DAY 129 DSA

GRAPHS STARTED

Date → 4 Feb, 22
Day → Friday

What is a Graph data structure?

→ A graph is a set of vertices and edges.

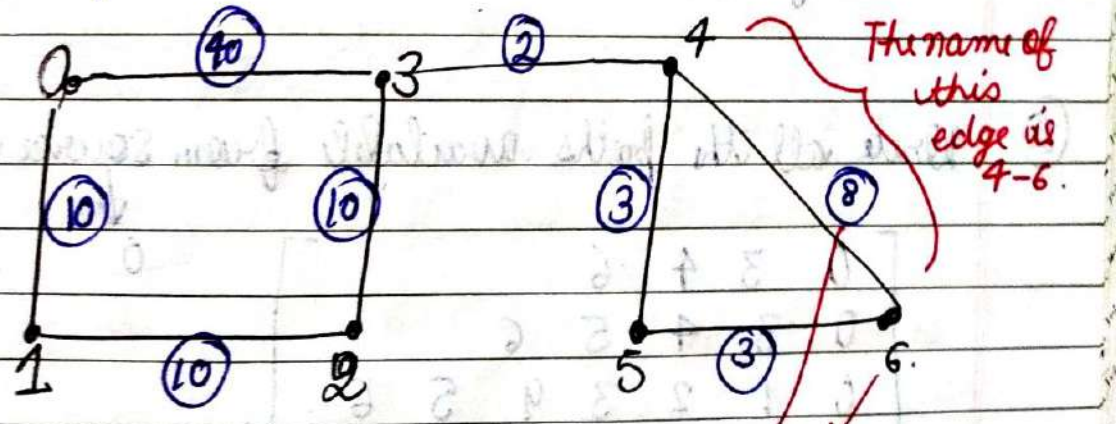
(एक Graph में vertices (0) होती हैं और edges (—) होती हैं)

Vertex में हम data store करते हैं.

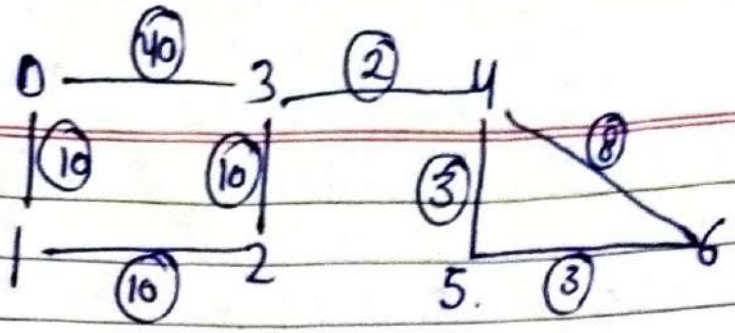
Vertices के नाम भी होते हैं. eg → 0, 1, 6, 9, etc

edges के भी नाम होते हैं eg → 0-1, 4-5, etc

edges पर weight भी हो सकता है.



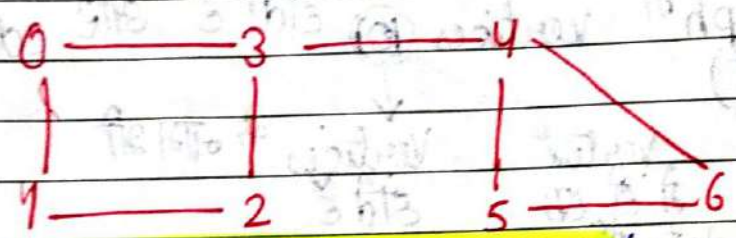
What sort of data can be stored using Graph



eg-1) यहाँ 0, 1, 2, 3, 4, 5, 6 (vertex) cities हो सकती हैं और edges हो सकती हैं उनके बीच की roads/path

eg-2) Vertex → Computers
Edges → LAN wires b/w computer

Some Common Questions that can be asked in this Graph



① Is there any path from 0 to 6 (Vertex 0 से 6 तक Path है).
source destination
✓ SO yes, there is Path available

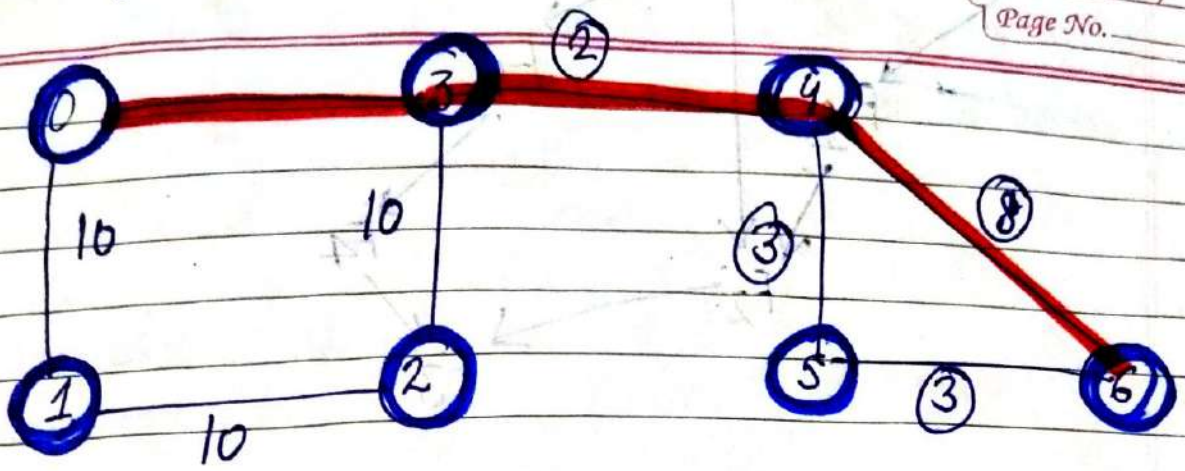
② Write all the paths available from source to destination
0 6

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

These are the paths from source to destination.

③ Write the smallest path available in the terms of edges from Source (0) to destination (6).

Date / /
Page No.

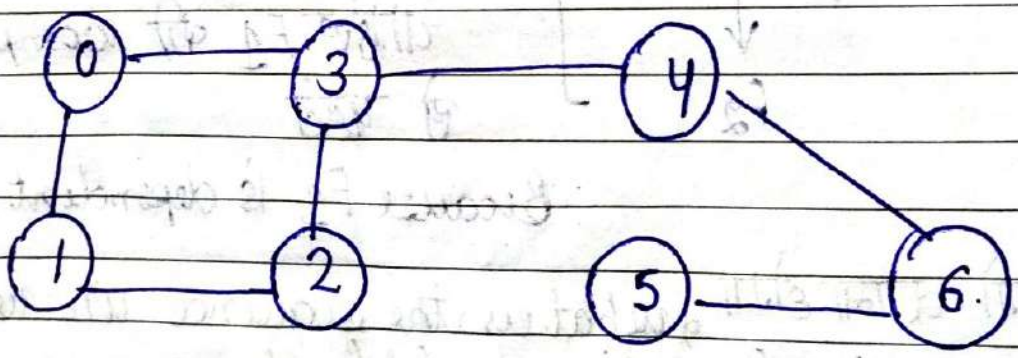


[0 3 4 6] is the smallest path available from source to destination in terms of edges.

edges की terms में smallest path find करने के लिए हम DFS use करते हैं।

④ Write the smallest path available in the terms of weight (smallest path to travel in kms

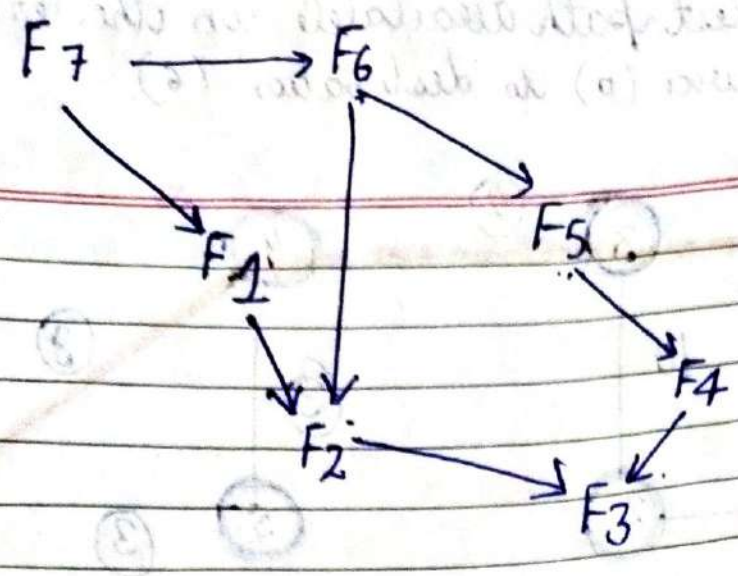
Vertex → Stops or Cities
Edges → Roads
Weights → No. of kms



[0 1 2 3 4 5 6] वाला path weight की terms में सबसे छोटा है।

weight की terms में smallest path find करने के लिए Dijkstra algorithm को use करते हैं।

(5)



F1, F2, F3, F4, F5, F6, F7

ये सब files हैं.

* इनमे से कुछ files दूसरी files पर dependent हैं. compile होने के लिए

eg → जब F2 compile होनी होगी तब भी F1 compile हो पाएगी

F1 ↓ F2 } Means F2 compile होनी चाहिए F1 की compilation से पहले

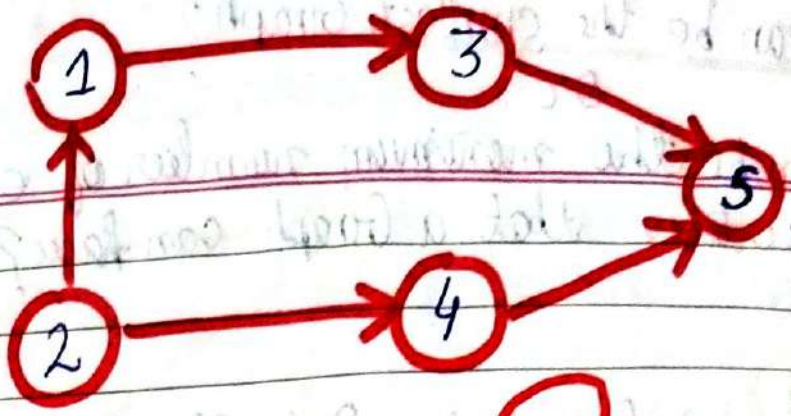
Because F1 is dependent on F2.

तो हमे बताना होगा, what is the sequence in which the compilation ~~the~~ should be done?

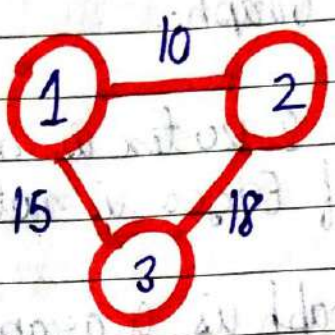
→ F3 F2 F4 F1 F5 F6 F7

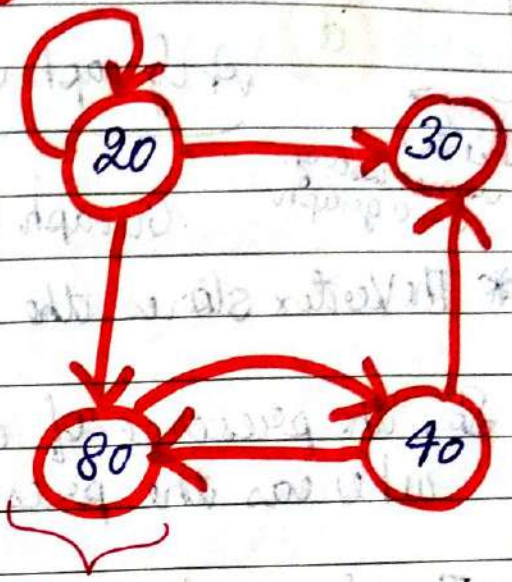
इस question को बोलते हैं Topological Sort. इसतरह की problem को solve करने के लिए भी data structure हम use कर सकते हैं।

Graph



So, Graph is a set of vertices and Edges.





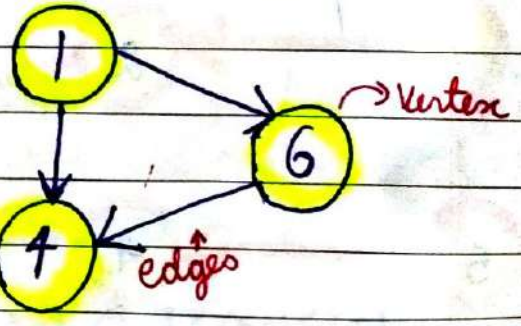
These circles in which the data is stored are the vertices of Graph

1. → Graph is a set of Vertices and Edges

$$G_1 = (V, E)$$

↳ So, we can define the graph as $G = (V, E)$ → E is the set of edges.
↳ Here V is the set of vertices

2. → In these Graphs, the circles in which the data is stored are the Vertices of graph and the lines that are connecting them together are called Edges of Graph



(3) What can be the smallest Graph?

Or

What can be the minimum number of edges and vertices that a Graph can have?

This is a Graph with 1 Vertex, Graph with only 1 vertex is also called Trivial Graph

(d) A Graph is a Data Structure (It is a way of storing data).

Graph में data को store करने के लिए V

→ At least 1 Vertex is necessary in a graph

* The Vertex stores the data in Graph.

So, the presence of at least 1 vertex is necessary whereas the presence of Edges is not.

Therefore, the Smallest Graph is a graph with 1 Vertex and 0 edge.

So, In $G = (V, E)$ → But, the set E can be empty.

↓

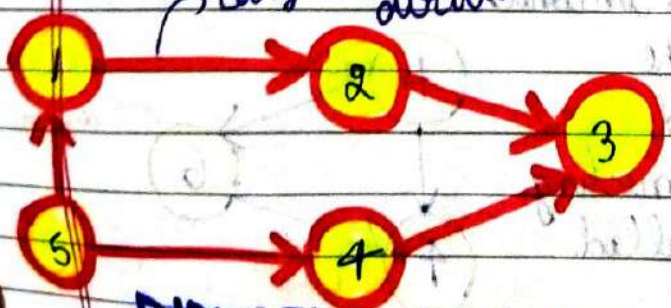
The Set V of vertices of a Graph G can not be empty

(4) Types of Graph

a)

DIRECTED GRAPH → If the edges of a graph has the directions, then the graph is known as the Directed Graph.

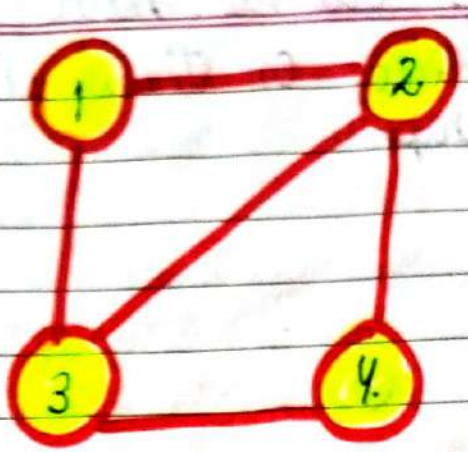
Edges have direction



DIRECTED GRAPH.

NON-DIRECTED GRAPH

Date / /
Page No.



A graph whose edges do not have any direction is called Non-directed/ Undirected Graph.

They are assumed to be Bi-directional.

Non-Directed Graph

→ यहाँ हम directions खुद ही assume कर सकते हैं।

1 → 2 and 2 → 1

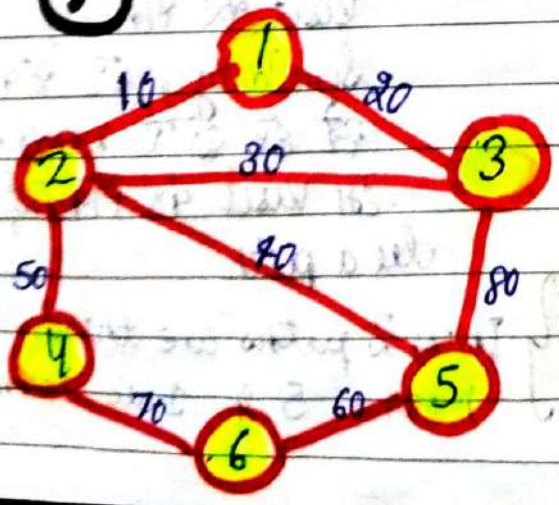
These both directions are correct for Non-Directed Graph we assume इन Edges दोनों direction में point कर सकती हैं।

The two vertices connected to each other via an edge are called Adjacent Vertices

10 and 20 are adjacent Vertices

Weighted Graph

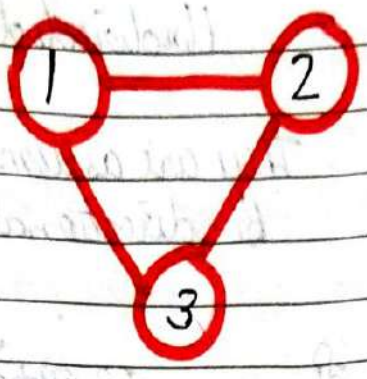
A graph whose edges have weights mentioned on them are called as a Weighted Graph.



The edge 1 → 2 has a weight 10. This means the cost of travelling from 1 → 2 or 2 → 1 is both 10.

d) NON-Weighted Graph

A graph whose edges do not have any weights mentioned on them is called a Non-weighted Graph.



e) Connected Graph

The graph in which from one node we can visit any other node in the graph is known as a Connected Graph.

Node also means vertices

इस Graph में हर Pair of Vertices में atleast 1 Path तो होगा ही, i.e. it is a Connected Graph.

Connected Graph

जिस Graph में, हर एक node से हर दूसरी node को visit किया जा सकता है, उस Graph को हम Connected Graph बोलते हैं।

eg → 1 से हम 4, 5, 3 को visit कर सकते हैं।

Similarly, हर एक node से हर दूसरी node को हम visit कर सकते हैं, i.e. there can be a path

Source → 5
Destination → 1

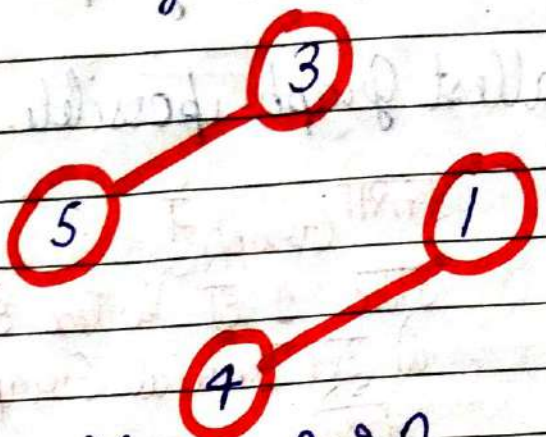
path → 5 3 1
5 4 1

} These 2 paths are there Means 5 & 1 is connected

6 DISCONNECTED GRAPH

Date / /
Page No.

The graph in which atleast one node is not reachable from a node is called Disconnected Graph

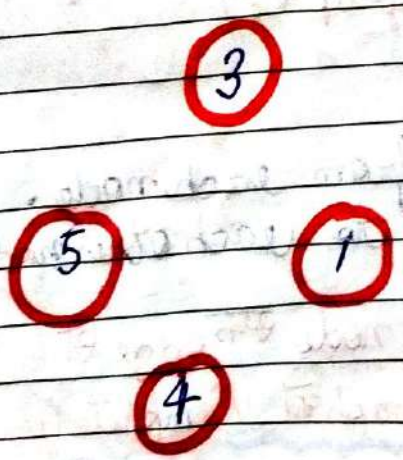


Graph में अगर कोई एक भी ऐसी node (Vertex) है जिसे हम किसी और node (vertex) से reach नहीं करा सकते तो वो graph Disconnected graph बोला जाएगा.

Eg → 5 से 1 तक का कोई Path नहीं है.
So, 5 से 1 reachable नहीं है.
So, this is a disconnected graph.

7 NULL GRAPH → A graph whose edge set is empty.

A graph is known as a null graph if there are no edges in the graph.



इस Graph में 4 Vertex है & 0 edges.
So, This graph can be said as the Null Graph.

h) Trivial Graph

Graph having only a single vertex, it is called a Trivial Graph.

It is the smallest Graph possible.

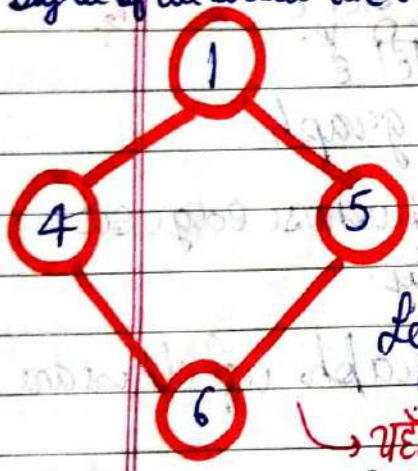
किसी Graph में अगर 1 ही Vertex हो तो उसे Trivial Graph कहेंगे।

Trivial Graph.

i) Regular Graph

Degree of all Vertices are equal

The graph in which the degree of every vertex is equal to the degree of every other vertex in the Graph, such type of Graph are said Regular Graph.



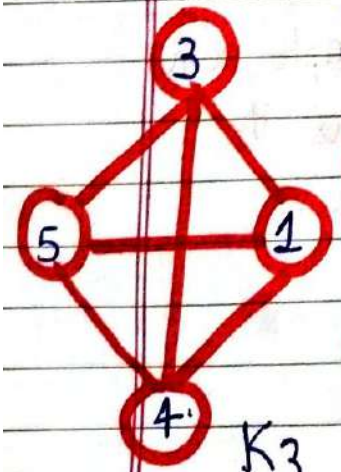
Degree of a vertex
↳ The no. of edges the vertex is connected to.

Let the degree of each vertex is k, then the Graph is called k-regular.
↳ यहाँ हर vertex की degree 2 है तो यह 2-Regular Graph है।

j) Complete Graph

The graph in which, from each node, there is an edge to each other node.

जिस graph में graph की हर node से हर दूसरी node एक edge से connected है, उस Graph को Complete Graph बोलेंगे।



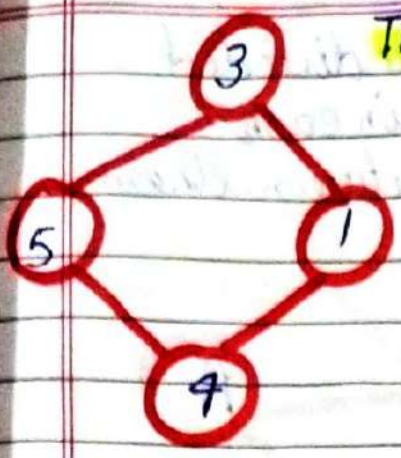
K_3

A complete graph of n vertices contain exactly ${}^nC_2 = \frac{n}{(n-2)\lfloor 2} = \frac{4\times3\times2}{2\times2} = 6$ edges

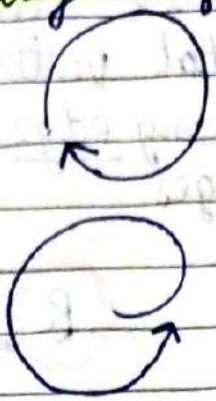
k) Cycle Graph

→ The graph in which the graph is itself a cycle is called the Cycle Graph.

Date / /
Page No.



The degree of each vertex is 2.



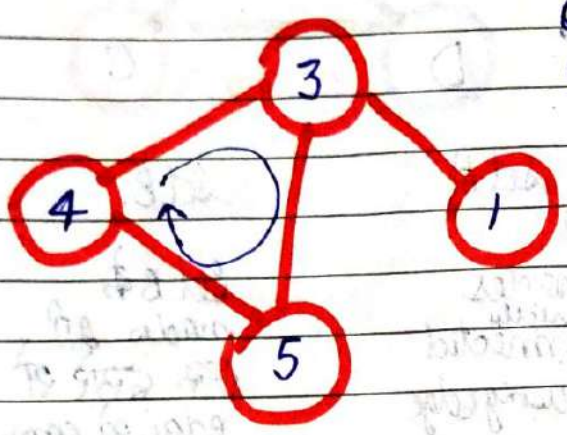
This Graph is itself a cycle तो इसे Cycle Graph बोलेंगे because हर node की degree 2 है।

degree of 1 → 2
degree of 3 → 2
degree of 4 → 2
degree of 5 → 2.

l) Cyclic Graph

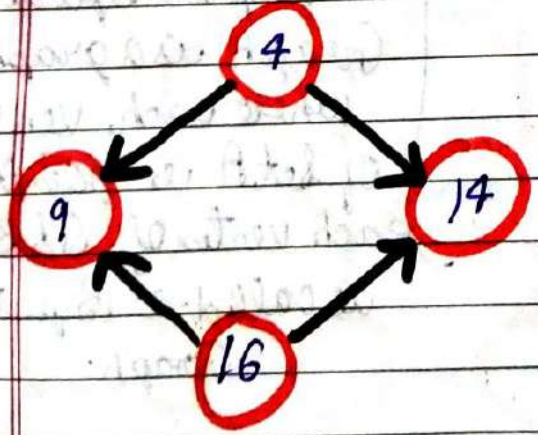
The graph containing atleast one cycle is known as Cyclic Graph.

Graph में अगर कोई 1 भी nodes की cycle बन रही है तो उस Graph को हम Cyclic Graph बोल सकते है।



m) Directed Acyclic Graph

A directed graph that does not contain any cycle in it is called the Directed Acyclic Graph



ये एक directed graph है क्योंकि इस graph में edges have directions.

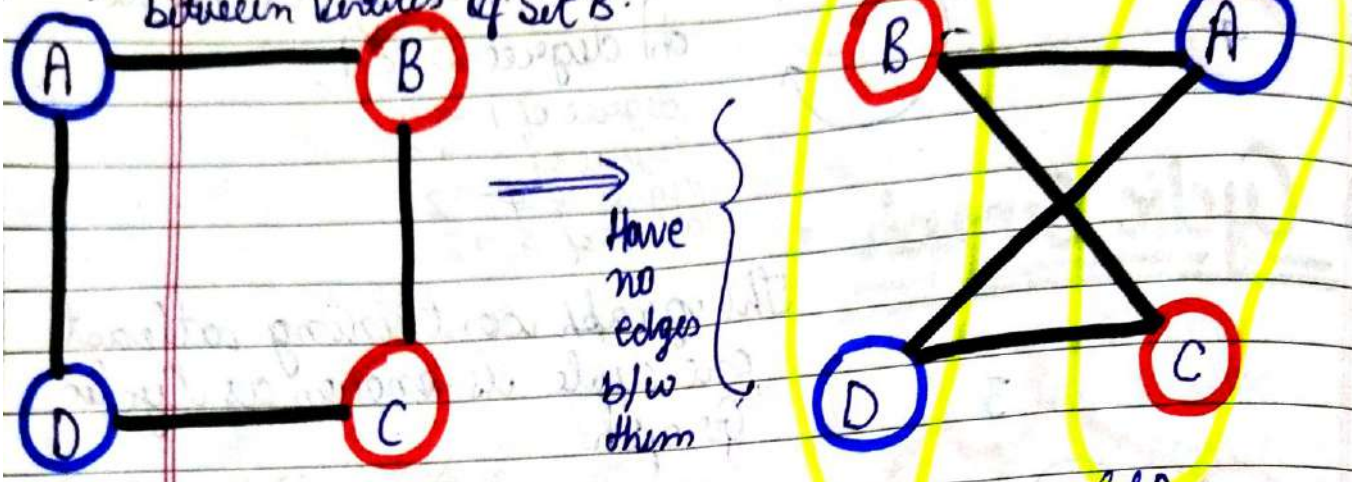
और इस Graph में directionally कोई cycle नहीं बन रही है तो इसे directed acyclic graph बोलेंगे।

7) BIPARTITE GRAPH

Date
Page No.

1) A graph in which vertex can be divided into 2 sets such that vertex in each set does not have any edge between them

2) The Vertices of Set A only have edges between Vertices of Set B.

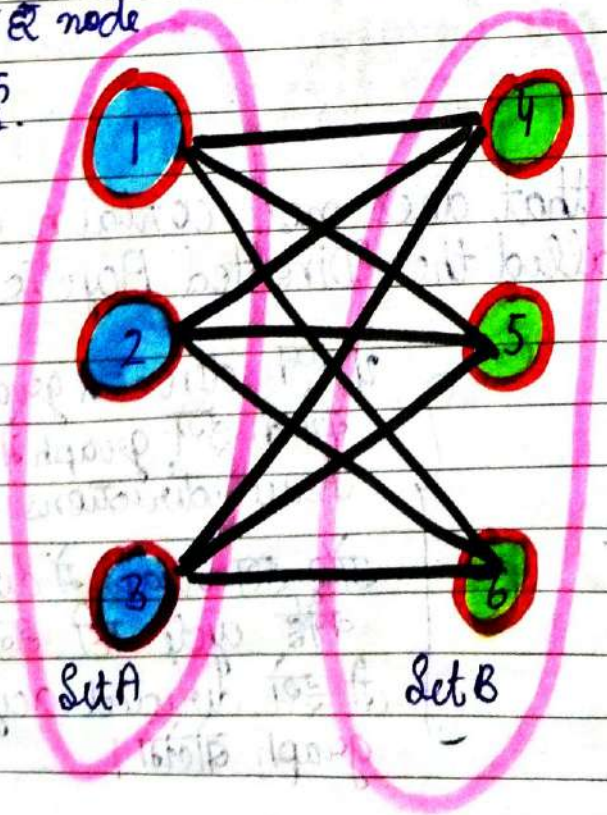


3) None of the Vertices belonging Set A to same Set join each other.

Set A के nodes आपस में directly connected नहीं होंगे using edge

Set B के nodes भी एक दूसरे से edge से connected नहीं हैं।

Set A के हर node से Set B की हर node के बीच में एक edge है तो ये एक Complete Bipartite Graph है।

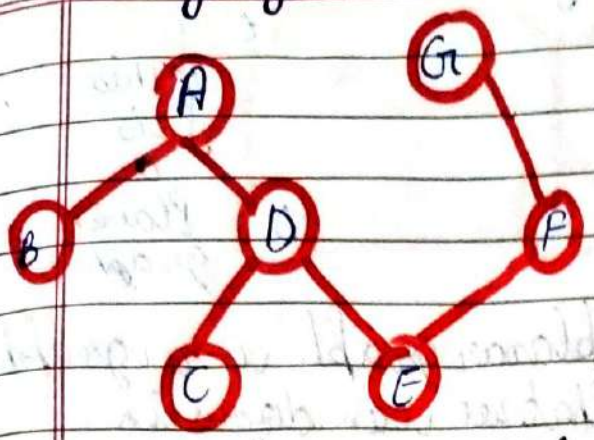


This is a Complete Bipartite Graph,

A Complete Bipartite Graph is a graph where each vertex of Set A is joined to each vertex of Set B is called a Bipartite Graph.

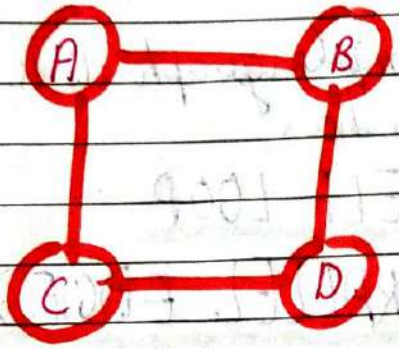
o) **Acyclic graph** → A graph which is not containing any cycles in it is called an acyclic graph

Date / /
Page No

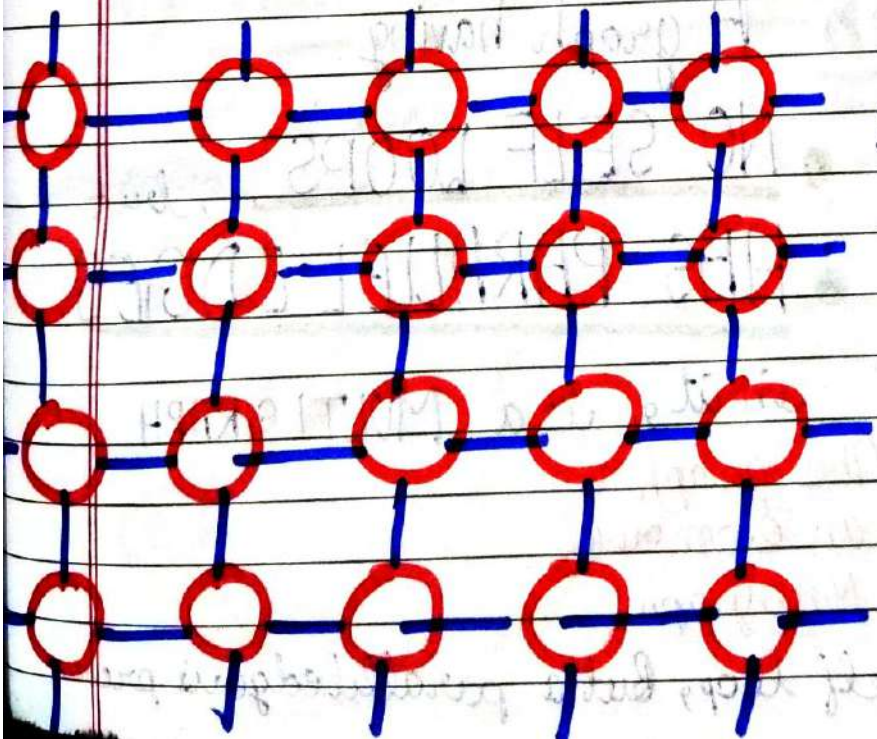


इस graph में nodes के बीच कोई cycle नहीं बन रही है, so this graph is a Acyclic graph.

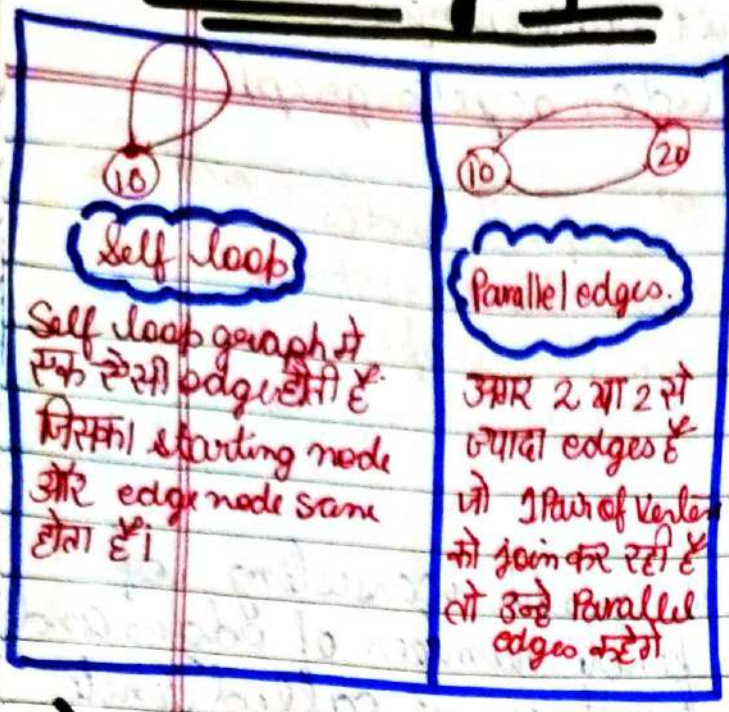
p) **Finite graph** → A graph consisting of finite number of Edges and vertices is called Finite graph.

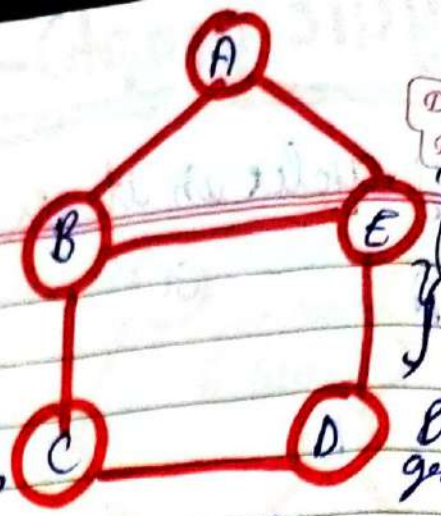


q) **Infinite graph** → A graph consisting of infinite number of edges and vertices is called a Infinite graph.



r) Planar graph





This is a Planar graph

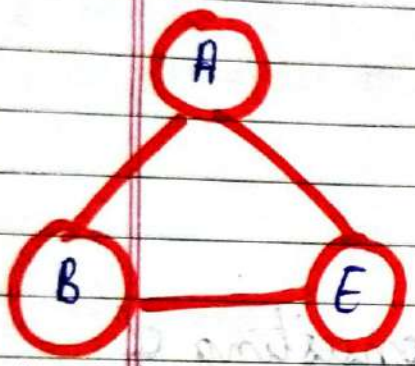
A planar graph is a graph that we can draw in a plane such that no 2 edges of it can cross each other.

s) Simple Graph

→ A Simple graph is a graph which has

- NO SELF LOOP
- NO PARALLEL EDGES

in it.



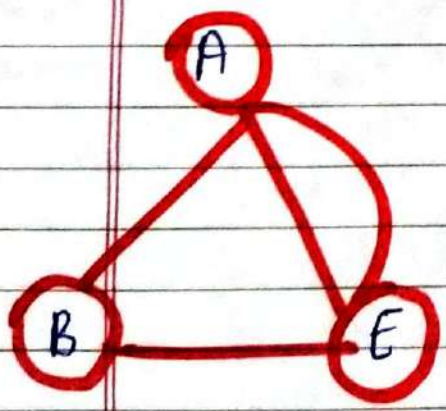
Example of Simple Graph.

t) Multi Graph

A graph having.

- NO SELF LOOPS, but
- HAS PARALLEL EDGES

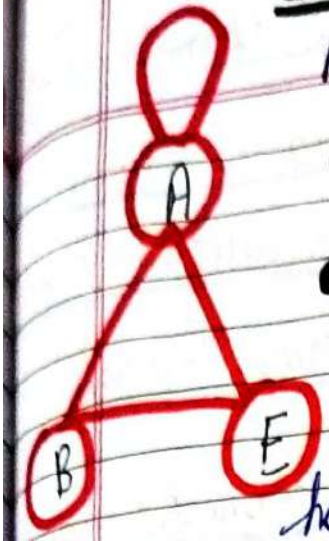
in it, is a MULTI GRAPH



This graph is an example of Multi Graph

↳ There is no self loop, but a parallel edge is present

u) Pseudo Graph



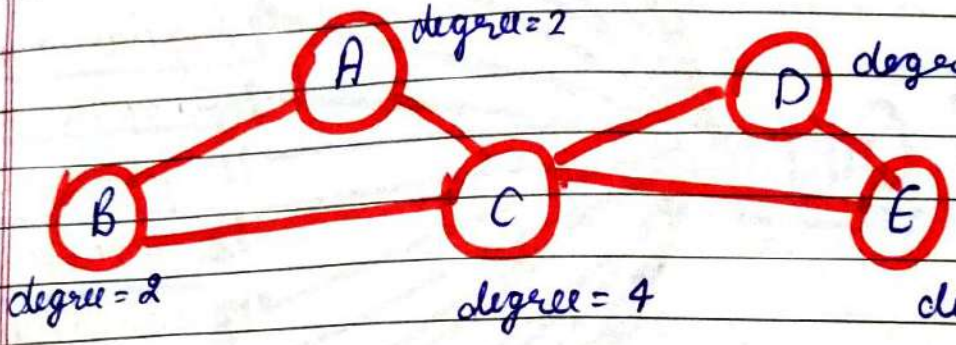
A graph having
- NO PARALLEL EDGE
- HAS SELF LOOPS.

in it, is a Pseudo Graph

has no parallel edge, but has / self loop.

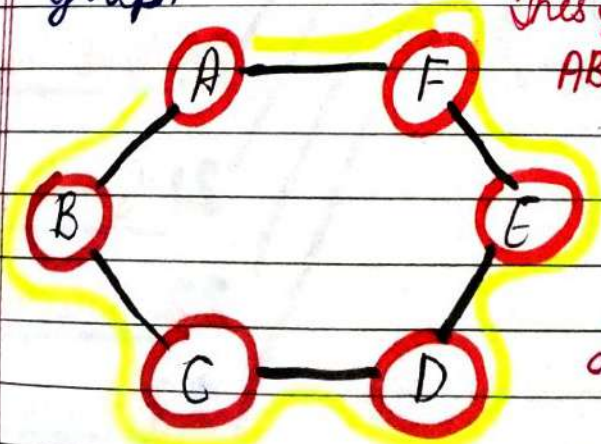
v) Euler Graph

→ A Euler graph is a graph in which all the vertices are of even degree



w) Hamiltonian Graph

→ If there exists a closed walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges, then such a graph is called Hamiltonian graph



This graph contains a closed walk ABCDEF (except starting vertex)

All the vertex are visited without repeating the edges

∴ This is Hamiltonian Graph.

	Self loop(s)	Parallel Edge(s)
Graph	Yes	Yes
Simple Graph	No	No
MultiGraph	No	Yes
Pseudo Graph	Yes	No

Types of Graph

1. Directed Graph
2. Non-directed Graph
3. Weighted Graph
4. Non-weighted Graph
5. Connected Graph
6. Disconnected Graph
7. Null Graph
8. Trivial Graph
9. Regular Graph
10. Complete Graph
11. Cycle Graph
12. Cyclic Graph
13. Directed Acyclic Graph
14. Bipartite Graph
15. Acyclic Graph
16. Finite Graph
17. Infinite Graph
18. Planar Graph
19. Simple Graph
20. Multi Graph
21. Pseudo Graph
22. Euler Graph
23. Hamiltonian Graph

(5) COMPONENTS OF GRAPH

Date / /
Page No.

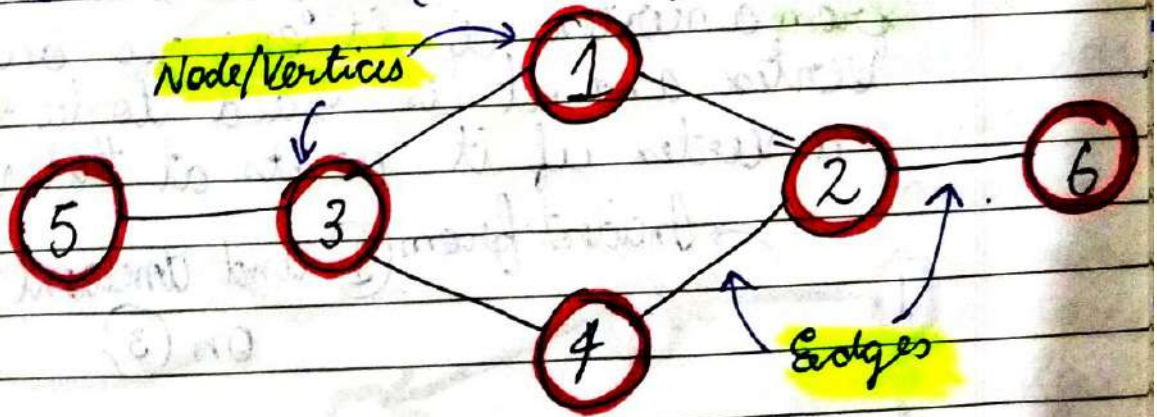
A graph is a non-linear data structure consisting of nodes and edges

The Nodes are also referred to as vertices and the Edges are lines or arcs that connect any 2 nodes in a graph.

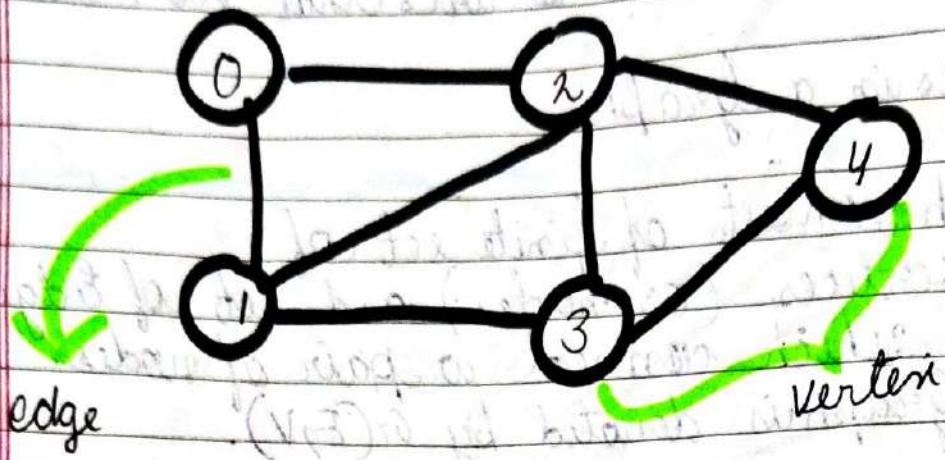
A graph consists of finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.
The Graph is denoted by G(E,V).

There are 2 components of a graph.

(1) Vertices → Vertices are the fundamental units of a graph.
Sometimes, vertices are also known as Vertex or nodes.
Every node/vertex can be labelled (given a name) or unlabelled (or no name given).



(2) Edges → Edges are used to connect 2 nodes of the Graph. It can be ordered pair of nodes in a directed graph. Edges can connect any 2 node in any possible way



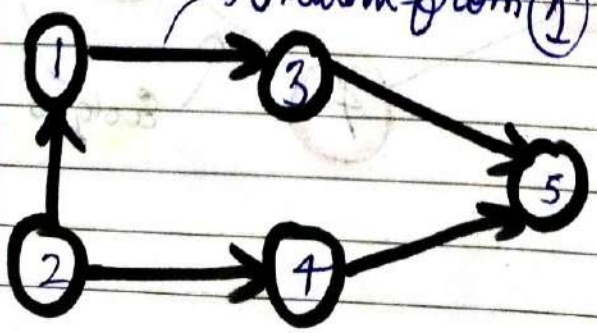
In this Graph,

the set of Vertices $V = \{0, 1, 2, 3, 4\}$,

the set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$

(6) INCIDENT TO AND INCIDENT FROM

In a directed Graph, an edge is said to be incident from a vertex if it emerges out of the vertex and it is said to be incident on a vertex if it points at that vertex

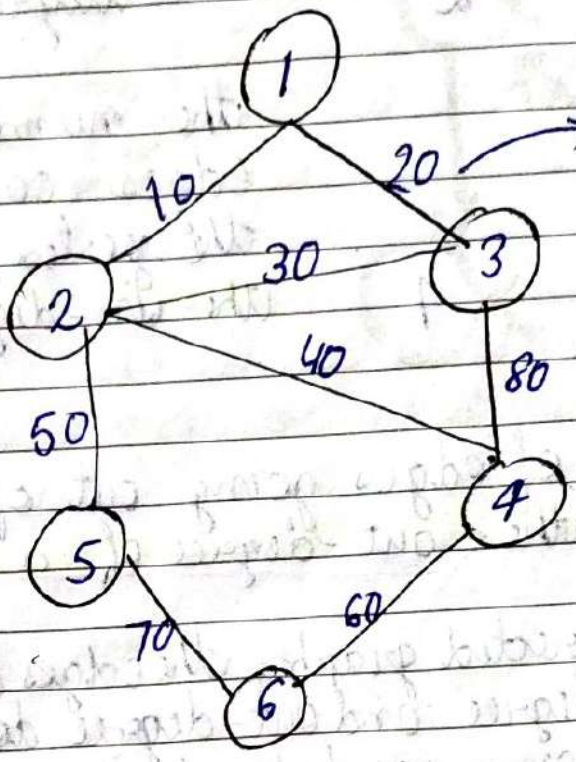


→ Incident from (1) and Incident on (3).

In a Non-directed graph, since the edges are Bidirectional, the "Incident-to" and "Incident From" phrases are not used.

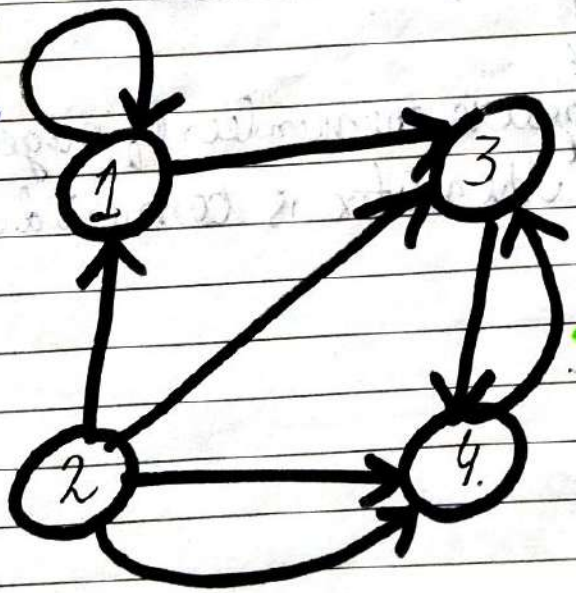
Date / /
Page No.

We just say that there is an edge b/w 2 vertices



→ Edge b/w 1 & 3

7 Self loop & Parallel Edges



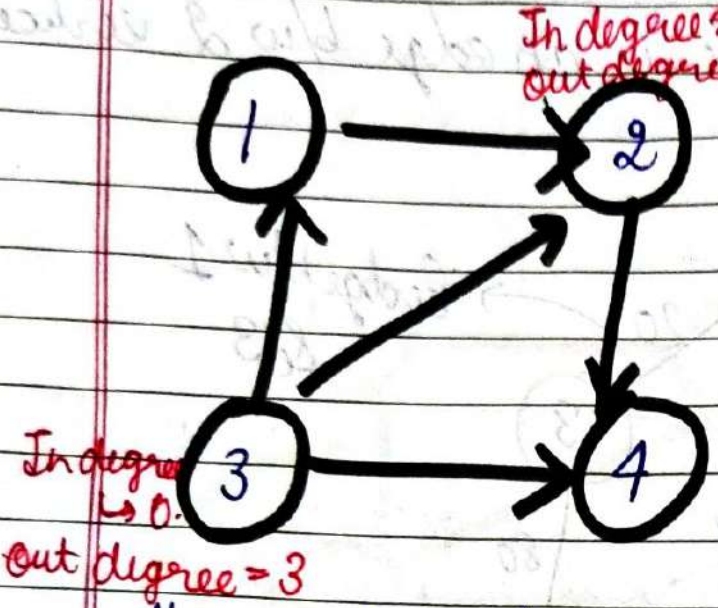
→ These 2 are parallel edges

As we can see, the direction of Parallel Edges can be same or opposite, Both are called Parallel Edges

These 2 are also Parallel edges.

8 In-degree and Out-degree

In a directed graph (also known as digraph),

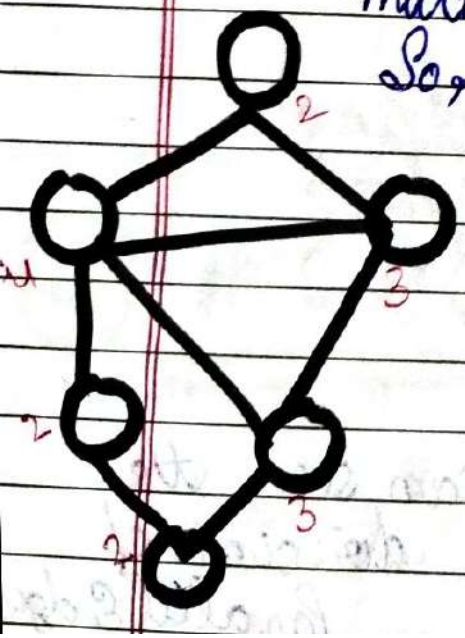


the number of edges coming into the vertex is called as the In-degree of a vertex

the number of edges going out of a vertex is called the out-degree of a vertex

For a Non-directed graph, the edges are Bi-directional ∴ The in-degree and out-degree do not make much sense as both of them are equal.

So, for a vertex V which belongs to a non-directed graph G, we have only degree



↓

Degree is the number of edges the vertex is connected to

(9) USAGE OF GRAPH DATA STRUCTURE

Date
Page No.

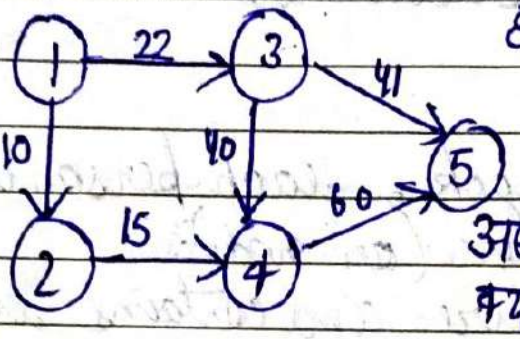
1. Graph are used to solve many real life problems
2. Graphs are used to represent networks.
3. The networks may include a) path in a city
   b) telephone network
   c) circuit network
4. Graphs are also used in Social Networks like a) LinkedIn
   b) Facebook.
5. For example, In Facebook, each person is represented with a vertex (or node).
6. Each node is a structure and contains information like person id,
   name,
   gender,
   locale, etc.
7) In facebook users are considered as vertices and if they are friends then, there is an edge running between them.

Facebook friend's suggestion algorithm uses graph theory. Facebook is an example of Undirected Graph.

8) In **Google map** uses graphs for building transportation system, where the intersection of 2 (or more) roads are considered to be a vertex and the road connecting 2 vertices is considered as an edge; thus their navigation system is based on the algorithm to calculate the shortest path between 2 vertices.

9) In **World Wide Web**, web pages are considered to be vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed Graph.

10)



हम assume करते हैं कि यहाँ 1 vertex/node 1 city को denote करती है।
अब हमें अगर ये check करना है कि क्या city 2 से city 5 तक जाने का कोई path है या नहीं, तो इस तरीके की problems के लिए हम Graph data structure का use कर सकते हैं।

Let's say, the weight of each edge denote the distance from 1 city (vertex) to another city (vertex)

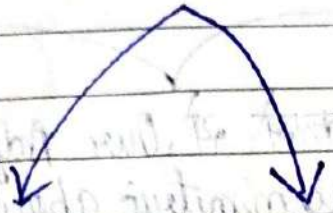
a) We want to find minimum distance/shortest path b/w 2 cities } We will use Graph Data Structure

10

REPRESENTATION OF GRAPH

Date
Page No.

Graphs को हम 2 ways से represent करते हैं



Adjacency Matrix

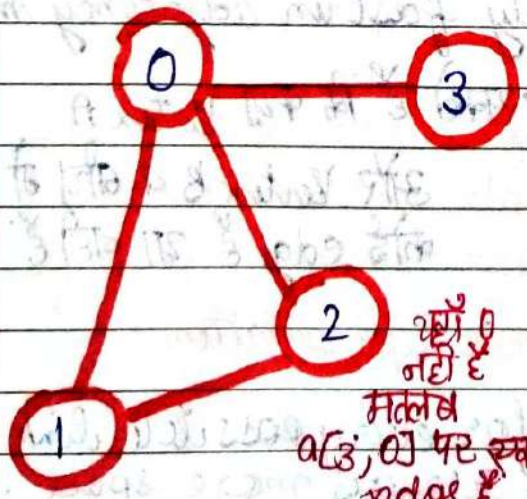
Adjacency List

1) ADJACENCY MATRIX

→ एक adjacency matrix basically एक 2-d array होता है where no. of rows and columns are equal to V (no. of vertex of graph)

UNDIRECTED GRAPH

If the value of any element a[i][j] is 1, it means that there is an edge that is connecting vertex i and vertex j.



यहाँ 1 नहीं है मतलब a[3,0] पर एक edge है मतलब 3 और 0 के बीच संबंध edge है

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

Graph adjacency matrix

Since, this was an undirected graph, for edge (0,2), we also need to mark edge (2,0);

Undirected graph में अगर (0,2) पर 1 mark किया तो (2,0) पर भी करना पड़ेगा।
क्योंकि Undirected Graph में Edge Bidirectional होती है।

This diagonal has each box filled with 0 (because there is no self loop)

इसा करने से, Our Adjacency matrix become symmetric about the diagonal

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

Our Adjacency Matrix is Symmetric about diagonal

Diagonal में सब 0 जगह same value है because our edges are Bidirectional

* Benefit of Adjacency Matrix Representation

→ Edge Lookup (checking if there is an edge that exist between Vertex A & Vertex B

is extremely fast in adjacency matrix (Adjacent Matrix में)

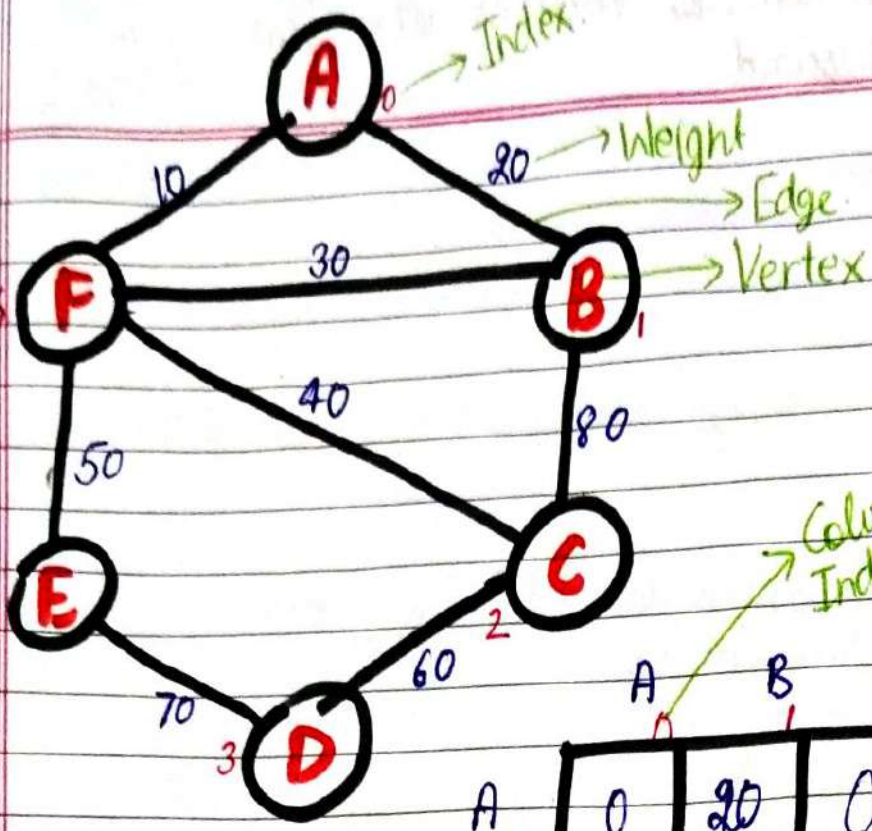
मतलब हम देखते ही, बता सकते है कि क्या Vertex A और Vertex B के बीच में कोई edge है या नहीं है तुरंत

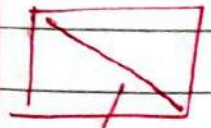
Cons of Using Adjacency Matrix Representation

We have to reserve space for every possible link b/w all vertices (V*V), so it requires more space.

(08) Weighted Non-directed / Undirected Graph

Date / /
Page No.





Row Index

Column Index

	A 0	B 1	C 2	D 3	E 4	F 5
A 0	0	20	0	0	0	10
B 1	20	0	80	0	0	30
C 2	0	80	0	60	0	40
D 3	0	0	60	0	70	0
E 4	0	0	0	70	0	50
F 5	10	30	40	0	50	0

In the adjacency matrix, all elements of this diagonal is 0 because there is no self loop in the Graph.

a[0,1] = 20

It means there is an edge between element a 0th Index (A) & 1st Index (B)

→ Row Index = 5, col Index = 1

Since, it is a weighted graph, we have inserted the weight of edges at a[i,j]
If it's would have been a non weighted graph, we would have inserted 1 at a[i,j].

We should use the adjacency matrix representation only if the number of vertices are less than 10 Thousand.

Date / /
Page No.

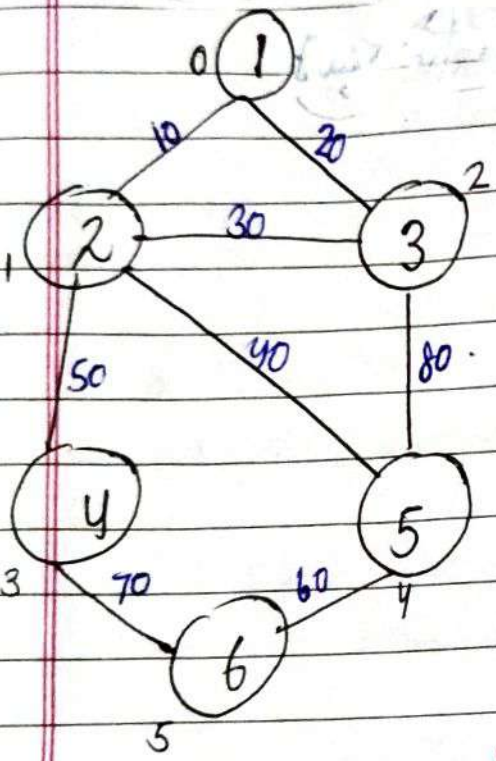
ADJACENCY LIST

Date
Page No.

↳ Graph की ये representation एक काफी famous representation है।

↳ Graph is represented as an Array of Arraylist <Edges>

Adjacency list is basically an array of ArrayList <Edges>



Arraylist के अंदर edges का data है।

start of edge, destination of edge, weight of edge

0 → 0-1 w10, 0-2 w20.
1 → 1-0 w10, 1-2 w30, 1-3 w50, 1-4 w40
2 → 2-0 w20, 2-1 w30, 2-5 w80.
3 → 3-1 w50, 3-1 w70
4 → 4-1 w40, 4-2 w80, 4-5 w60.
5 → 5-3 w70, 5-4 w60.

ये Edge class के object हैं जिसमें src, destination और weight होगा

हमने 6 size का array बनाया क्योंकि 6 vertices है। हर node के corresponding एक index होगा, Array के हर index पर corresponding Arraylists होगी, Arraylist contains Edges class Edge class में हर edge की 3 detail store करेगा

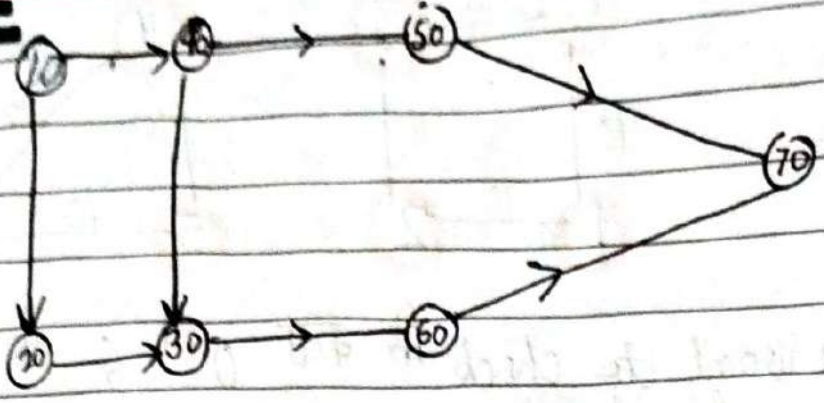
↳ start
↳ end
↳ weight

→ The source vertex of all element in arraylist at index i will be i only

Class-1

Date / /
Page No.

1. hasPath



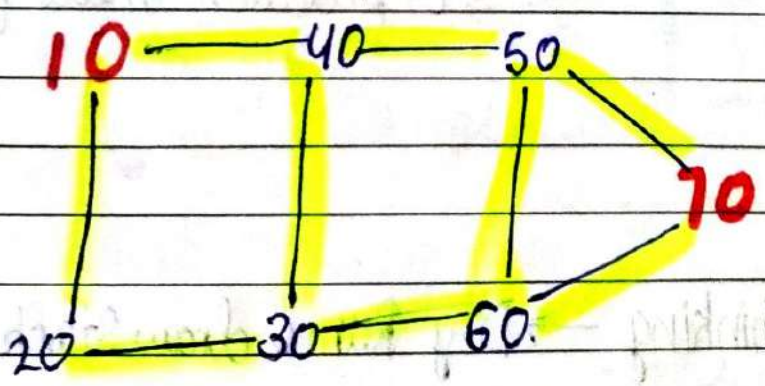
हमें एक graph given होगा
और 2 vertex given होगी → source
→ destination

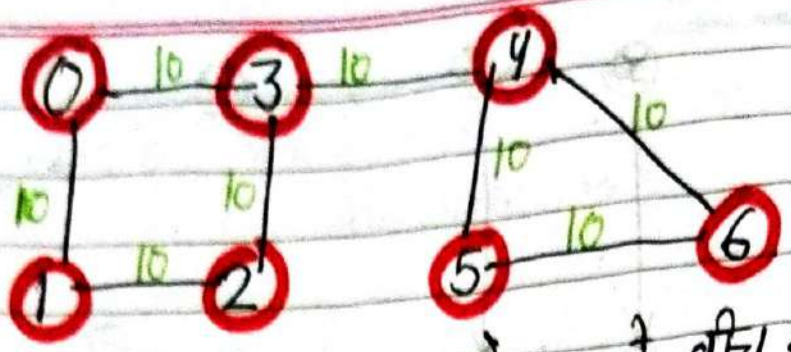
हमें ये बताना होगा कि क्या source vertex से destination vertex तक कोई path है?

अगर source और destination तक कोई path exist करता है, तो हम true print otherwise false.

eg → . Source = 10
destination = 70.

So, true print होगा क्योंकि 10 से 70 के बीच में path exist करता है।





Eg→2. We want to check कि क्या 0 से 6 के बीच में कोई path है?

Source = 0
destination = 6.

So, Yes we can see that there are Paths b/w 0 & 6.
So we will print true.

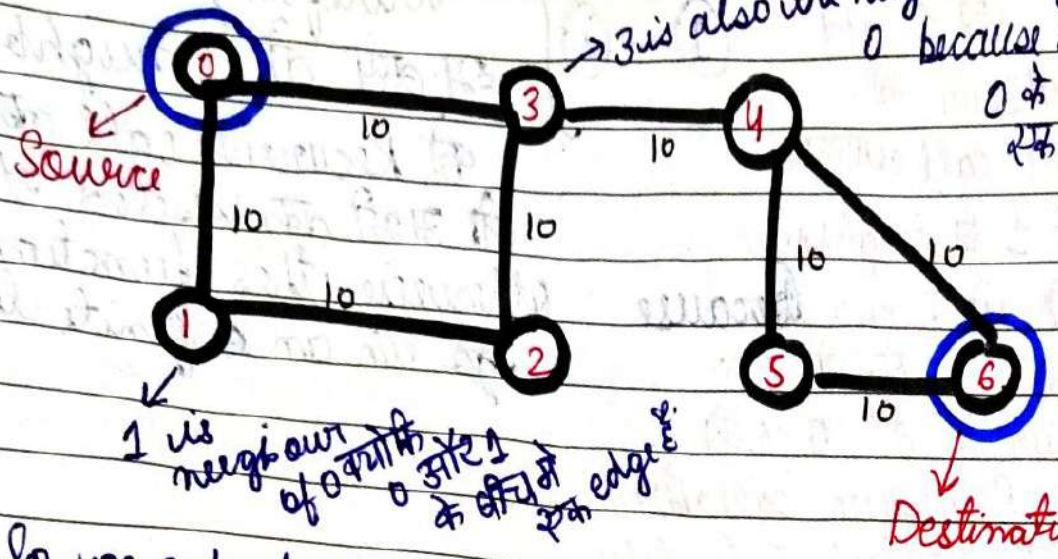
Approach → This problem is very much similar to the floodfill problem of Recursion.

We can use Recursion to solve this problem.

Recursion
- High Level Thinking → Expectation
  - → Faith
  - → Expectation meets faith.
- Low Level Thinking → Dry Run & draw Stack.

High Level Thinking

* **Expectation** → हम expect करते हैं कि hasPath() function हमें true return करेगा if there is a path in between source vertex & destination vertex



So we expect that hasPath() function will tell us whether the Path exist b/w src & dest.

* **Faith** →

अगर 0 से 10 तक का Path है तो 1 से 10 तक का Path भी जरूर होगा क्योंकि 1, 0 का ही neighbour है

neighbours of 0.

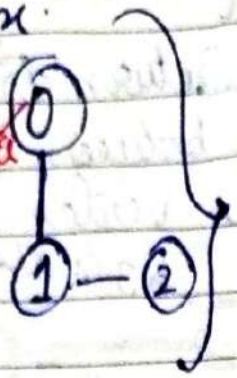
0 → (0-1-10), (0-3-10)
1 → (1-0-10), (1-2-10)
2 → (2-1-10), (2,-3-10)
3 → (3-1-10, (3-2-10), (3-4-10)
4 → (4-3-10), (4-5-10), (4-6-10)
5 → (5-4-10), (5-6-10)
6 → (6-4-10), (6,5-10)

4, 5 are neighbours of 10.

This is the Input Array given for graph.

So, hasPath() function पर हम faith रखते हैं कि अगर hasPath() function can tell that if there is a path b/w 0 to 6. तो hasPath() function ये जरूर बता देगा कि 0 के neighbours से कोई Path exist करता है या नहीं।

So, We will make a Recursive call on neighbours of source vertex



But to be very careful, हम बस वो ही neighbours को Recursive call लगाएँगे जो अभी तक visited नहीं है otherwise this function will go in an infinite loop

0 1 का neighbour है तो 1 को Recursive call लगी, अब 1 अपने सारे neighbour को Recursive call लगाएगा तो 0 भी 1 का neighbour है तो 0 को भी Recursive call, अब 1 फिर से 0 का neighbour है तो 0 1 को फिर से Recursive call लगेगी तो ऐसे ये loop Infinitely चलता ही जाएगा

So, this case can give us Stack Overflow error.

तो इस situation को handle करने के लिए हम हर एक vertex के लिए एक Boolean Array बनाएंगे और store करेंगे कि क्या इस vertex को पहले visit किया गया है। If the vertex is marked true in visited array, तो मतलब वो vertex पहले से ही visited है तो हम उस Vertex को फिर से Recursive call नहीं लगाएँगे

* Expectation meets faith →

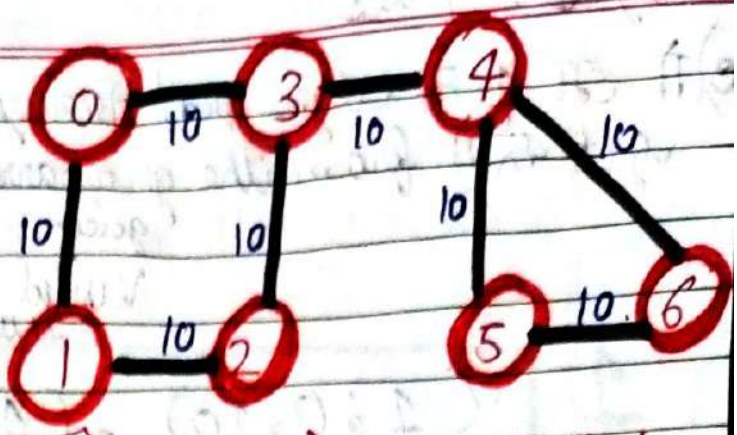
We will return true if the Recursive calls returns true else we return false.

अगर 0 के किसी भी neighbour (1/2) ने true return किया, तो हम true return करेंगे Otherwise we will return false.

Dry Run

(0)

We will get the graph as our input.
and a source → 0
a destination → 6.



(vertex 1) source neighbour (vertex 2) weight

0 → (0, 1, 10), (0, 3, 10). → 0 से 2 edge attached है। There are 2 neighbour of 0 → 1, 3 are neighbour of 0.
1 → (1, 0, 10), (1, 2, 10)
2 → (2, 1, 10), (2, 3, 10)
3 → (3, 0, 10), (3, 2, 10), (3, 4, 10)
4 → (4, 3, 10), (4, 5, 10), (4, 6, 10)
5 → (5, 4, 10), (5, 6, 10)
6 → (6, 4, 10), (6, 5, 10).

Approach → (1) We will make a function hasPath() जिसमें हम source, destination, graph, visited pass करेंगे। (→ boolean array)

(2) Get the arraylist of edges from array [src]. (arraylist of array) इससे हमें पता चल जाएगा कि source के neighbours कौन-कौन हैं।

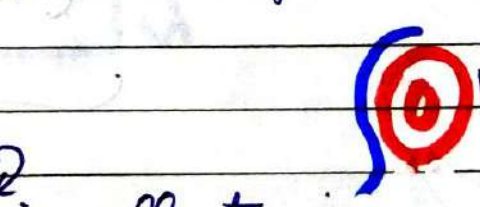
a) source = 0
destination = 6
b) Src ≠ dest

0 → [(0, 1, 10), (0, 3, 10).] ← 1 and 3 are neighbours of 0.

c) We got this arraylist at array [src]

source was 0

Initially first we check src ≠ destination & visited = true



d) So, Now we make Recursive calls to 1 and 3 and mark 0 as visited = true

e) Make Recursive calls to 1 & 3 because they are not visited

(3)

a) Now, we are having src = 1, dest = 6.

→ Mark 1 as visited now

b) src ≠ des

c) तो हम 1 के edges वाली Arraylist get करेंगे from the graph.

Visited = true

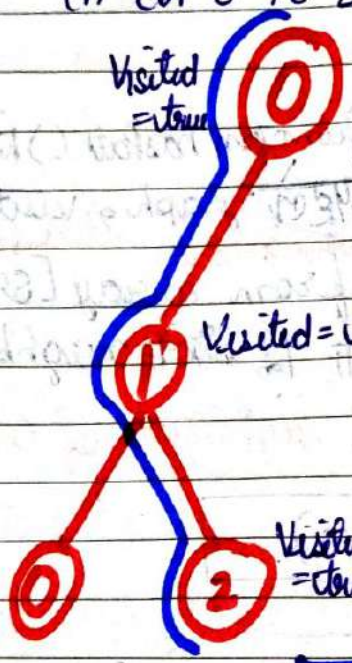
1 [(1, 0, 10) , (1, 2, 10)]

So this is the arraylist of edges.

d) तो हमे पता चला कि 0 & 2 are the neighbours of 1. लेकिन 0 is already a visited vertex तो हम 0 पर recursive call नहीं लगाएँगे।

e) तो फिर 2 पर recursive call लगेगी।

(4)



→ a) src = 2 → Mark 2 as visited
dest = 6.

b) check if src == dest
so 2 ≠ 6, so we will now make Recursive calls to neighbour of 2.

c) get the edges(arraylist of edges) at index 2 in graph array

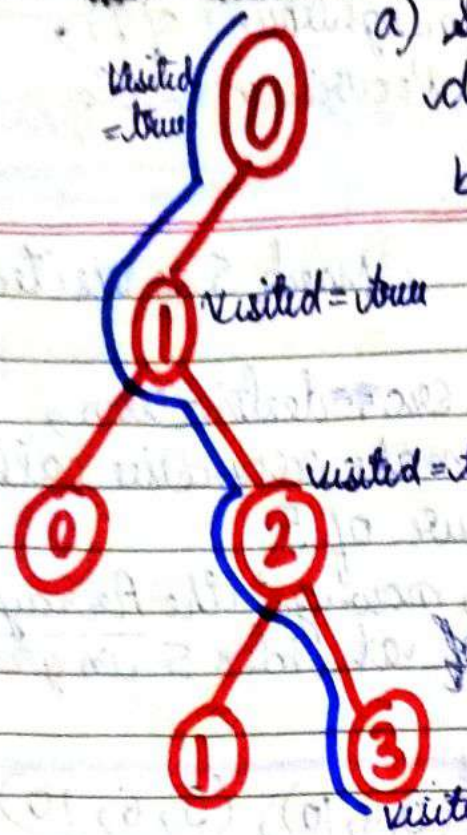
2 [(2, 1, 10), (2, 3, 10)]

So this is the arraylist of edges

d) तो हमे पता चला कि 1 and 3 are the neighbours of 2. लेकिन 1 is already a visited vertex, so 1 पर हम Recursive call नहीं लगाएँगे तो बस 3 पर Recursive call लगेगी

e) Recursive call to only 3.

5)



a) Src = 3 Mark 3 as visited
destination = 6.

b). Check if src == dest,
So, 3 ≠ 6, so we will now make Recursive calls to neighbour of 3.

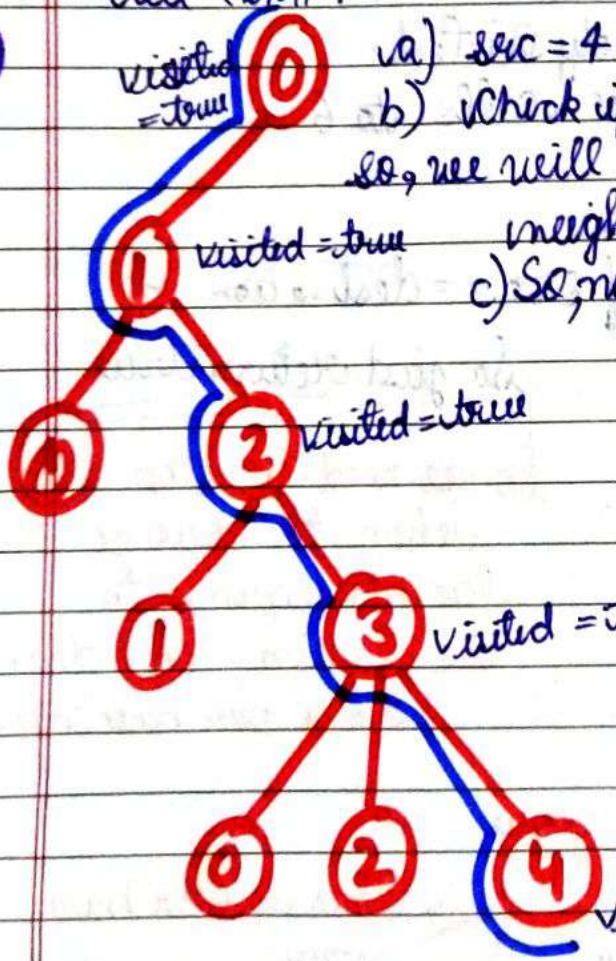
c) So, Now get the arraylist of edges at index 3 in graph.

3 [(3,0,10), (3,2,10), (3,4,10)]

So, this is the arraylist of edges

d) तो हमे पता चला कि 0, 2, 4 are the neighbours of 3. लेकिन 0, 2 are already visited vertex, so, 0, 2 पर recursive calls नही लगेगी तो बस 4 पर Recursive call लगेगी।

6)



a) src = 4, dest = 6 Mark 4 as visited.
b) Check if src == dest, so 4 ≠ 6
So, we will make recursive calls to neighbour of 4.
c) So, now get the Arraylist of edges at index 4 in graph.

4 [(4,3,10), (4,5,10), (4,6,10).]

So, this is the Arraylist of edges.

d). So, हमे पता चला कि 3 is already a visited node

e) So we can make recursive calls to 5, 6.

(7)

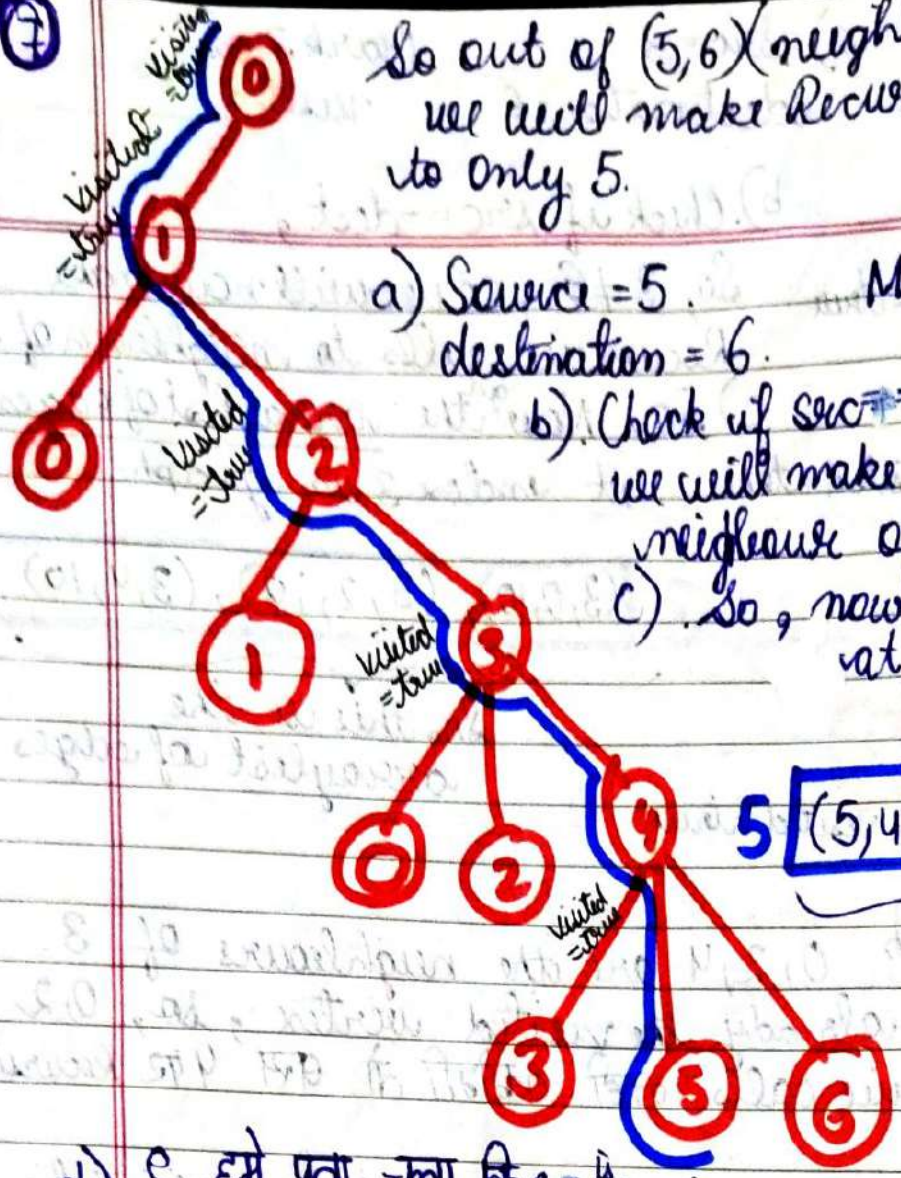
So out of (5,6) (neighbours of 4), we will make Recursive calls to only 5.

Date / / Page No.

a) Source = 5. Mark 5 as visited.
destination = 6.

b) Check if src == destination, we will make recursive calls to neighbour of 5.

c) So, now get the Arraylist of edges at index 5 in graph.

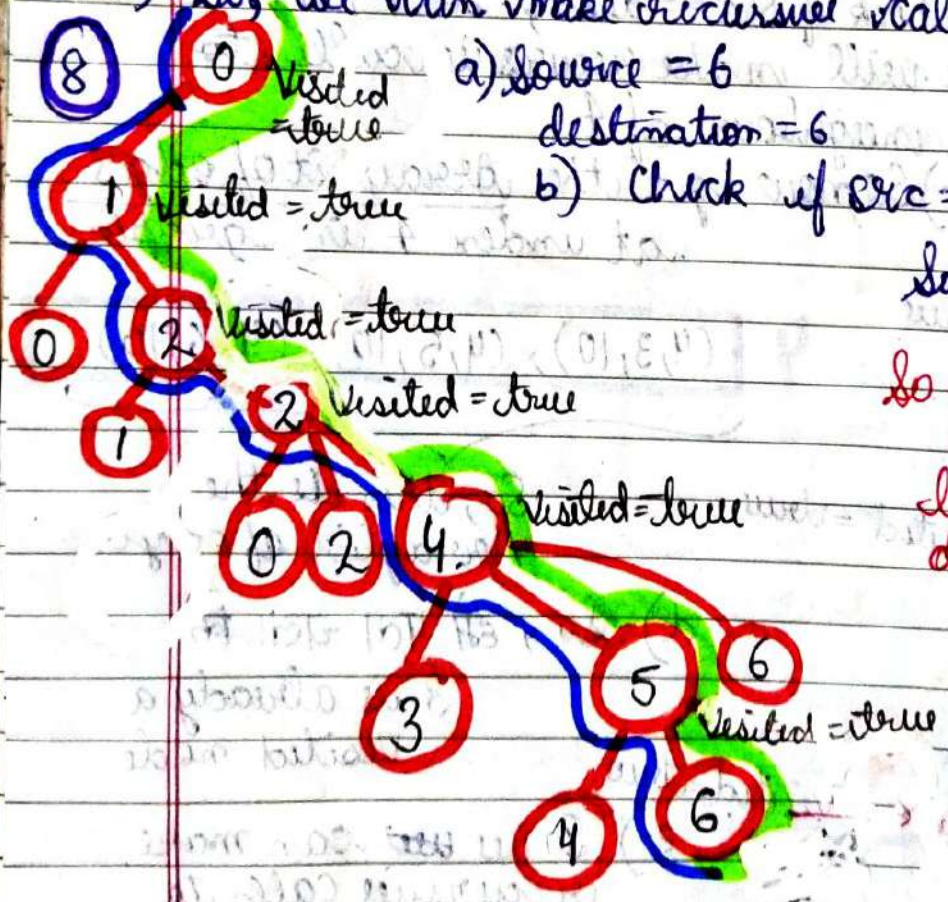


5 [(5,4,10), (5,6,10)]

So, this is the arraylist of edges

d) So, हमे पता चला कि 4 already visited.

e) So, we can make recursive calls to 6.



a) Source = 6
destination = 6

b) Check if src == destination ✓

So just return True

So we need to stop when the source become equal to destination, so this will be our base case.

→ So this is our base case.

hasPath <code>

Date
Page No.

```
import java.util.*;
public class Main
{
    static class Edge
    {
        int src;
        int nbr;
        int wt;

        Edge(int src, int nbr, int wt)
        {
            this.src = src;
            this.nbr = nbr;
            this.wt = wt;
        }
    }
}
```

→ यह Edge class है जिसमें हम edge की सारी details रखते हैं (source vertex (vertex 1), neighbour vertex (vertex 2) weight of edge).

↳ यह Edge class का constructor है।

```
public static Boolean hasPath(ArrayList<Edge> graph, int src, int destination, Boolean[] visited)
```

↳ हमें एक boolean value return करनी होगी which tells कि source और destination के बीच में कोई path है या नहीं।

We have passed the given graph, the source vertex, the destination vertex, the Boolean array visited as our parameter.

```
{
  if (src == destination)
  {
    return true;
  }
```

Base Case

```
  visited[src] = true;
```

Mark the visited[src] as true ताकि src vertex को फिर से visit न करें।

```
  for(int i=0; i<graph[src].size(); i++)
  {
```

Arraylist at src index के सभी Edges पर visit करेंगे और सिर्फ उन्हीं neighbour पर Recursive call लगेगी जो कि पहले से visited नहीं हैं।

हमने source पर जाने वाली सभी edge की Arraylist पर loop लगाया

```
    Edge e = graph[src].get(i);
    int nbr = e.nbr;
```

↳ हमने source के neighbour vertex को receive करलिया

```
    if (visited[nbr] == false)
    {
```

→ और neighbour vertex already visited नहीं है तो ही उस पर recursive call लगेगी

```
      boolean hasNbrPath = hasPath(graph, nbr, destination, visited);
```

neighbour vertex is passed as source now

```
if (hasNbrPath == true)
{
    return true;
}
}
}
}
return false;
}
```

Date / /
Page No.

अगर neighbour vertex से path है तो मतलब source से भी path है so we will return true.

अगर source और destination के equal नहीं है और source का कोई भी Recursive call से true return नहीं करता तो we will simply return false:

```
public static void main (String [] args)
{
    BufferReader br = new BufferReader (new InputStreamReader
                                        (System.in));
    int vertices = Integer.parseInt (br.readLine());
    ArrayList<Edge> [] graph = new ArrayList [vertices];
    for (int i=0; i< vertices ; i++)
    {  graph[i] = new ArrayList<>();
    }
    int edges = Integer.parseInt (br.readLine());
    for (int i = 0 ; i < edges ; i++)
    {  String [] parts = br.readline().split (" ");
       int v1 = Integer.parseInt (parts [0]);
       int v2 = Integer.parseInt (parts [1]);
       int wt = Integer.parseInt (parts [2]);
       graph[v1].add (new Edge (v1, v2, wt));
       graph[v2].add (new Edge (v2, v1, wt));
    }
    int src = Integer.parseInt (br.readLine());
    int dest = Integer.parseInt (br.readLine());
    Boolean visited [] = new Boolean [vertices];
```

no. of vertices input करेंगे

Array of Arraylist बनाया Because the input will be in form of adjacency list

Basically Adjacent list की structure होती है ; उसमें list होती है

Array के हर index पर new Arraylist को declare किया जो कि blank है।

no. of edges input करेंगे

String, edge के v1, v2 और wt को receive किया।

String को string array में split() use करके

edges को input किया और उनके corresponding Arraylist में add किया

v1, v2 index के corresponding Edge को accordingly Arraylist में add करेंगे

यहां vertex को visited है या नहीं, ये check करने के लिए Boolean array बनाया।

```
for(int i=0; i<visited.length; i++)
{
    visited[i]=false;
}
Boolean ans = hasPath(graph, src, dest, visited);
System.out.println(ans);
}
}
```

Date / /
Page No.

visited Array के सभी index पर by default false डाल दिया.

graph → Adjacency list
src → source vertex
dest → destination vertex
visited → Boolean Visited Array

Lec - 2.

Date
Page No.

Print ALL Path

↳ We need to find all possible path between 2 vertices of a graph
Print all the path in lexographical order

(जैसे dictionary में order होता है)

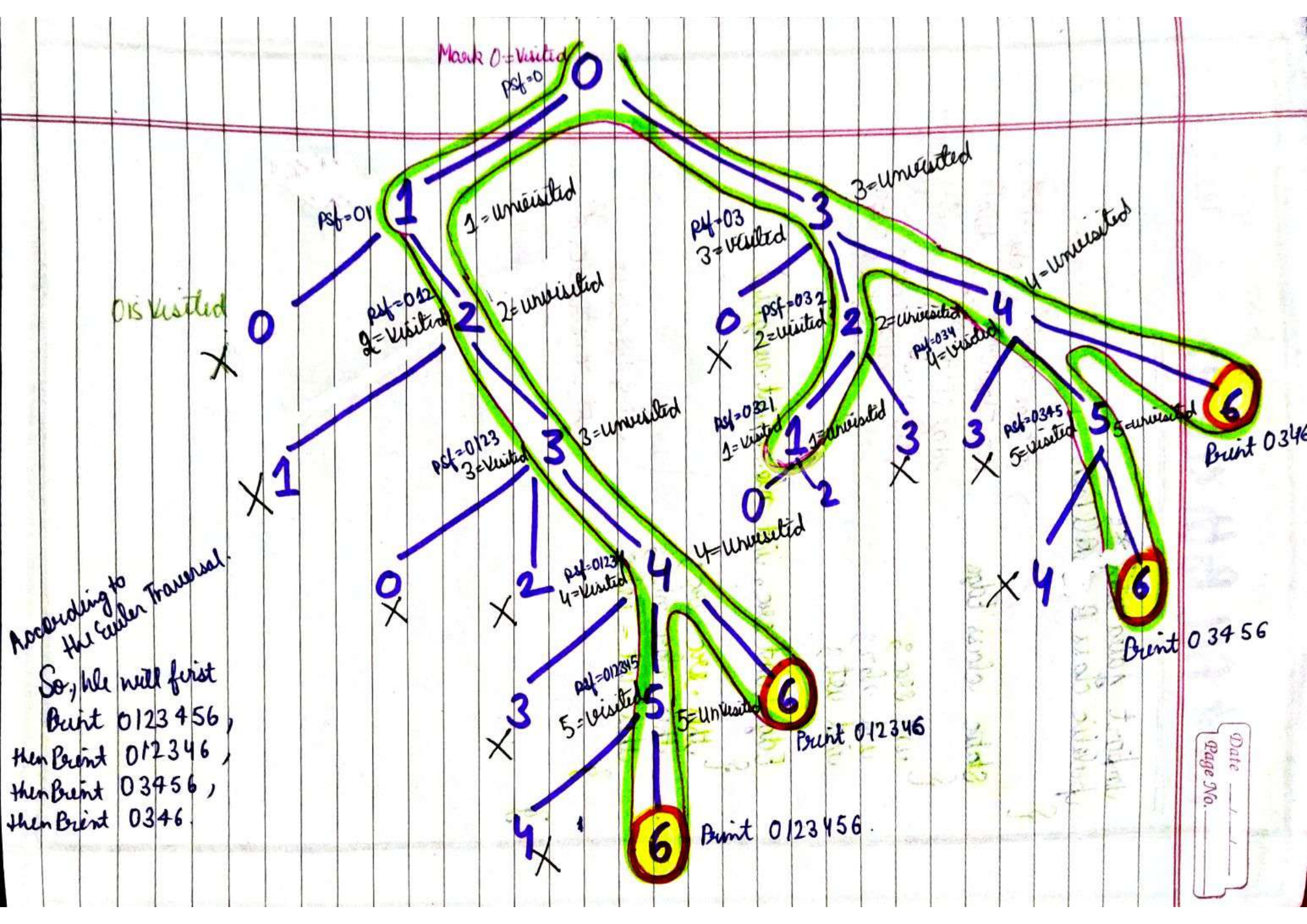
पहले सब 0 से start
01
02
034
फिर सब 1 से
15
16 & so on.

Approach

This problem is very similar to last problem.
But this time, we will add the visited vertex in pathSoFar & print pathSoFar if the source & destination vertex are same.
Also we will mark visited true before the recursive call & we will mark visited false after the recursive call
So, that the vertex can be visited in some other path.

Recursion → Expectation
We expect that Print All Path gives us the Path from source to destination. ... will check & print the path from.

Faith → We keep the faith that the function source's neighbour to destination
We will add source to psf before making recursive call to source's neighbour


Mask 0=Visited
Psf=0
0
1
Psf=01
1=Unvisited
3
3=Unvisited
Psf=03
3=Visited
0 is Visited
0
Psf=012
2=Visited
2
2=Unvisited
0
Psf=032
2=Visited
2
2=Unvisited
4
4=Unvisited
Psf=034
4=Visited
1
Psf=0321
1=Visited
1=Unvisited
3
3
Psf=0123
3=Visited
3
3=Unvisited
Psf=0345
5=Visited
5
5=Unvisited
6
Brint 0346
0
2
4
4=Unvisited
Psf=0123
4=Visited
4
6
Brint 03456
3
Psf=012345
5=Visited
5
5=Unvisited
6
Brint 012346
4
6
Brint 0123456.
According to the Euler Traversal.
So, We will first
Brint 0123456,
then Brint 012346,
then Brint 03456,
then Brint 0346.
Date
Page No.

Print All Path <code>

```
import java.util.*;
public class PrintAllPath
{
    static class Edge
    {
        int src;
        int nbr;
        int wt;

        Edge(int src, int nbr, int weight)
        {
            this.src = src;
            this.nbr = nbr;
            this.wt = wt;
        }
    }
}
```

→ यह Edge class है।
इस Edge class के object में,
edge से related 3 detail
होती है → src(vertex 1)
nbr(vertex 2).
weight of Edge.

→ यह Edge class
का constructor है।

```
public static void PrintAllPath(ArrayList<Edge>[] graph,
                                int source, int destination,
                                boolean[] visited, String psf)
{
```

function कुछ return नहीं करता

(Array of Arraylist) Graph, source vertex, destination vertex, visited array, path so far are passed as parameter

```
    if(source == destination)
    {
        System.out.println(psf);
        return;
    }
```

अगर source और destination equal मिल जाता है तो हम path so far को print कर देंगे और return कर जाएंगे लेकिन अगर source और destination equal नहीं हैं तो

→ हम source को सबसे पहले visited true mark करेंगे

→ source के unvisited neighbours पर Recursive call लगाएंगे (psf में source को add करके)

→ Recursive calls हो जाने के बाद (return करने के) post order में वापस से source को unvisited बना देंगे ताकि किसी और path में हम source से भी visit कर सके।

```
    visited[source] = true;
```

हमने source index की arraylist पर loop चलाया

```
for (int i=0; i < graph[source].size(); i++)
{
    Edge e = graph[source].get(i);
    int neighbour = e.nbr;
```

Arraylist से ith index की edge receive किया

edge से vertex2 (source का neighbour होगा).

```
    if (visited[neighbour] == false)
    {
        PrintAllPath(graph, neighbour, destination,
                     visited, psf + source);
    }
    visited[source] = false;
}
```

हमने neighbour पर Recursive call लगाई

psf में source add किया so now psf = psf + source

Recursive call के बाद (Post Order में) source को visited → false mark करदो ताकि किसी और Path में ये vertex source को visit कर पाए

```
public static void main(String[]args)
{  BufferReader br= new BufferReader
                      (new InputStreamReader
                              (System.in));
int vertices= Integer.parseInt (br.readLine());
ArrayList<Edge>[]graph = new ArrayList [vertices];
for( int i=0;i< vertices ; i++)
  {
     graph[i]= new Arraylist<>();

  }
  int edges=Integer.parseInt(br.readline());
  for( int i=0; i< edges; i++)
  {
    String []parts= br. readline ().split();
   int v1=Integer.parseInt(parts[0]);
  int v2= Integer.parseInt(parts
  graph[v1].add (new Edge (v1, v2, wt));
  }

int src = Integer.parseInt(br.readline());
int dest= Integer.parseInt (br.readLine());

Boolean visited [] = new Boolean[vertices];
for(int i=0 ; i< visited.length();i++)
  {
    visited[i]=false;
  }
  String psf = "";
Print AllPath (graph, src, dest, visited, psf);
}
}
```

initially path so far blank रहेगा

Ques-3.

MULTISOLVER

→ ऐसा ही question हम ने Generic Tree में solve किया है

Date
Page No.

हमे एक graph दिया होगा
तब के सारे path निकालने

अब हमे source से destination आने हैं तो अब हमे

अब हमे बताना होगा about -

→ Smallest Path
→ Largest Path
→ Just Larger Path than x
→ Just Smaller Path than x.
→ yth largest path

x को input किया जाएगा
call x as criteria

y हमें input दिया जाएगा.

Graphs के question बिल्कुल Generic Tree के जैसे होते हैं
(बस visited का concept generic tree में लगाओ और सभी nodes के child को जगह neighbours को visit करना होगा)

Input given ↓

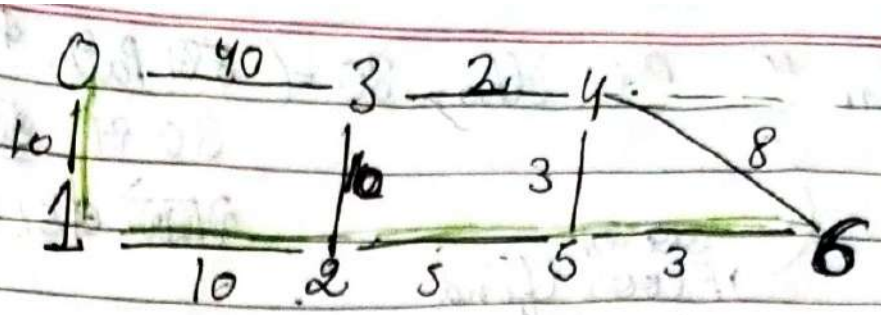
```
7        → no. of vertex
9        → no. of edges
0 1 10
2 3 10
0 3 40
3 4 2
4 5 3
5 6 3
4 6 8
2 5 5
0        → source
6        → destination
30       → x (criteria)
4        → y
```

So these are the edges with

we need to print smallest and largest path from source to destination

→ The Path should have total weight
→ a) just larger than x
→ b) just smaller than x

we need to print yth largest path



* We need to print

a) → Smallest Path (in terms of weight) from source (0) to destination (6)

= 10 + 10 + 5 + 3 = 28

0 — 1 — 2 — 5 — 6

01256@28

This is Smallest Path in terms of weight.

b) → Largest Path (in terms of weight) from source (0) to destination (6).

= 40 + 10 + 5 + 3 + 8 = 66

0 — 3 — 2 — 5 — 4 — 6

032546@66.

c) Just Larger Path than 30 (उस Path का weight 30 से बड़ा में सबसे छोटा होना चाहिए

↳ Let find करनी है Paths के weights में (30 की)

= 10 + 10 + 5 + 3 + 8 = 36

0 — 1 — 2 — 5 — 4 — 6 @ 36

012546@36.

d) Just Smaller Path than 30 = (उस Path का weight 30 से छोटे weights में सबसे बड़ा होना चाहिए)

↳ 30 का Floor find करना है (Paths के weight में).

40 − 2 − 3 − 3 = 48.

0 — 3 — 4 — 5 — 6 @ 48.

03456@48.

e) 4th largest Path

↳ इसे find करने के लिए हम Priority Queue का use करेंगे

हम 4th largest find करना है

↳

हम Numm Priority Queue का logic use करेंगे

→ हम K size की PQ में first 4 path डालेंगे फिर एक Path remove होगा PQ से और next path add होगा, ये follow करने पर last में 4th largest element हमें peek पर मिलेगा

Priority is based on weight of Path

अगर next path का weight PQ के top path के weight से small होगा तो हम next path PQ में add होगा

* 1. Smallest Path → When source and destination vertex becomes equal, if the weight so far (wsf) is less than the smallestpath weight (spathwt) then value of spathwt get updated to the value of wsf.

Date / / Page No.

1' We will make 8 static variables
Static String spath
Static int spathwt
Static String lpath
Staic int lpathwt
static String cpath
Static int ceilpathweight
static String fpath
static int fpathwt

∴ Our spath is the psf.

```
if (wsf < spathwt)
{
    spathwt = wsf;
    spath = psf;
}
```

2. Largest Path → When source and destination vertex become equal, if the weight so far (wsf) is greater than largestPathWeight (lpathwt), then value of lpathwt get updated to value of wsf & our lpath will be psf.

```
if (wsf > lpathwt)
{
    lpathwt = wsf;
    lpath = psf;
}
```

3. Ceil Path ⇒ When source & destination vertex become equal, if the weight so far (wsf) is greater than given criteria, also wsf needs to be smaller than ceil path weight. If these conditions are fulfilled, then ceilpathwt get updated to value of wsf and cpath becomes path so far (psf).

```
if (wsf > criteria && wsf < cpathwt)
{
    cpathwt = wsf;
    cpath = psf;
}
```

4. **Floor Path** To find the floor path, we need to find the largest weight out of all weights smaller than the criteria.

To do so, we first need to check if the wsf (weight so far) is smaller than given criteria, also wsf needs to be larger than current floor path weight (ffpathwt). If these conditions are fulfilled, then ffpathwt is updated to this wsf and ffpath becomes the path so far (psf).

```
if (wsf < criteria && wsf > ffpathwt)
{
    ffpathwt = wsf;
    ffpath = psf;
}
```

5. **Kth Largest Path** → kth largest Path find करने के लिए हम Priority Queue को use करेंगे Priority Queue में हम numbers को किसी भी order में add कर सकते हैं लेकिन, जब Priority Queue में से numbers remove होंगे तो highest priority वाला number सबसे पहले remove होगा। By default, PQ में smaller number को ज्यादा priority दी जाती है;

अगर हमें 3rd largest element (Path) get करना है, मतलब K=3, then PQ must always have 3 element in it. So, the size of PQ can be max 3. We will store pair of psf, wsf. Priority of PQ will be according to weight of pair. We remove pair from PQ, while traversing through all the paths (we remove 1 pair if the wsf is greater than pq.peek().weight) & add a new pair of wsf, psf.

```
if(pq.size()<k)
{
pq.add(new Pair(vesf,psf);
}
else
{
  if(vesf>pq.peek().vesf)
  {
  pq.remove();
  pq.add(new Pair(vesf,psf));
  }
}
```

MULTISOLVER

<code>

```
import java.util.*;
public class Main
{
    static class Edge
    {
        int src;
        int nbr;
        int wt;

        Edge(int src, int nbr, int wt)
        {
            this.src = src;
            this.nbr = nbr;
            this.wt = wt;
        }
    }

    static class Pair implements Comparable<Pair>
    {
        int wsf;
        String psf;

        Pair(int wsf, String psf)
        {
            this.wsf = wsf;
            this.psf = psf;
        }

        public int compareTo(Pair other)
        {
            return this.wsf - other.wsf;
        }
    }
}
```

This is constructor of Edge class

Comparable is implemented to compareTo function of PriorityQueue

wsf = weight so far

psf = path so far

This is constructor of Pair class

smaller element will get priority of PQ (smaller weight)

Date
Page No.

```
public static void main(String [] args)
{
    BufferReader br = new BufferReader
                        (new InputStreamReader(System.in));
    int vertices = Integer.parseInt(br.readLine());
    ArrayList <Edge>[] graph = new ArrayList[vertices];
    for(int i=0; i< vertices; i++)
    {
        graph[i]= new ArrayList<>();
    }
    int edges= Integer.parseInt(br.readLine());
    for(int i=0; i<edges; i++)
    {
        String[] parts= br.readLine().split(" ");
        int v1 = Integer.parseInt(parts[1]);
        int v2 = Integer.parseInt(parts[2]);
        int wt = Integer.parseInt(parts[3]);

        graph[v1].add(new Edg(V1, V2, wt));
        graph[v2].add(new Edge(V2, V1, wt));
    }
    int src= Integer.parseInt(br.readLine());
    int dest = Integer.parseInt(br.readLine());
    boolean visited[] = new boolean[vertices];
    int criteria = Integer.parseInt(br.readLine());
    int k = Integer.parseInt(br.readLine());
    multisolver(graph, src, dest, visited, criteria,
                k, src+" ", 0);
```

Annotations:

- no of vertices input की
- → इसने array of arraylist बनाया
- array के हर index पर एक arraylist declare की
- → No. of edges input की
- Adjacency list को fill किया
- → source की value input की
- ↳ destination vertex की value input की
- visited check करने के लिए boolean array बनाया
- criteria की value input की
- ↳ Kth largest path बताने के लिए K की value input की
- multisolver function को call किया
- psf (src+" "), wsf (0)

Date / / Page No.

```
System.out.println("Smallest Path = " + spath
    + "@" + spathwt);
```

spathwt is a static variable for smallest path weight

spath for smallest path; spath is a static variable

```
System.out.println("Largest Path = " +
    lpath + "@" + lpathwt);
```

lpath is a static variable for largest path

lpathwt is a static variable for largest path weight.

```
System.out.println("Just Larger than " +
    criteria + " = " + cpath
    + "@" + cpathwt);
```

ceil path weight

ceil Path (path) is a static variable

```
System.out.println("Just Smaller Path than "
    + criteria + " = " +
    fpath + "@" + fpathwt);
```

floor path is static variable

floor path weight is static variable

```
System.out.println(k + "th largest path: " +
    pq.peek().left + "@" +
    pq.peek().wsf);
```

value of k was taken as input

kth largest path print ho rha hai

}

Date / /
Page No.

21 static variables

```
static String spath;
static Integer spathwt = Integer.MAX_VALUE;
static String lpath;
static Integer lpathwt = Integer.MIN_VALUE;
static String cpath;
static Integer cpathwt = Integer.MAX_VALUE;
static String fpath;
static Integer fpathwt = Integer.MIN_VALUE;
static PriorityQueue<Pair> pq = new
                    PriorityQueue<>();
```

```
public static void multisolver(ArrayList<Edge>[] graph, int src, int dest,
        boolean[] visited, int criteria, int k,
        String psf, int wsf)
{
    if (src == dest)
    {
        if (wsf < spathwt)
        {
            spath = psf;
            spathwt = wsf;
        }
        if (wsf > lpathwt)
        {
            lpath = psf;
            lpathwt = wsf;
        }
        if (wsf > criteria && wsf < cpathwt)
        {
            cpathwt = wsf;
            cpath = psf;
        }
        if (wsf < criteria && wsf > fpathwt)
        {
            fpathwt = wsf;
            fpath = psf;
        }
```

Date ___ Page No. ___

Notes:

- if (src == dest) → अगर source destination के equal हो जाता है
- if (wsf < spathwt) → और weight so far smallest path weight से smaller है तो smallest path हटाकर path so far और smallest path weight हटाकर weight so far
- if (wsf > lpathwt) → अगर weight so far बड़ा है largest path weight से तो lpath और largest path weight की value update करदो
- if (wsf > criteria && wsf < cpathwt) → अगर weight so far given criteria से बड़ा है और wsf ceil path weight से छोटा है तो ceil path weight और ceil path की value update करदो
- if (wsf < criteria && wsf > fpathwt) → अगर wsf criteria से smaller है और floor path weight से बड़ा है तो fpath और fpath weight की value update करदो

```
if(pq.size()<k){
        pq.add(new Pair(wsf, psf));
    }
    else { if (wsf > pq.peek().wsf)
        {
            pq.remove();
            pq.add(new Pair(wsf,psf));
        }
    }
}

visited[src] = true;

for(int i=0; i<graph[src].size(); i++)
{   Edge e = graph[src].get(i);
    int nbr = e.nbr;

    if(visited[nbr]==false)
    {
        multiSolver(graph, nbr, dest, visited,
                    criteria, k, psf+e.nbr,
                    wsf+e.wt
        );
    }
}
visited[src] = false;
}
}
```

अगर अभी pq का size k से smaller है तो psf, wsf का नया pair बनाके pq में add करदो

और अगर pq का size k है और pq के peek Pair का weight wsf से कम है तो pq से वह Pair remove करदो और pq में add करदो नया pair बनाके psf और wsf का

और अगर src और dest vertex equal नहीं है तो हम ये steps follow करेंगे

visited mark करदो src को

source के unvisited neighbour पर recursive call लगेगी

wsf में edge का weight add करदो

psf में unvisited neighbour को add करदो

src को visited = false mark करदो after the recursive calls