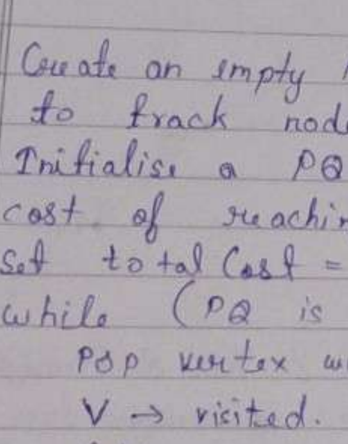


vis[]: | T | T | T | T |
0 1 2 3

* Cost = ~~10~~ (0+15) 25 (25+30) 55

सबसे पहले zero को निकाल लेंगे इसके बाद हम लोग उसे nodes को छूकर लेंगे जहाँ जाने का weight minimum होगा।

Answer.



∴ Total min Weight = 55

S1. Create an empty MST to store vertices, & visited[] to track nodes are visited or not (T or F).
S2: Initialise a PQ (min-heap) to store the cost of reaching them.
S3: Set totalCost = 0;
S4. while (PQ is not empty)
POP vertex with min cost from PQ (edge with smallest weight)
V → visited.
Add cost of current edge
Add this ~~edge~~ vertex to MST
}
S5. Return Total Cost & MST.

// Time Complexity - O(ElogE)

⇒
```
static class Pair implements Comparable<Pair>{
    int v;
    int cost;
    Public Pair(int v, int c){
        this.v = v;
        this.cost = c;
    }
    @Override
    public int compareTo(Pair p2){
        return this.cost - p2.cost;
    }
}

public static void prims(ArrayList<Edge> graph){
    boolean vis[] = new boolean[graph.length];
    PriorityQueue<Pair> pq = new PriorityQueue<>();
    pq.add(new Pair(0, 0));
    int finalCost = 0; // MST Cost / Total min Weight
    while(!pq.isEmpty()){
        Pair curr = pq.remove();
        if(!vis[curr.v]){
            vis[curr.v] = true;
            finalCost += curr.cost;
            for(int i=0; i<graph[curr.v].size(); i++){
                Edge e = graph[curr.v].get(i);
                pq.add(new Pair(e.dest, e.wt));
            }
        }
    }
    Sysо("Final (min) cost of MST = " + finalCost);
}
```

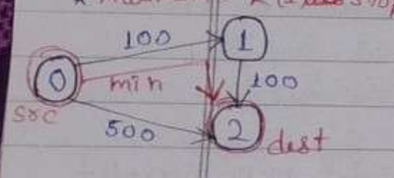
GRAPHS - V

★ CHEAPEST FLIGHTS WITHIN K STOPS

→ There are n cities connected by some number of flights. You are given an array flights where flights[i] = [from, to, price] indicates that there is a flight.
You are also given three integers src, dst & k, return the cheapest price from src to dst with at most k stops. If there is no such route, return -1.

All values are Positive.

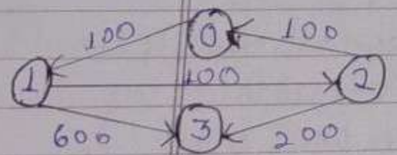
* maximum = k (1) stops



→ flights = [[0,1,100],[1,2,100],[0,2,500]]
→ src = 0, → dst = 2, → k = 1
⇒ ans = 200

* We will use Dijkstra's Algorithm + modification.

Approach: flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]]
src = 0, dst = 3, k = 1



* We will use Queue<Info>,

vertex · path (cost) · stops

F R

(0,0,0) (1,100,1) (2,200,2) (3,700,2)

dist[u] + wt < dist[v]
dist[v] = dist[u] + wt

dist[]: 0 | 100 | 200 | 700
0 1 2 3

ans: 700


```
⇒ static class Edge {
    int src;
    int dest;
    int wt;
    public Edge (int s, int d, int wt) {
        this.src = s;
        this.dest = d;
        this.wt = wt;
    }
}
public static void createGraph (int flights[][],
                                ArrayList<Edge> graph[]) {
    for (int i=0; i<graph.length; i++) {
        graph[i] = new ArrayList<>();
    }
    for (int i=0; i<flights.length; i++) {
        int src = flights[i][0];
        int dest = flights[i][1];
        int wt = flights[i][2];
        Edge e = new Edge (src, dest, wt);
        graph[src].add(e);
    }
}
static class Info {
    int v; int cost; int stops;
    public Info (int v, int c, int s) {
        this.v = v;
        this.cost = c;
        this.stops = s;
    }
}
```

```
public static int cheapestFlight(int n, int[][] flights, int src, int dest, int k) {
    ArrayList<Edge> graph[] = new ArrayList[n];
    createGraph(flights, graph);
    int dest[] = new int[n];
    for (int i=0; i<n; i++) {
        if (i != src) {
            dist[i] = Integer.MAX_VALUE;
        }
    }

    Queue<Info> q = new LinkedList<>();
    q.add(new Info(src, 0, 0));
    while (!q.isEmpty()) {
        Info curr = q.remove();
        if (curr.stops > k) {
            break;
        }
        for (int i=0; i<graph[curr.v].size(); i++) {
            Edge e = graph[curr.v].get(i);
            int u = e.src;
            int v = e.dest;
            int wt = e.wt;
            if (curr.cost + wt < dist[v] && curr.stops<=k){
                dist[v] = curr.cost + wt;
                q.add(new Info(v, dist[v], curr.stops+1));
            }
        }
    }
    if (dist[dest] == Integer.MAX_VALUE) {
        return -1;
    } else {
        return dist[dest];
    }
}
```


★ CONNECTING CITIES WITH ~~MAX~~ MINIMUM COST

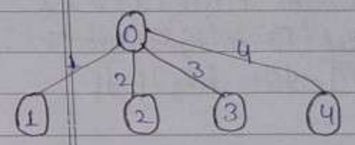
- Find the minimum cost for connecting all cities on the map.

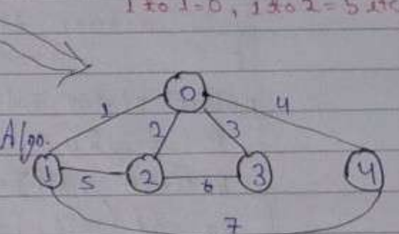
Ex → cities[][] = {{0,1,2,3,4}, {1,0,5,0,7}, {2,5,0,6,0}, {3,0,6,0,0}, {4,7,0,0,0}}

⇒ cities[i][j] = (i) —dist— (j)

* distances are stored in 2-D array i.e., 0 to 0 = 0, 0 to 1 = 1, 1 to 2 = 5 etc.

ans = 10





*using Prim's Algo.

PQ (edges) – it sort edges automatically

(1,1) (2,2) (3,3) (4,4) (2,5) (4,7) (3,6)

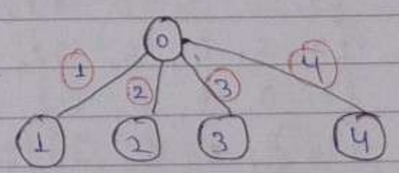
MST: 0, 1, 2, 3, 4 — Visited

Non-MST: 0, 1, 2, 3, 4 — Non-Visit

F	F	F	F	F
0	1	2	3	4

⇒ Cost = 0 + 1 + 2 + 3 + 4 = 10

ans graph →



⇒
```
static class Edge implements Comparable<Edge> {
    int dest;
    int cost;
    public Edge(int d, int c)
        this.dest = d;
        this.cost = c;
    @Override
    public int compareTo(Edge e2) {
        return this.cost - e2.cost;
}
public static int connectCities(int cities[][]) {
    PriorityQueue<Edge> pq = new PriorityQueue<>();
    boolean vis[] = new boolean[cities.length];
    pq.add(new Edge(0, 0));
    int finalCost = 0;
    while (!pq.isEmpty()) {
        Edge curr = pq.remove();
        if (!vis[curr.dest])
            vis[curr.dest] = true;
            finalCost += curr.cost;
            for (int i=0; i<cities[curr.dest].length; i++)
                if (cities[curr.dest][i] != 0)
                    pq.add(new Edge(i, cities[curr.dest][i]));
    return finalCost;
}
public static void main(String args[]) {
    int cities[][] = {{0, 1, 2, 3, 4},
                      {1, 0, 5, 0, 7},
                      {2, 5, 0, 6, 0},
                      {3, 0, 6, 0, 0},
                      {4, 7, 0, 0, 0}};
    Sysо(connectCities(cities));
}
```


★ DISJOINT SET or UNION FIND DATA STRUCTURE

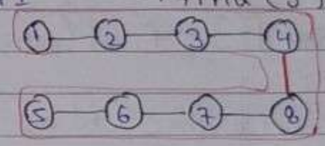
* It is used to store & track non-overlapping sets.
* It provides operations like 1. Find - which determines which subset a particular element is in and 2. Union - which merges two subsets into a single subset.

Ex→ graph 1 (set 1) : ① — ② — ③ — ④

graph 2 (set 2) : ⑤ — ⑥ — ⑦ — ⑧

→ find (2) : set 1 → find (5) : set 2

→ Union (4, 8) :



Now,

find (2) : set 1 } because both are merged & are in same set.
find (5) : set 1 }

* These DS are used in implementation of 1. Cycle Detection & 2. Kruskal's Algorithm.

* Implementation (Optimized). Parent + union by rank

→ n = 8

→ par []:

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

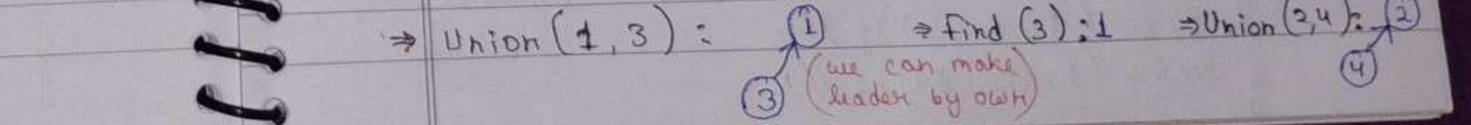
(3 → 1, 4 → 2) { at starting all elements are parent of their own.

→ rank []: (height)

0	1	2	3	4	5	6	7
0	1	1	0	0	0	0	0
0	1	2	3	4	5	6	7

find: return leader.
Union: join 2 groups

⇒ Union (1, 3) : ⇒ find (3) : 1 ⇒ Union (2, 4) :



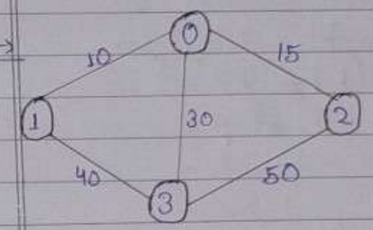
// Time Complexity → O(1) – Constant

⇒
```
static int n=7;
static int par[] = new int[n];
static int rank[] = new int[n];
public static void init() {
  for(int i=0; i<n; i++){
    par[i] = i;
  }
}

public static int find(int x) {
  if(x == par[x]){
    return x;
  }
  return par[x] = find(par[x]); // Path compression (directly store first ancestor)
}
public static void union(int a, int b){
  int parA = find(a);
  int parB = find(b);
  if(rank[parA] == rank[parB]) {
    par[parB] = parA;
    rank[parA]++;
  } else if(rank[parA] < rank[parB]) {
    par[parA] = parB;
  } else {
    par[parB] = parA;
  }
}
public static void main(String args[]) {
  init(); Syso(find(3)); // 3   union(1,3);
  Syso(find(3)); // 1
  union(2,4); union(3,6); union(1,4);
  Syso(find(3)); // 1
  Syso(find(4)); // 1
}
```


★ KRUSKAL'S ALGORITHM

* It is used to find out MST. (min cost tree)
* It is a Greedy Algorithm.
* Algorithm says,
  i) Sort Edges
  ii) Take min. cost edge
  iii) Check that cycle shouldn't exist
  iv) Include answer.

Ex→



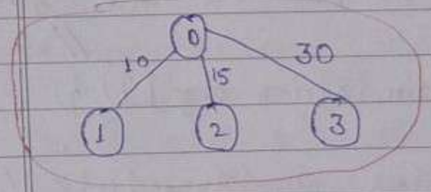
* ArrayList<Edge>
  L→ Collections.Sort()
  (it will sort edge)

Sl: Sorted Edges — Weight

① → (0, 1) — 10 ✓
② → (0, 2) — 15 ✓
③ → (0, 3) — 30 ✓
④ → (1, 3) — 40 ✗ if we include it, then cycle will exist.
⑤ → (2, 3) — 50 ✗ it will also generate cycle so not include.

ans: 10 + 15 + 30

55



MST

* for(i=0 to V-1){
  we will use union-find for detecting cycle.
  include answer
}

//Time-Complexity : O(V) + E log E)

⇒
```
static class Edge implements Comparable<Edge>{
    int src;
    int dest;
    int wt;
    public Edge(int s, int d, int w){
        this.src = s;
        this.dest = d;
        this.wt = w;
    }
    @Override
    public int compareTo(Edge e2){
        return this.wt - e2.wt;
    }
}
static void createGraph(ArrayList<Edge> edges){
    //edges
    edges.add(new Edge(0, 1, 10));
    edges.add(new Edge(0, 2, 15));
    edges.add(new Edge(0, 3, 30));
    edges.add(new Edge(1, 3, 40));
    edges.add(new Edge(2, 3, 50));
}
```

// write code of union, find of page-533, excluding main(); at place of n=7 → n=4, // vertices

```
public static void main(String args[]){
    int V=4;
    ArrayList<Edge> edges = new ArrayList<>();
    createGraph(edges);
    KruskalsMST(edges, V);
}
```

//Next page


```
public static void kruskalsMST(ArrayList<Edge> edges, int V){
    init();
    Collections.sort(edges);
    int mstCost = 0;
    int count = 0;
    for(int i=0; count < V-1; i++){
        Edge e = edges.get(i);
        // (src, dest, wt)
        int parA = find(e.src); //src = a
        int parB = find(e.dest); //dest = b
        if(parA != parB){
            union(e.src, e.dest);
            mstCost += e.wt;
            count++;
        }
    }
    System.out.println(mstCost);
}
```

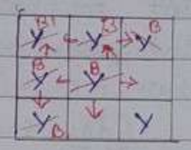
★ FLOOD FILL ALGORITHM

- Given a m*n integer grid image where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc & color. You should perform a flood fill on the image starting from the pixel image[sr][sc].
- 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 same color) & so on. Replace color of all of the aforementioned pixels with color.

Ex⇒ image = [[1, 1, 1],
[1, 1, 0],
[1, 0, 1]]

* sr = 1 (starting row), sc = 1 (starting column), colOr = 2

* Let's imagine, 1 → yellow
2 → blue
0 → white



* image[sr][sc] से पहले color change then 4-directions में फैलते रहना।

ans: [[2, 2, 2],
[2, 2, 0],
[2, 0, 1]]

Logic: Recursively call {
Left (sr, sc-1)
right (sr, sc+1)
up (sr-1, sc)
down (sr+1, sc)
}

//Base Cases
sr, sc - invalid (boundary) < 0
=> image.length

vis[sr, sc] = 1
image[sr][sc] != orgColor

* Above 3 condition, ~~are~~ when they will be true so don't change color.

//Time Complexity - O(m*n)

```
⇒ public void helper(int[][] image, int sr, int sc, int color, boolean vis[][], int orgCol) {
    if (sr<0 || sc<0 || sr>=image.length || vis[sr][sc]
        || sc>=image[0].length || image[sr][sc] != orgCol) {
        return;
    }
    //Left
    helper(image, sr, sc-1, color, vis, orgCol);
    //Right
    helper(image, sr, sc+1, color, vis, orgCol);
    //Up
    helper(image, sr-1, sc, color, vis, orgCol);
    //down
    helper(image, sr+1, sc, color, vis, orgCol);
}
public int[][] floodFill(int[][] image, int sr, int sc, int color) {
    boolean vis[][] = new boolean[image.length][image[0].length];
    helper(image, sr, sc, color, vis, image[sr][sc]);
    return image;
}
```

★ FLOYD WARSHALL ALGORITHM

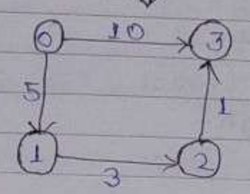
- It is an algorithm for finding the shortest path b/t all the pairs of vertices in a weighted graph. It works for both the directed & undirected weighted graphs.

But, it does not work for the graphs with negatives cycles (where sum of the edges in a cycle is negative).

//Next page

Ex→ Input : graph[][] = {{0, 5, INF, 10},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0}}

{∵ INF ⇒ ∞

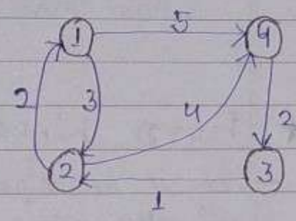


Output: 0 5 8 9
INF 0 3 4
INF INF 0 1
INF INF INF 0

Approach: → Create n × n matrix, n is no. of vertices.
→ Each cell is filled with distance between (i, j).
→ If there is no path ⟶ ∞.

$$A^0 = \begin{bmatrix} 1\to1 & 1\to2 & 1\to3 & 1\to4 \\ 0 & 3 & \infty & 5 \\ 2\to1 & 2\to2 & 2\to3 & 2\to4 \\ 2 & 0 & \infty & 4 \\ 3\to1 & 3\to2 & 3\to3 & 3\to4 \\ \infty & 1 & 0 & \infty \\ 4\to1 & 4\to2 & 4\to3 & 4\to4 \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

in graph form



→ Let k be intermediate vertex in shortest path.
* if (A[i][j] > A[i][k] + A[k][j]) {
⇒ A[i][k] + A[k][j] will be written
}
→ Elements of 1st column & 1st row are remain same.

S1: k is vertex 1.

$$A1 = \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & & \\ \infty & & 0 & \\ \infty & & & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 3 & \infty & 5 \\ 2 & 0 & 9 & 4 \\ \infty & 1 & 0 & 9 \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

→ Now, calculate distance from source vertex to destination.
→ Elements of IInd row & column will remain as it is.
→ **S2:** K is second vertex (i.e. 2).

$$A^2 = \begin{bmatrix} 0 & 3 & & \\ 2 & 0 & 9 & 4 \\ & 1 & 0 & \\ \infty & & & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ \infty & \infty & 2 & 0 \end{bmatrix}$$

→ Similarly, A3 & A4 is also created.

S3: K is vertex 3.

$$A^3 = \begin{bmatrix} 0 & & \infty & \\ 2 & 0 & 9 & \\ \infty & 1 & 0 & 8 \\ & & 2 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 3 & 9 & 5 \\ 2 & 0 & 9 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix}$$

→ **S4:** K is vertex 4.

$$A^4 = \begin{bmatrix} 0 & & & 5 \\ & 0 & & 4 \\ & & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 3 & 7 & 5 \\ 2 & 0 & 6 & 4 \\ 3 & 1 & 0 & 5 \\ 5 & 3 & 2 & 0 \end{bmatrix}$$

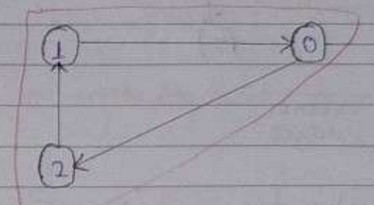
* We ~~two~~ will perform this steps till our no. of vertex i.e., no. of vertex = 4 so there are A0, A1, A3, A4.
* Floyd-Warshall Algorithm uses three nested loops, each running V times, where V is no. of vertices in ~~to~~ graph. — $O(V^3)$ Time Complexity
* 2D-Array of size V × V, Space Complexity $O(V^2)$.

⇒
```
static int INF = 99999, V = 4;
void floydWarshall (int graph[][]) {
    int dist[][] = new int [V][V];
    int i, j, k;
    for (i=0; i<V; i++) {
        for (j = 0; j<V; j++) {
            dist[i][j] = graph[i][j];
        }
    }
    for (k=0; k<V; k++) {
        for (i=0; i<V; i++) {
            for (j=0; j<V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
    printSolution(dist);
}
void printSolution (int dist[][]) {
    Syso("Following Matrix shows Shortest Path: ");
    for (int i=0; i<V; ++i)
        for (int j=0; j<V; ++j)
            if (dist[i][j] == INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j]+" ");
    Syso();
}
```


★ STRONGLY CONNECTED COMPONENT

- SCC is a component in which we can reach every vertex of the component from every other vertex in that component.

Ex:

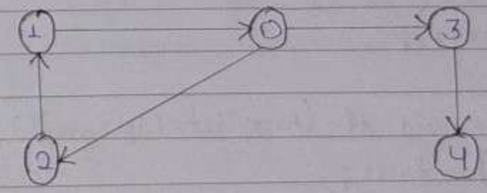


* There are 3 SCC.

* SCC can only happen in Directed graph.

→ We will use Kosaraju's Algorithm:

* Kosaraju's Algorithm is used to find SCC in a directed graph.
* This Algorithm uses DFS.

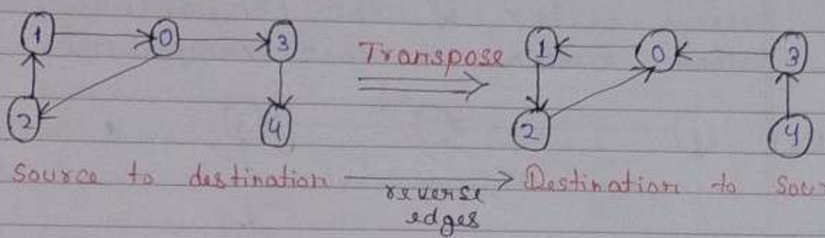


Normal DFS	Reverse DFS
0 2 1 3 4	4
	3
	2 1 0

✓ SCC

S1: Get nodes in Stack (Topological Sort)
S2: Transpose the graph
S3: Do DFS according to stack nodes on the transpose graph.

* Transpose of the graph means reversing the edges i.e.,



// Time Complexity - O(V + E)

```
⇒ public static void topSort(ArrayList<Edge> graph[], int curr, boolean vis[], Stack<Integer> s){
    vis[curr] = true;
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(!vis[e.dest]){
            topSort(graph, e.dest, vis, s);
        }
    }
    s.push(curr);
}

public static void dfs(ArrayList<Edge> graph[], int curr, boolean vis[]){
    vis[curr] = true;
    System.out.print(curr + " ");
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(!vis[e.dest]){
            dfs(graph, e.dest, vis);
        }
    }
}
```

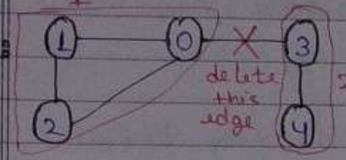
// Next page


```
public static void kosaraju(ArrayList<Edge> graph[], int V){
    //Step-1
    Stack<Integer> s = new Stack<>();
    boolean vis[] = new boolean[V];
    for(int i=0; i<V; i++){
        if(!vis[i]){
            topSort(graph, i, vis, s);
        }
    }
    //Step-2
    ArrayList<Edge> transpose[] = new ArrayList[V];
    for(int i=0; i<graph.length; i++){
        vis[i] = false;
        transpose[i] = new ArrayList<Edge>();
    }
    for(int i=0; i<V; i++){
        for(int j=0; j<graph[i].size(); j++){
            Edge e = graph[i].get(j); //e.src -> e.dest
            transpose[e.dest].add(new Edge(e.dest, e.src)); //reverse edge
        }
    }
    //Step-3
    while(!s.isEmpty()){
        int curr = s.pop();
        if(!vis[curr]){
            System.out.print("SCC -> ");
            dfs(transpose, curr, vis); //src
            System.out.println();
        }
    }
}
```

Page No: 545
Date: / /

★ BRIDGE IN GRAPH

- Bridge is an edge whose deletion increases the graph's number of connected components.

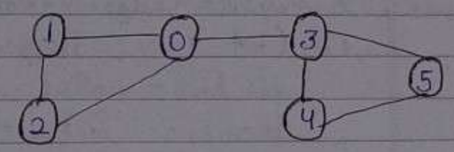
Ex: 

* Before deletion, it is 1 connected component.
* After deletion, there will be 2 connected components.

* We will use Tarjan's Algorithm.

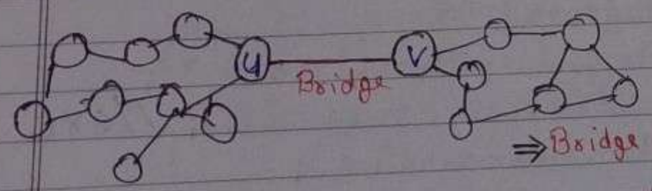
Approach:

V = 6.



* We will make two arrays.
  - → dt[] = new int[V] → discovery time of node. (means वो Time जब उस node को discover या visit. कर लिए हैं)
  - → low[] = new int[V]. lowest discovery time of all neighbors.
* We will use modify DFS.

→ LDT of all neighbors मतलब ~~discov~~ min. discovery उस node के neighbors & including उसके खुद का भी।



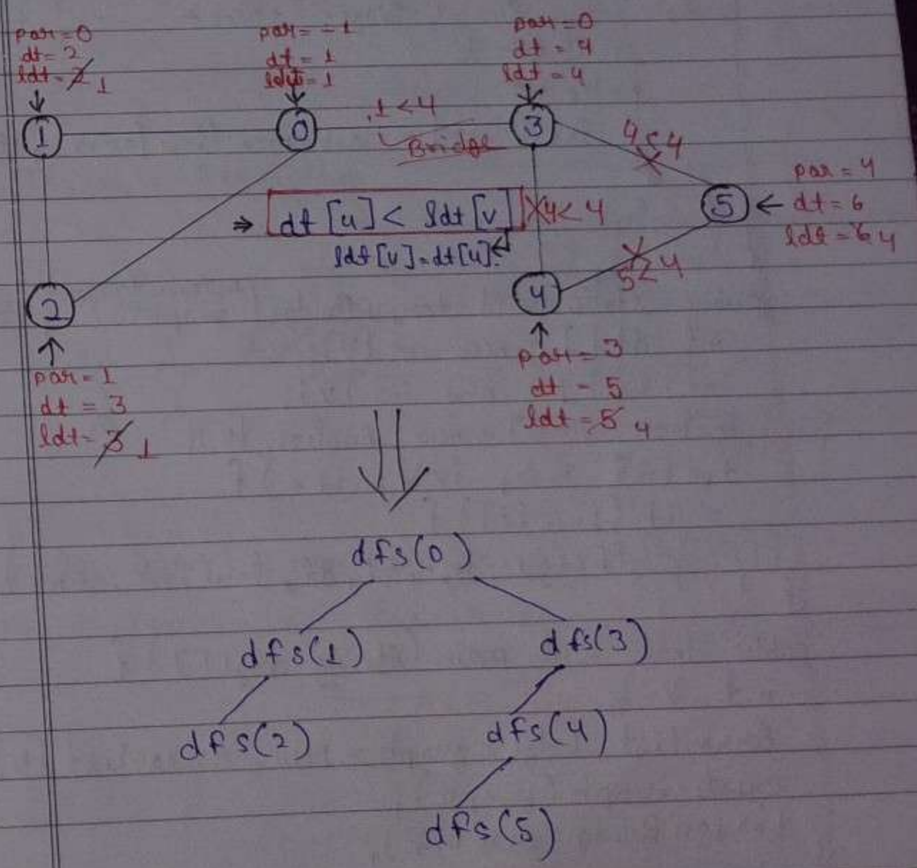
⇒ Bridge Case: dt[u] < low[v]
src < dest

* कोई Edge, Bridge तभी होगा जब वो single path हो u से तक v तक जाने का।

Page No: 546
Date: / /

Pseudo code:

```
vis[curr] = true
dt[curr] = low[curr] = ++time
for (all neighbors) {
    Edge e ;  → src, dest
① neighbor = parent then ignore.
② !vis[neighbor]
      dfs(neighbor)
      low[curr] = min[low[curr], low[neighbor]])
      if (dt[curr] < low[neighbor])
          print (Bridge (curr, neighbor)
③ vis[neighbor]
      low[curr] = min(low[curr], dt[neighbor]])
```



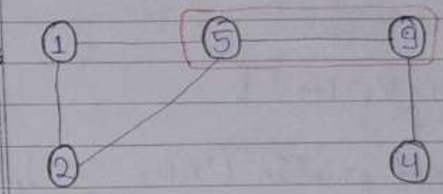
// Time Complexity - O(V+E)

⇒
```
public static void dfs(ArrayList<Edge> graph[], int curr, int par, int dt[], int low[], boolean vis[], int time){
    vis[curr] = true;
    dt[curr] = low[curr] = ++time;
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i); //e.src ---e.dest
        int neigh = e.dest;
        if(neigh == par){
            continue;
        } else if(!vis[neigh]){
            dfs(graph, neigh, curr, dt, low, vis, time);
            low[curr] = Math.min(low[curr], low[neigh]);
            if(dt[curr] < low[neigh]){
                System.out("Bridge: "+curr+"---"+neigh);
            } else{
            low[curr] = Math.min(low[curr], dt[neigh]);
        }
    }
}

public static void tarjanBridge(ArrayList<Edge> graph[], int V){
    int dt[] = new int[V];
    int low[] = new int[V];
    boolean vis[] = new boolean[V];
    for(int i=0; i<V; i++){
        if(!vis[i]){
            dfs(graph, i, -1, dt, low, vis, time);
        }
    }
}

public static void main(String args[]){
    int V=6;
    ArrayList<Edge> graph = new ArrayList[V];
    createGraph(graph);
    tarjonBridge(graph, V);
}
```


★ ARTICULATION POINT

- An articulation point (or cut vertex) is a vertex in an undirected connected graph if removing it (& edges through it) disconnects the graph.
- Articulation points are single points in a connected network that if they fail, split the network into two or more components.
- An articulation point in a disconnected undirected graph is a vertex removal that increases the number of connected components.

Ex:

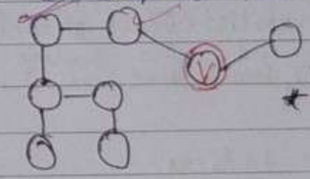


5 & 9 are the articulation points.

* Tarjan's Algorithm

- A node 'A' that was discovered before current node in DFS, is the ancestor of current.

Ex:



* for this V node, जो V node से पहले Visited nodes हैं वो उसके Ancestor होंगे।

→ low[curr] = min(low[curr], dt[neigh])

⇒ dt[curr] <= low[neigh]

↳ Articulation Point condition.

// Time - Complexity - O(V*(V+E))

⇒
```
public static void getAP(ArrayList<Edge> graph[], int V){
    int dt[] = new int[V];
    int low[] = new int[V];
    int time = 0;
    boolean vis[] = new boolean[V];
    boolean ap[] = new boolean[V];
    for(int i=0; i<V; i++){
        if(!vis[i]){
            dfs(graph, i, -1, dt, low, time, vis, ap);
        }
    }
    //print all APs
    for(int i=0; i<V; i++){
        if(ap[i]){
            System.out.println("AP: "+i);
        }
    }
}
//O(V+E)
public static void dfs(ArrayList<Edge> graph[], int curr, int par, int dt[], int low[], int time, boolean vis[], boolean ap[]){
    vis[curr] = true;
    dt[curr] = low[curr] = ++time;
    int children = 0;
    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        int neigh = e.dest;
        if(par == neigh){
            continue;
        } else if(vis[neigh]){
            low[curr] = Math.min(low[curr], dt[neigh]);
```



```
        } else {
            dfs(graph, neigh, curr, dt, low, time, vis, ap);
            low[curr] = Math.min(low[curr], low[neigh]);
            if(par != -1 && dt[curr] <= low[neigh]){
                ap[curr] = true;
            }
            children++;
        }
    }
    if(par == -1 && children > 1){
        ap[curr] = true;
    }
}
```