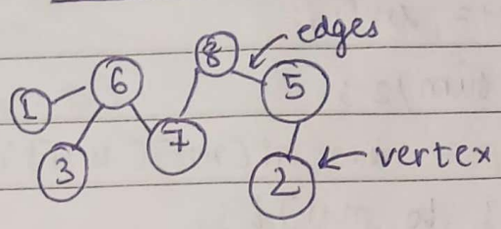
Ajanta
Page No.
Date 10-07-25

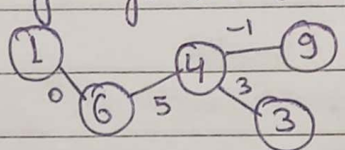
GRAPHS

* N/w of nodes =>

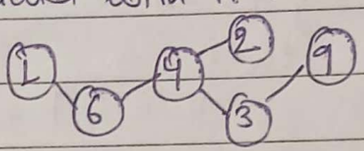


* Edges :-
  - Unidirectional / Directed graph → A → B
    ie. we can't go from B to A.
  - Undirected graph → A — B or A ⇄ B .

* Every edge has a property associated with it i.e. called Weight.



(0,5,-1,3) = weights
Weighted graph

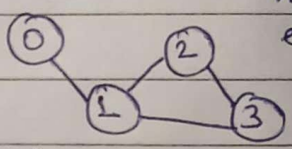


Unweighted graph

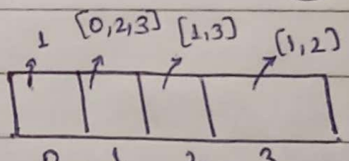
* Storing a graph (representation)

I. Adjacency list → • list of lists

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



Vertices = 4
edges = 4
unweighted
undirected

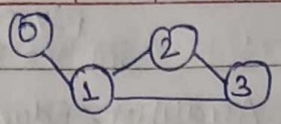


- Can be stored as :- ArrayList<ArrayList>
  Array<ArrayList>
  HashMap<int, list>

- Array of ArrayList<Edge>
  Edge → src, dest, wt (by default → 1)

e.g. edge b/w 1 & 3 has → src=1, dest=3, wt. = 1.

II. Adjacency matrix



V=4
E=4

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

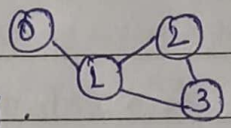
more space req. than adjacency list → $O(V^2)$

Problem → Doing 2 times for single vertex.

→ For 1 vertex we're checking for all vertices.

Benefit → we can store weighted info. ~~in~~ using this matrix.

III. Edge List



Edges = { {0,1}, {1,2}, {1,3}, {2,3} }
e1 e2 e3 e4

with weights → { {src dest wt} } { {0,1,4}, {1,2,0}, {1,3,-4}, {2,3,5} }
e1 e2 e3 e4

Eg. Min. Spanning Tree – here we sort edges using edge list, so it become easier through this.

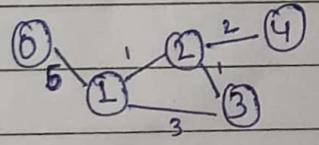
IV. Implicit graph → where cells are there with (i,j) order of one cell.

* Creating a Graph

using Adjacency list → v.v.v. easy → Array of ArrayList<Edges>

Edges = (src, dest, wt.)

e.g.



0 → {0,1,5}

1 → {1,0,5}, {1,2,1}, {1,3,3}.

vertex and src are always same.

⇒
```
import java.util.*;
public class adjList {
    static class Edge {
        int src;
        int dest;
        int wt;

        Const^r of all three
    }
}
```

psvm {
int V = 5 ; //vertices = V

Array of ArrayList of Type Edges with size V. ← ArrayList<Edge>[] graph = new ArrayList[V]; (int[] arr = new int[V])

Adding arraylist to all index of graph array → for (i=0 to V){
graph[i] = new ArrayList<>();
}

Add arraylist info to all indices → graph[0].add(new Edge(0,1,5));

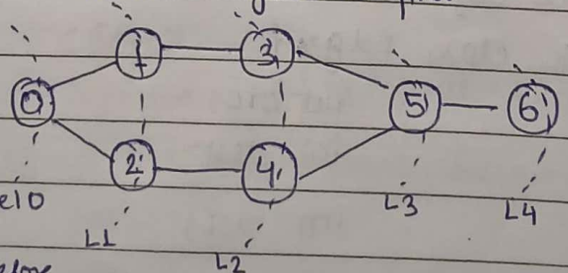
graph[1].add(new Edge(1,0,5));
graph[1].add(new Edge(1,2,1));
graph[1].add(new Edge(1,3,3));

Ily, for index = 2, 3, 4.

checking neighbours → e.g. 2's neighbour → for (i=0 to graph[2].size()){
Edge e = graph[2].get(i);
print(e.dest + " ");
}

* Graph Traversals
1. BFS (Breadth First Search)
2. DFS (Depth First Search)

I. BFS → Level order traversals
→ go to Immediate neighbours first



eg. can't access 3 before 2,1.
can't access 5 before 3,4

0 → 1 → 2 → 3 → 4 → 5 → 6

concept:→ 1. make a boolean type array for checking visited.
2. queue to store elements.

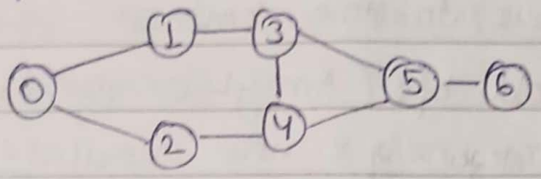
code→
```
static void bfs (ArrayList<Edge>[] graph){
    Queue<Integer> q = new LinkedList<>();
    boolean[] vis = new boolean[];

    q.add(0); → src=0, as we're starting from 0.
    while (!q.isEmpty()){
        int curr = q.poll();
        if (!vis[curr]){ → if not visited
            print(curr+" ");
            vis[curr] = true; → make it visited
            for (i=0 to graph[curr].size()){
                Edge e= graph[curr].get(i);
                q.add(e.dest);
            }
        }
    }
```

☆ Har vertex ek yehle saara destination visit hona then proceed karna → BFS.

define classes of Edge

```
static void createGraph(ArrayList<Edge> graph[]){
    for (i=0 to graph.length){
        graph[i] = new ArrayList<>();
    add arraylist on all index→
    graph[0].add(new Edge(0,1,5));
    :
    :
psvm{
    int V=7;
    ArrayList<Edge>[]graph = new ArrayList[V];
initialise Array of Lists. ←
    createGraph(graph);
    bfs(graph);
}
```

II. DFS (Depth first search)



0 → 1 → 3 → 4 → 2 → 5 → 6.

already covered ←

approach → 0 → 1(~~0~~, 3) → 3(~~4~~, ~~1~~, 5) → 4(~~2~~, ~~3~~, 5) → 2(~~0~~, ~~4~~) → 5(~~3~~, ~~4~~) → 6.

priority on basis of neighbours not levels. Recursive approach

imp. since we're tracking vis in every recursive call

code → defining class Edge, createGraph as before.

```
static void dfs(ArrayList<Edge>[] graph, int curr, boolean[] vis){
    print(curr + " ");
    vis[curr] = true; → marked current one true
    for(i=0 to graph[curr].size()){
        Edge e = graph[curr].get(i);
        if(!vis[e.dest]){    // if ~~current~~ neighbours is not visited
            dfs(graph, e.dest, vis);
        }
    }
}
```

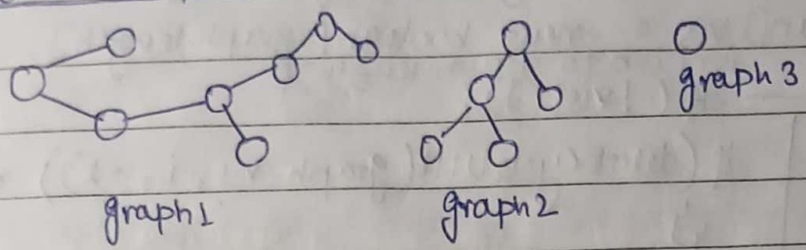
same as psvm of bfs → dfs(graph, 0, new boolean[v]);
current starting from 0.

* Has Path

to check in a graph does (5) vertex exist in the path or not if starting from (0) vertex. → True.

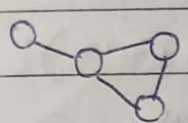
```
static boolean hasPath(ArrayList<Edge>[] graph, int src, int dest, boolean[] vis){
    if(src == dest){
        return true;
    }
    for(i=0 to graph[src].size()){
        Edge e = graph[src].get(i);
        if(!vis[e.dest] && hasPath(graph, e.dest, dest, vis)){
            return true;
      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

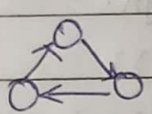
* Connected Component → check



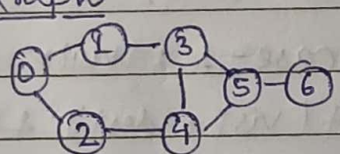
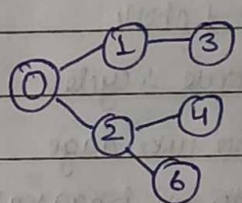
combining graph 1 and 2 with 1 edge ∴ connected component = 1.

* Detect cycle in Graph

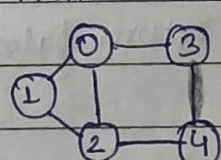
*  undirected → DFS, BFS, Disjoint set union

*  directed → DFS, BFS, Topological sort (Kahn's Algo)

I. Undirected Graph

cycle →  ; nocycle → 

*Condition for cycle: When the neighbour (e.dest) of any node is already visited.

3 cases:- (think properly) 

1 is the parent of 2
0 " " " " 1
since in DFS → 0 → 1 → 2.

every node has these 3 cases

1. vis[neigh] ✓, parent ✗ → neigh. visited but not parent ∴ Always a cycle
2. vis[neigh] ✓, parent ✓ → cycle not necessary ∴ continue
3. vis[neigh] ✗ → proceed further ∴ DFS call

Starting node parent's be -1. (let)

→ In the given graph for node 2 → neighbours (reached till 2) → (0) → visited, parent ✗
(1) → visit ✓, parent ✓
(4) → visit ✗

DFS Traversal from 0 → (0 → 1 → 2 → 3 → 4)

```
static boolean detectCycle(ArrayList<Edge>[] graph){
    boolean[] vis = new boolean[graph.length];
    for(i=0 to graph.length){
        if(!vis[i]){
            if(detectCycleUtil(graph, vis, i, -1)) return true;
        }
    }
    return false;
}
static boolean detectCycleUtil(ArrayList<>..., ....., int curr, int parent
    vis[curr] = true;
    for(i=0 to graph[curr].size()){
        // case-3 : Not visited neigh.
        Edge e = graph[curr].get(i);
        if neigh is not visited ← * if(!vis[e.dest] && detectCycleUtil(graph, vis, e.dest, curr)){
            return true;
        // case-1 : visit ✓, parent ✗
        else if(vis[e.dest] && e.dest != parent){
            return true;
        // case-2 : Do nothing → continue (by default)
    return false;
```

why always checking visit again ek baar pta chal gya that curr. is not visited then proceed. But no, since there might be components of graph too.

we could've done nested(if) but(&&) is shortcut for it

if neigh is not visited than proced DFS & check if it returns true ∴ cycle exist → since uske aage wala parent to nhi hoga kabhi aur visited milega ∴ 100% cycle

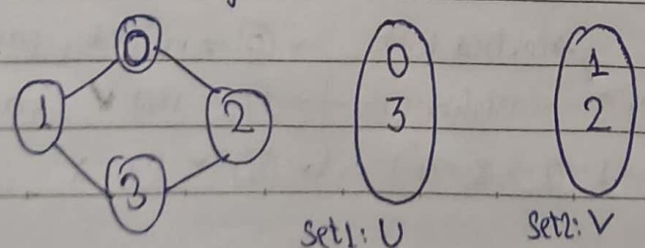
✓* Bipartite Graph

For every U and V set, the edge connecting two vertices then one of it's vertex if is in set U then second must be on set V. Never on same set.

⇒ i.e. Vertices connected through one edge should come in diff. sets.

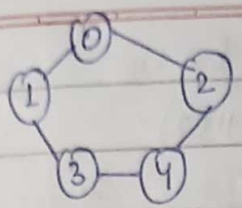
if its True for all vertices ∴ True or else false.

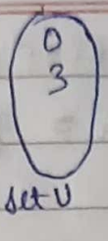
e.g.1

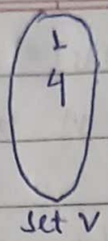


Covered all nodes & graph got separated in Two sets ∴ True.

eg 2







can't place (2) in any of sets
∴ Not Bipartite graph
false

How to check if given graph is Bipartite or not ?

=> Coloring method

→ start from one node & color it orange & its one of connected as blue then proceed from next edge till it reach to the same node then if final edge connected nodes are of diff. color ∴ True else false.

→ colors → -1 → no color
0 → yellow
1 → blue

→ BFS based.

→ 3 cases for one node → case 1: neigh → color → same ∴ false
case 2: " → " → diff. ∴ True
case 3: " → no color → assign opposite & proceed

Code → Define Edge class, create graph.

```
static boolean bipartite(ArrayList<Edge> [] graph){
    int[] color = new int[graph.length];
```

Initialise all with no color(-1) →

```
    for (i=0 to color.length){
        color[i] = -1;
    }
```

for BFS traversal →

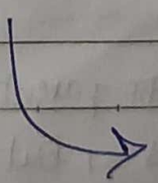
```
Queue<Integer> q = new LinkedList<>();
```

Travelling all subparts of graphs →

```
for (i=0 to graph.length){
```

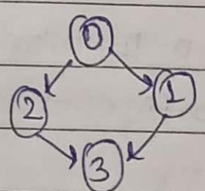
Start BFS traversal →

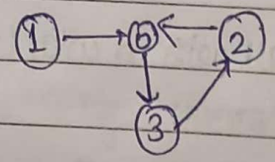
```
    if (color[i] == -1){
        q.add(i);
        color[i] = 0; → yellow to first
```



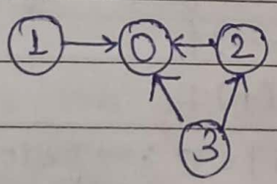
```
while( !q.isEmpty() ){
    int curr = q.poll();

Adding every neigh of current to queue → for(j=0 to graph[curr].size()){
        Edge e = graph[curr].get(j);
        // case-3 : No color
        if( color[e.dest] == -1){                    (you know this v. easy)
            int nextColor = col[curr]==0 ? 1 : 0;
            color[e.dest] = nextColor;   (put into next neighbour)
            q.add(e.dest);
        }
        // case-1 : same color
        ~~else if( col[e.dest] ==~~
        else if( color[curr] == color[e.dest]){
            return false;
        } // case-2 : continue
    }
    return true;
}
```

II. Directed Graph (DFS)

Not cycle → 

; cycle → 

Why DFS approach fails in directed graph which was true for undirected?



1. Starting Node = 0 with parent = -1, visited ✓
2. Check neighbours → None; Breaks
3. Starting Node = 1 with parent = -1, visited ✓
4. Neighbour → Node = 0 → already visited

Hence, since 1 का parent (-1) wasn't its next one + neigh is visited (0 is visited) ∴ cycle exists, But actually it does not ∴ fails.

∴ we will use modified DFS = DFS + stack (self made i.e. explicit)

↓

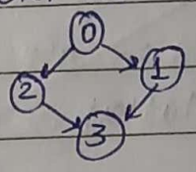
★ to check immediate neighbour in that particular call.

pseudo code:- (logic)

```
dfs(curr){
  vis[curr] = true;
  stack[curr] = true;
  for (all neigh) {
    if (stack[neigh] == true) {
      cycle exists ∴ True
    }
    if (!vis[neigh]) {
      isCycle(neigh);
    }
  stack[curr] = false;
  }
```

→ end me first falls karado.

Understand this thoroughly using DRY RUN.



code →

```
static boolean isCycle(ArrayList<Edge>[] graph){
  boolean[] vis = new boolean[graph.length];
  "  "  stack = "  "  "  "  ;
  for(i=0 to graph.length){
    if(!vis[i]){
      if(isCycleUtil(graph,i,vis,stack)){
        return true;
    }}}
  return false;
static boolean isCycleUtil(Arraylist..., int curr, boolean[]vis...){
  vis[curr] = true;
  stack[curr] = true;
  for(i=0 to graph[curr].size()){
    Edge e = graph[curr].get(i);
    if(stack[e.dest]){ return true;}
    if(!vis[e.dest] && isCycleUtil(graph,e.dest,vis,stack)){
      return true;
    }
  stack[curr] = false; → removing when return from call
  return false;
  }
```

immediate neigh exists ∴ CYCLE ←

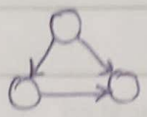
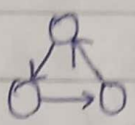
no parent ka scene since 'directed' me next neigh can't be parent

Page No.
Date 10-09-25

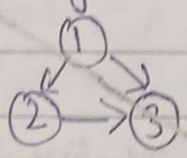
* **Topological sort** (DFS) — $O(V+E)$ = Time complexity

e.g. Leetcode Q. → Connecting Flights, Meeting Room

→ works for DAG (Directed acyclic graphs) only.

DAG =  ; NOT DAG = 

→ Order of Topological sorting is linear ~~of~~ order vertex $u \to v$, and in all edges order 'u' always before 'v'

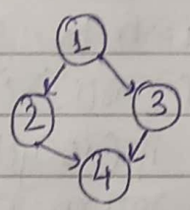
Matlab ⇒  = Topological sort order = 1 2 3

∴ for all u,v pairs, 1 always before 2,3. and 2 before 3 only.

All edge ⇒ 1→2, 2→3, 1→3 ✓ → This satisfies this

→ Works for Depedency graphs (items depending on each other)

e.g. 1. boil water
2. Add maggie
3. Add masala
4. Serve



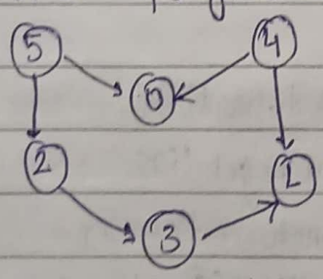
Topological sort order:

1 2 3 4 ✓
1 3 2 4 ✓ } any

wo chiz pehle hogi jisse aage koi aur depend krta hai

e.g. here 1 kisi pe depend nhi krta ∴ at first.

→ e.g. Find Topological sort



Step1: check DAG ✓

Step2:→ ans: 5 4 2 3 1 0

same for 1 ↑ ; same for 0 ie. 0 to 5,4.

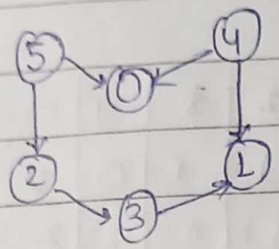
2 can only write this if 5 is done

3 only when if 2 have already occured anytime before

also, 4 5 2 3 1 0 Ans

→ Approach: modified DFS = DFS + Implicit stack

this will track kon pehle aani chahiye aur kon baad me



(curr from 0 to 5)

pehle ~~visited~~ aur neigh ko visit kar rahe hai fir aur saare uske dependent nodes add ho gaye then stack mein daalo in last.

⇒ dfs(graph, curr, vis, stack){
  vis[curr] = true;
  neigh → call(unvisited)
  stack.add(curr)
}

☆ Imp.

∴ Least dependent pehle ayega

∴ Final Ans = arr[i] = stack.pop();

Think Recursive-ly

Code →

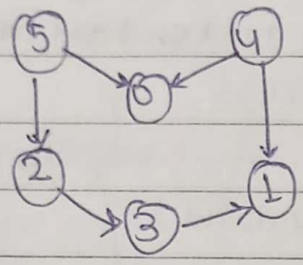
```
static void topSort(ArrayList<Edge>[] graph){
    boolean[] vis = ..... ✓
    Stack<Integer> s = new Stack<>();
    for(i=0 to graph.length){
        if(!vis[i]){
            topSortUtil(graph, i, vis, s);
        }
    }
    while(!s.isEmpty()){
        print(s.pop() + " ");
    }
}
static void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

for all connected components →

vibe DFS

Ajanta
Page No.
Date

* Topological sort using BFS → Kahn's Algorithm



Vertex =	0	1	2	3	4	5
(incoming)* indegree =	2	2	1	1	0	0
(outgoing) outdegree =	0	0	1	1	2	2

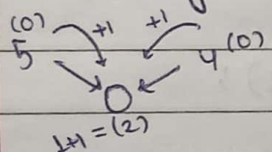
- Note ⇒ In any graph, for any longest path, the node from path starting has indegree = 0, and end of it's path has outdegree = 0.

  ∴ A DAG has atleast one vertex with indegree 0 & one vertex with outdegree 0.

- Hence, In topological sort those nodes comes first jiska indegree = 0.
- For destination, indegree + 1.

∴ Indegree →

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



- BFS uses queue (FIFO) ∴ store those element with least inorder first.
  1. First add those whose ~~inorder~~ degree = 0 value nodes in queue | 4 | 5 | | |
  2. Now first element will be removed and gets printed, while removing call for its neighbours.

Note:- Queue me ander sirf wohi element jayegi jiski indegree = 0.

∴ while removing (4) makes it's indegree = -1, so it's neighbour value gets reduced by 1.

∴ (4) gets removed and indegree of 0 & 1 became 2-1=1, 2-1=1 respectively

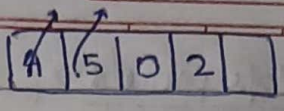
  3. Doing these if we didn't got any with indegree = 0 ∴ Now remove (5) and do the steps.

∴ indegree of 0 & 2 became → 1-1, 1-1 = 0,0 respectively

∴ add 0 and 2 in queue.

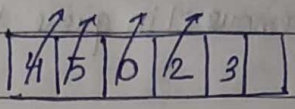
Ajanta
Page No.
Date 11-07-25

4. Now remove 0, print its value & check for neigh → No neigh ∴ proceed with 2.

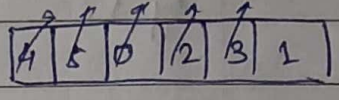


understood process

5. Remove 2, print, reduce indegree of 3 (neighbour) by 1 ∴ will become 1-1=0 ∴ add in queue (if become 0)



6. Remove 3, print, neigh = 1 → reduce its indegree by 1 ∴ 1-1=0 ∴ Now add in queue.



7. Remove 1, print, no neigh + all removed ∴ Aus.

Ans. = 4 5 0 2 3 1

Note :- Here we don't need to keep track of vis[] as to check ki wo visit hua tha ya nhi ∵ Dobara wo node kabhi nhi aayega in further steps.

Code ⇒ ① Calculate Indegree.

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

graph ke saare vertex pe jao use connected jo neighbours hai uske indegree ko badha do.

② Topsort

```
static void topSort(ArrayList<Edge>[] graph){
    int[] inDeg = new int[graph.length];
    calcIndeg(graph, inDeg); → saara Indeg ka value in inDeg.
    Queue<Integer> q = new LinkedList<>();

    for(i=0 to inDeg.length){
        if(inDeg[i]==0){
            q.add(i);
    }}
```

//bfs proceed →

```
while (!q.isEmpty()){
    int curr = q.remove;
    print ( curr + " ");
```

Read steps (remove el in queue & print it)

check for it's neigh and reduce by 1. ←

```
for ( i=0 to graph[curr].size(){
    Edge e = graph[curr].get(i);
    inDeg[e.dest] --;
```

add only those if inDeg=0 ←

```
    if ( inDeg[e.dest] == 0){
        q.add ( e.dest);
}}}
```

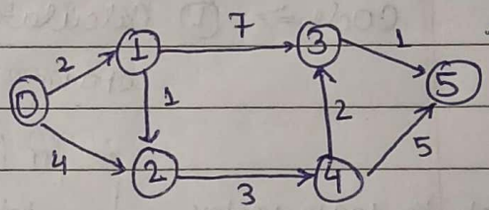
(cheapest Flights)

* Dijkstra's Algorithm O(nlogn)

↳ works for weighted undirected / Directed graphs

→ use to find shortest path from src to all vertices.

→ We'll be storing distance of all the nodes if starting from 0.

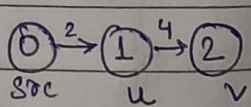


→ All those questions which asks for particular value, needs an optimised sol^n (greedy, dp)

→ ∴ Djikstra algo. is a greedy algo. in graphs. (Bellman Ford is DP algo.)

e.g. from src=0 to vertex=2 → min(2+1, 4) = 3 Ans.

Approach :→ 1. Initialise non-neigh of src as ∞.



2. main concept → (easy)

obv. from src

dist[u] + wt(u,v) < dist[v]

then dist[v] = dist[u] + wt(u,v) → got updated

eg. dist[u] = dist[1] = 2 , wt(u,v) = wt(1,2) = 4 , dist[2] = ∞

∴ 2+4 < ∞ ∴ update dist[2] = 2+4 = 6.

3. Solve this Priority Queue based approach. (some do with HashMap also)
(minHeap)

based on Dijkstra condition

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

pehle sab ∞ except 0 → 0(0) → put dist. of neigh 1(2), 2(4)
→ check their neigh, first ① → one is ③ and other is ②
∴ condition check for ② ⇒ 2+1 < 4 ∴ updated to 3.
Illy, " " " ③ ⇒ 2+7 < ∞ ∴ " " 9.
→ move forward, now check for 2 neigh. & proceed.

4. We've to store in pairs which have (node, dist)
↳ from which shortest can be obtained.

Pseudo code ⇒ 1. initialise dist.
2. PQ<Pair>(node, dist) pq
while (pq is not empty) {
visit → curr
if (vis[curr] == false) {
neighbours → dist[u] + wt(u,v) < dist[v]
update dist[v]
}

edge
src (u) dest (v)

code → ✓ (Define edge class with wt. also now, create Graph)

to store node and its shortest path ← static class Pair implements Comparable<Pair> {
int n;
int path;
const² of both

@Override
public int compareTo(Pair p2) {
return this.path - p2.path;
}
}

smaller path will be selected (path based sorting)

whenever we implement Comparable Interface then we need to Override our compareTo fnc. (∵ to define object kiske basis be compare hoga.)

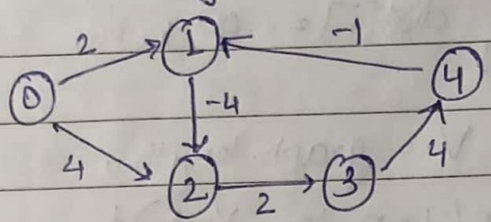
```
static void dijkstra(ArrayList<Edge>[] graph, int src){
    int[] dist = new int[graph.length];   // contains dist. from src to 'i'
    for(i=0 to graph.length){
        if(i != src){
            dist[i] = Integer.MAX_VALUE; // assigned ∞ except src, to all nodes.
        }
    }
    boolean[] vis = new boolean[graph.length];
    PriorityQueue<Pair> pq = new PriorityQueue<>();
    pq.add(new Pair(src, 0));   // (add first path 0→0(0))
                                //   Node  path distance
    while(!pq.isEmpty()){
        Pair curr = pq.poll();   // shortest distance wala gets removed from queue
                                 // tells about pair → tells about node
        if(!vis[curr.n]){
            vis[curr.n] = true;
            for(i=0 to graph[curr.n].size()){   // neighbours check →
                Edge e = graph[curr.n].get(i);
                int u = e.src;
                int v = e.dest;
                int w = e.wt;
                if(dist[u] + w < dist[v]){   // main condition →
                    dist[v] = dist[u] + w;
                    pq.add(new Pair(v, dist[v]));   // pair gets updated →
                }
            }
        }
    }
    for(i=0 to dist.length){   // print all shortest dist. from src →
        print(dist[i] + " ");
    }
```

* Check code for all paths from src to target. (Github Now)

* **Bellman Ford Algorithm** $O(V*E)$

→ Shortest path from src to all vertices majority for Negative edges, as Djikstra doesn't guarentee for negative edges.

→ Higher Time Complexity than Djikstra.



V = 5
E = 6

→ Perform this operation V-1 times : for (i = 0 to V-1){
for all edges (u → v)
if (dist[u] + wt < dist[v])
dist[v] = dist[u] + wt ;
}

→ Vertices →	0	1	2	3	4
(V-1) iterations ↓	0	∞	∞	∞	∞
1	0	2	-2	0	4
2	0	2	-2	0	4
3	0	2	-2	0	4
4	0	2	-2	0	4

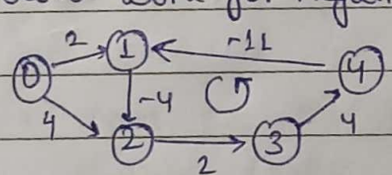
→ all 3 same since all nodes are connected.
↓
Jaha pe ek iteration me mil gya but willl've to perform V-1 iterations.

distances

→ Last iteration value will be final ans.

→ NOTE :- Does'nt work for negative weight cycles.

One line explain in next page



weight of cycle = -4 - 11 + 4 + 2 = -9
∴ Negative

wont work
↓
since, using this (1) ki dist. -7 ho jayega from 2. ↓

This should not be logically correct ∵ aapke path mein ek baar (1) aa gya, then aap first jab (1) pe pahuche to wo kam tha.

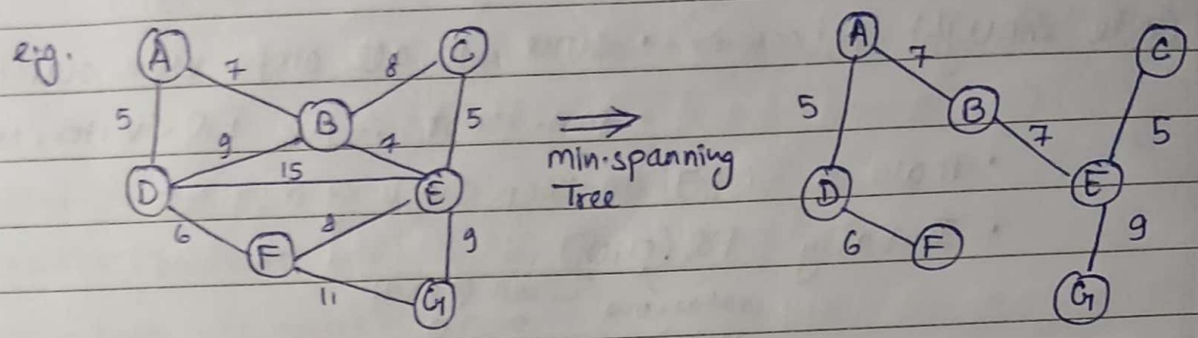
ie. for -7 path should be → (0) →[2] (1) →[-2] (2) →[0] (3) →[4] (4) →[-7] (1) ∴ NOT PRACTICAL

☆ (1) pe already pahuch gye then wapas ghum ke first (1) pe aana kaha se shortest path hua.

```
Code →  static void bellmanFord ( graph , int src){
        int[] dist = new int [ graph.length ];
assign ∞ →  for ( i=0 to dist.length){
              if ( i != src) {
                dist[i] = ∞ ;
            } }
            int V = graph.length ;
No. of iterations req. →  for ( i=0 to V-1 ){
to get correct ans
1 iteration ka traversing → for ( j=0 to graph.length ) {

Travel all neigh of nodes →  for ( k=0 to graph[j].size() ){
                Edge e= graph[j].get(k);
                int u= e.src;
                int v= e.dest;
                int w= e.wt;
∵ pehle 'u' to determine ho.  if( dist[u]!=∞ && dist[u]+w < dist[v]){
                dist[v] = dist[u] +w ;
            } } }
            Print dist[i] for all nodes.
```

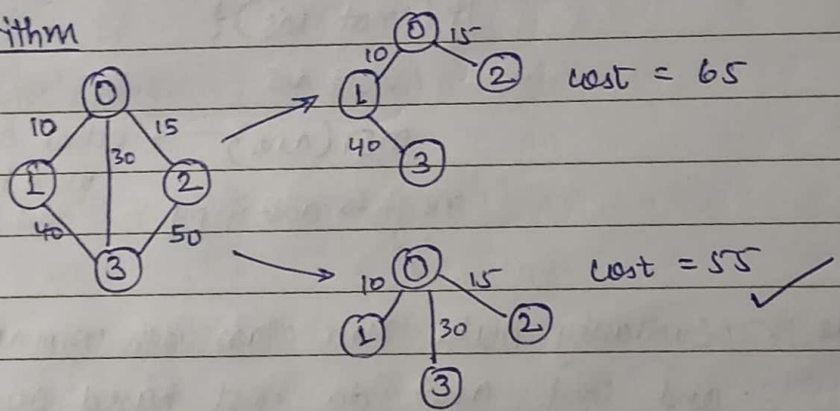
* Minimum Spanning Tree

→ also called as min. wt. spanning tree which is a subset of graph connecting all nodes with min. wts. only.

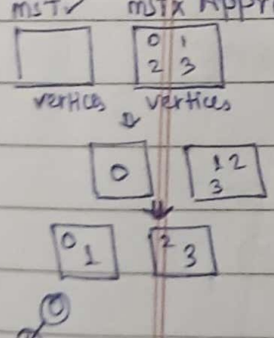


→ overall cost of a vertex should be min. not necessary for a single edge only (it may hamper other node).
i.e.
→ overall cost of a graph should be min. connecting all nodes, as subset.

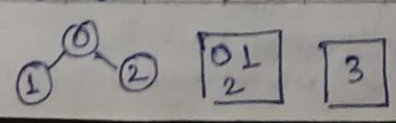
* Prim's Algorithm



MST ✓ MST ✗ Approach: 1. Make one MST and one not in MST set that contains vertices. (initially all in "not in MST set", tracked by vis[])

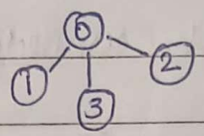


2. ~~Check for those edges~~ Pick any one node add in MST set.
3. " " " " going from MST → Not MST vertex.
4. Add that vertex in MST set whose edge cost in min. (Till this step make 0 and 1 as true)
5. Now from MST set (0,1) check all edges from 0 and 1 in "not MST set" and add min. one. $0 \xrightarrow{15} 2$ ✓, $0 \xrightarrow{30} 3$ ✗, $1 \xrightarrow{40} 3$ ✗ ∴ (2) is added in MST



6. Must keep a track kis node ke edge se aaya hai min. (must be subset of graph (subgraph)). Proceed further...

7. Then obtain the tree from MST set ⇒ MST ✓



- Code thought process ⇒ • since, in all steps we're searching for min. cost ∴ use PQ<vertex, cost> → min Pair
  - make a vis[] to keep track of MST set
  - Initially PQ(0,0) ← add any one vertex; ← cost of itself only = 0

- Pseudo code ⇒

```
vis[]
PQ < vertex , cost>
PQ(0,0)
while (!pq.isEmpty()){
  curr → (v, cost) ie. visit and get cost
  if ( not vis){
    visit ✓
    MST(ans) → either edge based or cost based (we're taking cost but edge bhi ques. me aa skta hai)
  neigh → add in pq
  }}
```

- code ⇒ Similarly make Pair class with comparable for vertex (v) and cost, and do cost based sorting.

```
static void prims (graph){
    boolean[] vis = ✓
    Priority Queue < Pair> pq = new PQ<>(); ✓
                                int finalCost = 0;
add any one node with its own → ~~pq.add(0,0);~~
cost zero                       pq.add(new Pair(0,0));
```

Linked list me Node ko kya direct use karte the!
Node root = new Node(val);

graphs → trie → Bit Ma. → Backtracking.

Ajanta
Page No.
Date

```
while ( ! pq.isEmpty()){
    Pair curr = pq.remove();
    if ( ! vis[curr.v] ){
        vis[curr.v] = true;
        finalCost += curr.cost;
        for( i=0 to graph[curr.v].size(){
            Edge e = graph[curr.v].get(i);
            pq.add(new Pair(e.dest , e.wt));
        }}
```

min. element le liya as curr (with low cost) ←

☆ add neigh in pq for considering low cost which will be curr and added in MST →

print (finalCost); Ans.

Hw. write code to print edges also with min. cost.

***Q. Cheapest flights within K stops, more ques. after theory completion. flood fill, etc.

12-07-25

* Disjoint Set Union

→ used for cycle detection, Kruskal's algo. (MST).

eg. n = 8

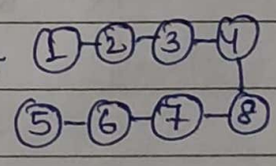
set1 : (1)–(2)–(3)–(4)

set2 : (5)–(6)–(7)–(8)

find(2) → set 1 (gives which set element belongs to).

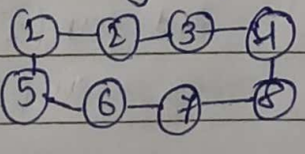
find(5) → set2

union(4,8) → connect(4,8), Now all elements in one set



if again union in same set = cycle formed.

eg. union(1,5) →



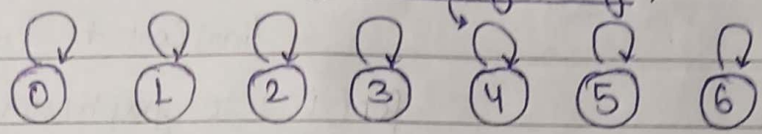
find(3) = set1

find(7) = set1.

easy example

- Approach :→ 1. find → leader info., union → 2 groups join (via leader-leader)

2. Initialize on parent array and rank array.
Initially all nodes are parent of itself, rank = 0.

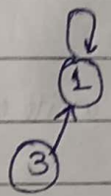
n = 7 

∴ par[] = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
0 1 2 3 4 5 6

∴ rank[] = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | → Height
0 1 2 3 4 5 6

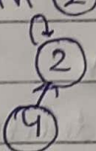
3. Perform operations :-

a. union (1,3) → connect (3) with (1).
∴ parent of (3) updated to 1
rank of (1) " " +1.

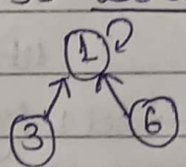


b. find(3) → gives parent(3) = 1 ans.

c. union (2,4) → connect (4) with (2).
∴ par(4) = 2
rank (2) = +1

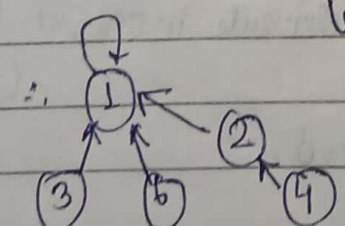


d. union (3,6) → connect (6) with 3's leader → top node (par=0)
∴ par(6) = 1
rank(1) = +1+1 = 2 ✗
→ rank will remain 1 ∵ it is Height.



e. union (1,4) → connect (4)'s leader with (1) or (1) with 4's leader. Since both have same ranks i.e 1.

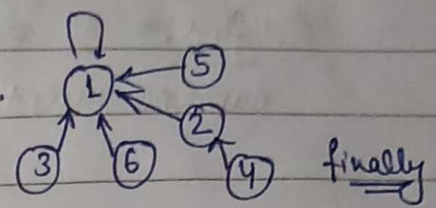
∴



Now rank of (1) = 1+1 = 2.
and par(2) ~~= par(4)~~ = 1.
par(4) = 2.

f. find(3) = 1.

g. union(1,5) → connect (5) with (1). ∴ par(5) = 1



finally.

Till now par[] =

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

rank[] =

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

- Pseudo code → ~~find~~ int n, par[n], rank[n]

initially par[i] = i, ~~rank[i] = 0~~.

```
find(x)
  if (x == par[x]){
    return x;
  return find(par[x]);
```

find()

union(a,b)

get → par(A), par(B) ✓

↳ using → find(a), find(b)

if (rank of parA == parB) {

connect anyone → par[parA] = parB;

rank[parB] ++;

Ily, if (rank of parA < parB rank) {

connect A with B ~~increase~~ not rank

★ update :: koi chota height wala aake judega to rank ko farak nhi padega (until their rank is same)

Ily if (rank parA > rank parB) {

Connect B with A .

}

- Code → Here, no needy created graph already.

Complete code → public class disjointSet {

static int n = 7;

static int[] par = new int[n];

static " rank = " ;

rather writing static we could've declared & initialised it in psvm (fatto ka tez banna).

initialise parent (cuz rank is already 0 using array property) → public static void initialise () {

for (i=0 ton) {

par[i] = i; (self parent at start)

}

find:→ public static int find(int x){

supreme parent → if(x == par[x]){

return x;

}

Khojo till parent ka parent ≠ → return par[x] = find(par[x]);

until supreme parent reached

union → static void union(int a, int b){

get both parents ⇒ int parA = find(a);

int parB = find(b);

if(rank[parA] == rank[parB]){

connect anyone → par[parA] = par[B];

rank[B]++;

}

else if(" < "){

A will connect to B → par[parA] = par[B];

}

else{

par[parB] = par[A];

}

psvm → write yourself ~~first~~ initialise();

* Kruskal's Algorithm O(V+ElogE)

↳ used to find MST (greedy approach)

(Easy) Approach: → 1. Sort edges

2. Take min cost edge

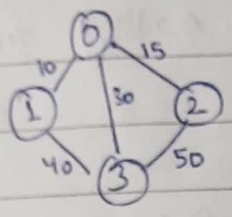
↓

such that it does not form cycle.

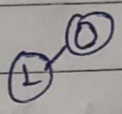
Include in ans.

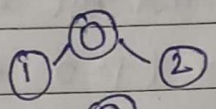
Ajanta
Page No.
Date

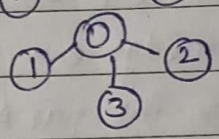
eg.



	Sorted edges	weight
a.	(0,1)	10
b	(0,2)	15
c.	(0,3)	30
d.	(1,3)	40
e.	(2,3)	50

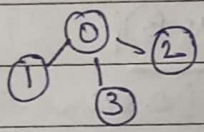
for a). Select edges in ans and will form => 

b.) (0,2) ko select karo => 

c) (0,3) " " " => 

d). ☆ (1,3) → ko select kiye to will form cycle ∴ Don't include.

e). (2,3) → " " " " " " " " ∴ " " .

final MST =>  Ans

Pseudocode => ArrayList<Edge> → collections.sort() basis of wt.

(V-1) times to count MST in V vertex connections we needs (V-1) edges.

→ for(i=0 to V-1) {

find union to check cycle exist
i.e. union (a, b) then if parA & parB are in same set ∴ cycle ✓ ∴ Don't include

Edge e → a (src)
→ b (dest)

parA, parB
same set → don't include ∴ cycle
diff set → include → union(parA, parB).

}

check code

ArrayList<Edge> edges
(Arraylist of type Edge).

Ajanta
Page No.
Date

code → create ~~graph~~ edge, create Edge class with wt. sorting and write code for disjoint sets.

```
                                    edges
static void kruskals(~~graph~~, int V){
parents initialise → initialise();
sort edges →  Collections.sort(edges);

    int mstCost = 0;
    for(i=0 to V-1){
      Edge e = edges.get(i);
      int parA = find(e.src);
      int parB = find(e.dest);

if they don't belong to same → if(parA != parB){
set                              union(e.src, e.dest);
                                 mstCost += e.wt;
    }}
    print(mstCost);
```

Watch 9. videos of Graph part-5.