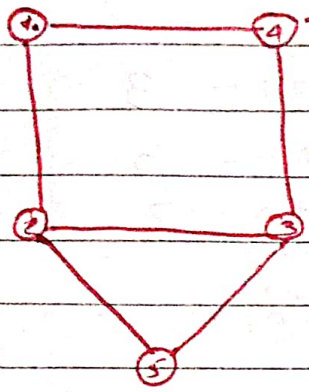
Discrete mathematics

Page No.: YOUVA

Graph

- Undirected graph



→ Node / Vertex

← edge (undirected edge)

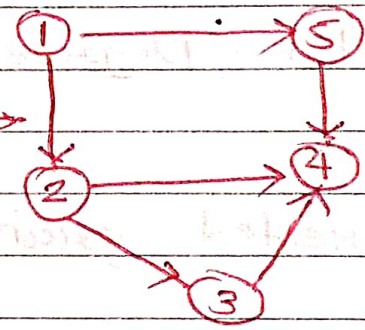
(Undirected Cyclic graph)

- directed graph

(directed Acyclic graph) (DAG)

Node →

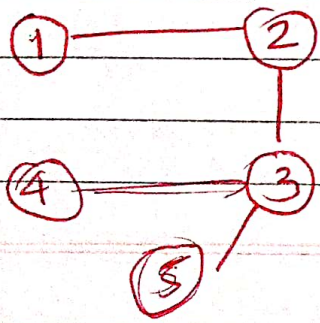
edge (directed) →



← edge (directed)

- Cycle in graph :- Start from a node end at that node.

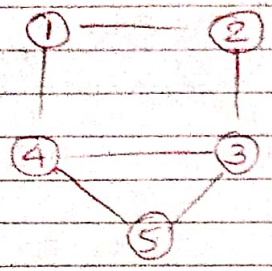
- Path :- No one Node appears twice.



1 2 3 5 ✓

1 2 3 2 1 ✗

Degree



D(3) = 3
D(4) = 3
D(5) = 2

Total Degree of graph = 2XE

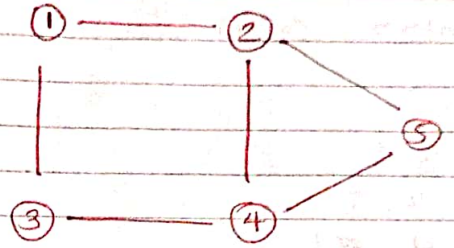
E = 6
Total Degree = 12

Directed graph → Indegree
Outdegree

Indegree → edges Coming In The node

outdegree → edges going out from node

Graph representation



Input

n = nodes
m = edges

n = 5
m = 6

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

Adjacency matrix

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

S(c) = O(N*m)

Adjacency list

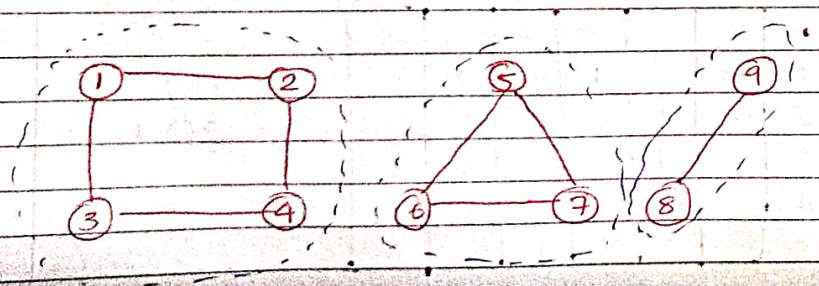
~~Vector~~ < Vector < int > > (n+1)

0 →
1 → {2, 3}
2 → {1, 4, 5}
3 → {1, 4, }
4 → {3, 2, 5}
5 → {2, 4}

S(c)
O(2E)

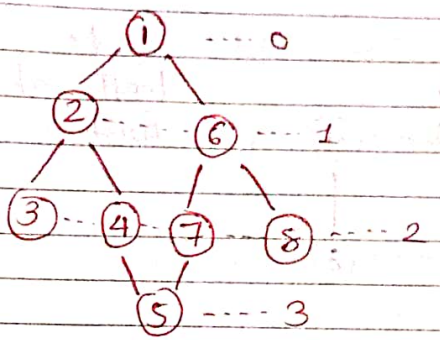
Vector < Vector < Pair < int, int > > >
in case of weighted

< to, weight >



4 Component of
Singal graph

BFS on graph (Traversal tech)

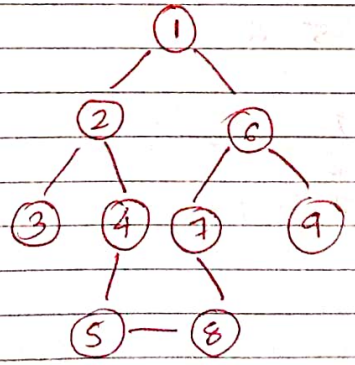


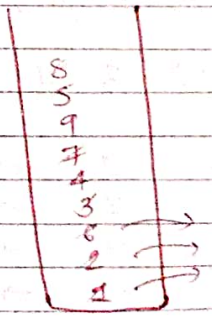
Levelvise Traversal
Starting Node = 1

1 2 6 3 4 7 8 5
0 1 2 3

=> starting Node = 6

6 1 7 8 2 5 3 4
0 1 2 3



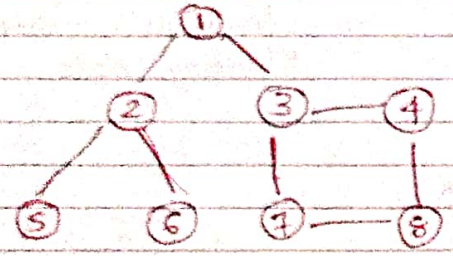


1 2 6 3 4 7 9 5 8

T(c) → O(N) + O(2E)
↑ ↑
!q.empty() for inside while

S(c) → O(N)

DFS on graph (Recursion)



go to depth at first

- starting Node = 1

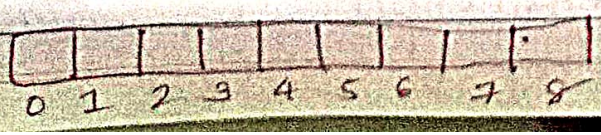
⇒ 1 2 5 6 3 7 8 4

- starting Node = 3

⇒ 3 4 8 7 1 2 5 6

1 → { 2, 3 }
2 → { 1, 5, 6 }
3 → { 1, 4, 7 }
4 → { 3, 8 }
5 → { 2 }
6 → { 2 }
7 → { 2, 8 }
8 → { 4, 7 }

vis



BFS

```
vector<int> bfs(int V, vector<int> adj[])
{
    int vis[V] = {0};
    vis[0] = 1;
    queue<int> q;
    q.push(0);
    vector<int> bfs;
    while(!q.empty()) {
        int node = q.front();
        q.pop();
        bfs.push_back(node);
        for(auto it : adj[node]) {
            if(!vis[node]) {
                vis[it] = 1;
                q.push(it)
            }
        }
    }
    return bfs;
}
```

T.C. → O(n) + O(2E)
(while, for(it))

S.C. → O(n)

DFS

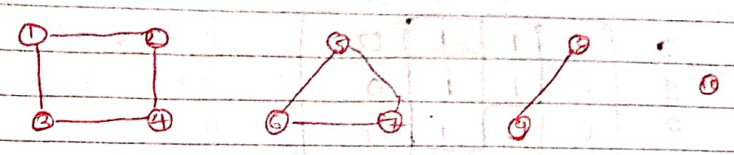
```
Void dfs (int node, vector<int> adj[],
          int vis[], vector<int> &ls)
{
    vis[node] = 1;
    ls.push_back(node);
    for (auto it : adj[node]){
        if (!vis[it]) {
            dfs(it, adj, vis, ls);
        }
    }
}

vector<int> dfsofgraph (int V, vector<int>
                        adj[])
{
    int vis[V] = {0};
    int start = 0;
    vector<int> ls;
    dfs(start, adj, vis, ls);
    return ls;
}
```

T(n) → O(n) + O(2E)
- O(n) ← for (dfs)
- O(2E) ← for (it)

S(n) → O(n) + O(n) + O(n)

No. of Provinces



⇒ 4 Province

```
start from 1 do dfs
    if node is unvisited after
    dfs cnt++;

for (i = 0 → V) {
    if (!vis[i]) {
        dfs(i);   cnt++;
    }
}
```

No. of Connected Components

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

ans = 3

BFS
Starting Node = {0, 1}
Starting Node = {4, 0}
Starting Node = {4, 3}

{4,2}
{1,1}
{1,2}
{0,2}
{0,1}

```
for (row → 0 — n) {
    for (col → 0 — m) {
        if (!vis[row][col]) {
            bfs(row, col);
            cnt++;
        }
    }
}
```

T(C) → $O(N^2) \times 9$
$\approx O(N^2)$

S(C) → $O(N^2)$ + $O(N^2)$

Flood fill Algorithm

1 (0)	1 (0)	1 (0)
1 (0)	1 (0)	0
1 (0)	0	1

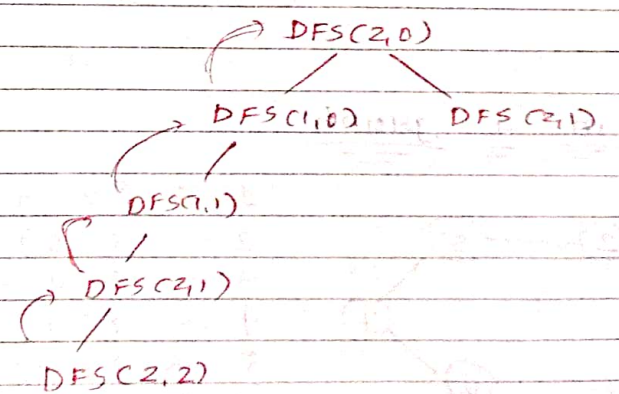
sr = 1 sc = 1
newColor = 2

initial = 1
Colour all neighbours of four direction from starting row sr, starting column sc } initial

both bfs, dfs will consist

	0	1	2
0	1	1	1
1	2	2	0
2	2	2	2

sr = 2 sc = 0
newColor = 3
initial = 2



Rotten oranges

0	1	2
0	1	1
2	1	1

2 → rotten orange
1 → Fresh orange
0 → empty

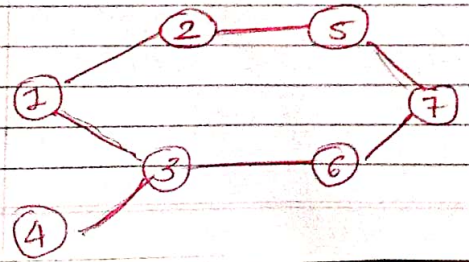
⇒ In one second rotten orange will infect all four neighbours

0	1	2
0	1	1
2	1	1

In 2 Second

2,2 → 2
1,1 → 2
0,1 → 1
1,2 → 1
2,1 → 1
0,2 → 0
2,0 → 0

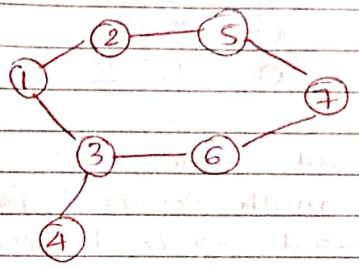
Cycle in a graph



1 → {2,3}
2 → {1,5}
3 → {1,4,6}
4 → {3}
5 → {2,7}
6 → {3,7}
7 → {5,6}

Vis →

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



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

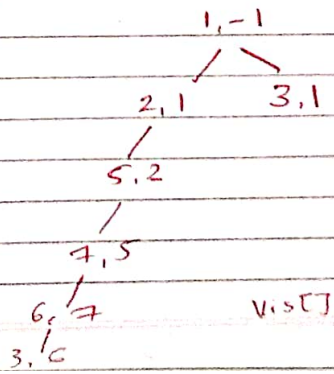
Q → cycle

T(C) → O(N+2E)

S(C) → O(N) + O(N)
≈ O(N)

⇒ Same thing can be done using DFS

⇒ If visited node come again then return true



1 → {2,3}
2 → {1,5}
3 → {1,4,6}
4 → {3}
5 → {2,7}
6 → {3,7}
7 → {5,6}

vis[] =

0	1	2	3	4	5	6	7	8

Distance of nearest cell having 1 in a binary matrix

1	0	1		0	1	0
1	1	0	→	0	0	1
1	0	0		0	1	2

→ nearest 1 from all nodes → multi source bfs

→ BFS should be used → level wise

	0	1	2
0	0	0	0
1	0	1	0
2	1	0	1

	0	1	2
0	2	1	2
1	1	0	1
2	0	1	0

(0,0), 2
(0,2), 2
(1,0), 1
(2,1), 1
(1,2), 1
(0,1), 1
(2,2), 0
(2,0), 0
(1,1), 0

Surrounded Regions

X	O	X	X	→ Convert this to X
X	O	X	X	→ set of X connected to boundry O can't be converted
X	O	O	X	
X	O	X	X	
X	X	O	O	

	0	1	2	3	4
0	X	X	X	X	X
1	X	O	O	X	O
2	X	X	O	X	O
3	X	O	X	O	X
4	O	O	X	X	X

→ Target all O's at boundry dfs from them and mark all Neighbours unconvertable

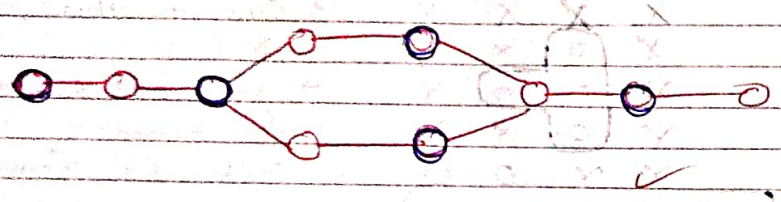
dfs(4,0) → dfs(4,1) → dfs(3,1)

mark Visited

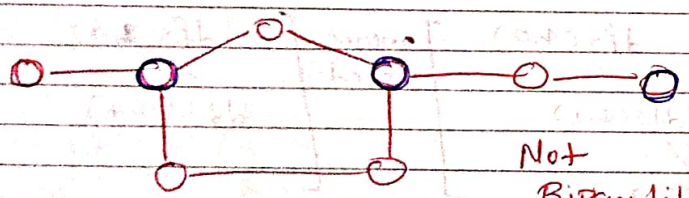
dfs(2,4) → dfs(2,4)

Bipartite graph

0 → 4
0 → 6



→ Pick two Colors
Color all node
No adjecent node should have Same Color



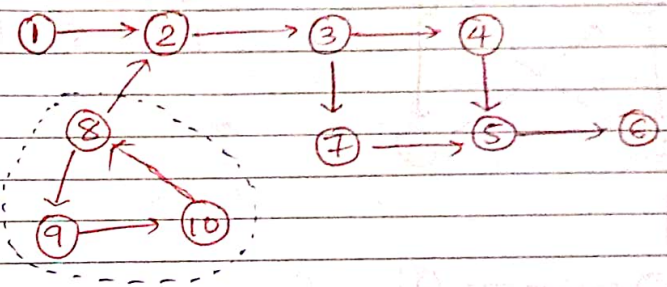
Not Bipartite graph

→ Linear graph } always Bipartite
No cycle

graph with even cycle length } Bipartite

graph with odd cycle length } Not Bipartite

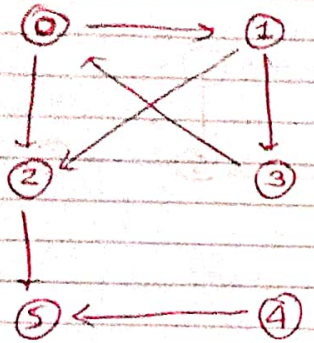
Detect cycle in Directed graph



→ DFS will fail here
1, 2, 3, 4, 5, 6
7, 5 visited

⇒ have to be visited on Same Path
Without Backtracking

Find eventual Safe nodes



ans = [2,4,5,6]

6

5 terminal node (Safe node)
No outgoing

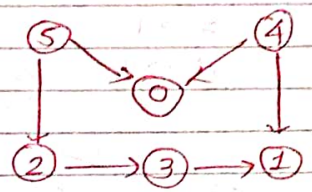
→ Search all nodes ending at terminal Node.

→ Part of cycle cannot be In ans ✗
any one leads to cycle ✗

Topological Sorting Algorithm (DFS)

In Direct Asyclic graph (DAG)

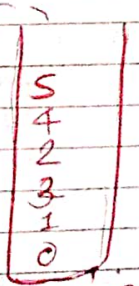
↳ linear ordering of Vertices such that if there is a edge between u & v, u appears before v in that ordering



5 → 0
4 → 0
5 → 2
2 → 3
3 → 1
4 → 1

5 4 2 3 1 0 ✓
4 5 2 3 1 0 ✓

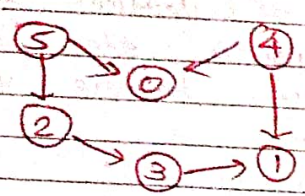
0 → { }
1 → { }
2 → {3}
3 → {1}
4 → {0,1}
5 → {0,2}



(5,4,2,3,1,0)

Topological Sorting (BFS)

Kahn's Algorithum



5 → 0
4 → 0
5 → 2
2 → 3
3 → 1
4 → 1

5 4 0 2 3 1 ✓
4 5 2 3 1 0 ✓

Topo Sort → Valid on (DAG) only

- Insert all Node with Indegree Zero (0)
- 4 → {0, 1}
  - decrease Indegree of 0 and 1
  - remove 4 in short

0	1	2	3	4	5
2↓	2↓	1↓	1↓0	0	0

Indegree

Incoming edges

1
3
2
0
5
4

Q

0 →
1 →
2 → {3}
3 → {1}
4 → {0, 1}
5 → {0, 2}

4 5 0 2 3 1

Course schedule I and II

Based on topological sort
Search study by yourself both
Code at

"github.com/vinayak501/DSA-CPP/Course_schedule"

Alien dictionary

	According to english
baa	abca
abcd	abcd
abca	baa
cab	cab
cad	cad

Find out alien order

b d a c

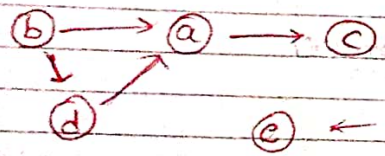
Something before Something

Topological Sorting

a b c d
0 1 2 3

baa
abcd
abca
cab
cad

(b)(a)(a)
(a)(b)(c)(d)
(a)(b)(c)(a)
(c)(a)(b)
(c)(a)(d)

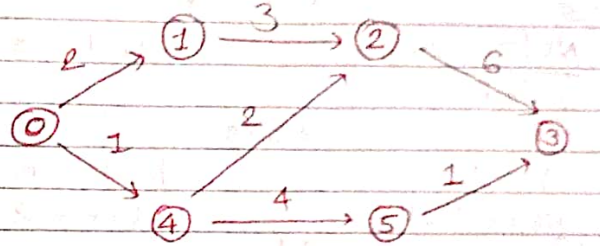


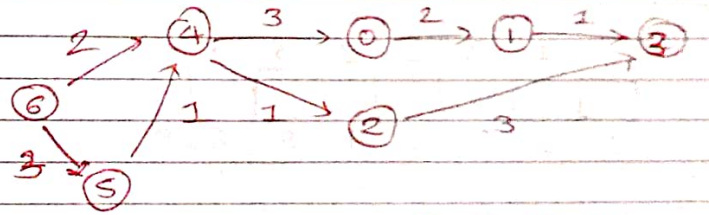
e ← if exist and no direction found

Topo sort

Shortest Path in DAG

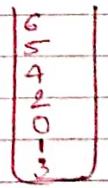
SHC = 0





0 → {1,2}
1 → {3,1}
2 → {3,3}
3 →
4 → {0,3}{2,1}
5 → {4,1}
6 → {4,2}{5,3}

0 step - 1
do a topo sort on DAG



6 5 4 2 0 1 3

→ Node out of stack relax edges

Node = 6 dist = 0

+3 +2

Node = 5 Node = 4

Node = 5 dist = 3

+1

Node = 4

dist = 4 > 2

Node = 4

+3 +1

Node = 0 Node = 2

dist = 5 dis = 3

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

Node = 2 dist = 3

+3

Node = 3

dist = 6

Node = 0 dist = 5

+2

Node = 1

dist = 7

Node = 1 dist = 7

+1

Node = 3

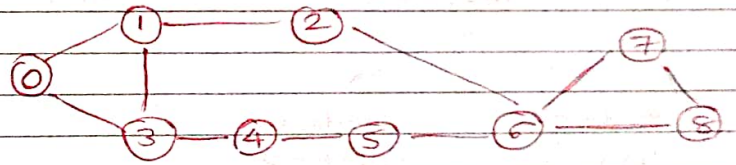
dist = 8 ×

Node = 3

dont update

dist = 2

Shortest Path In Undirected graph



{5,3}
{6,3}
{4,2}
{2,2}
{3,1}
{1,1}
{0,0}

Q

0 → 1,3
1 → 0,2,3
2 → 1,6
3 → 0,4
4 → 3,5
5 → 4,6
6 → 2,5,7,8
7 → 6,8
8 → 6,7

0	1	2	3	4	5	6	7	8
0	1	2	1	2	3	3		

{node, dist}

node = 0 dist = 0

node = 1 dist = 1
node = 3 dist = 1

node = 1 dist = 1

node = 0 dist = 2
node = 2 dist = 2
node = 3 dist = 2

node = 3 dist = 1

node = 0 dist = 2
node = 4 dist = 2

node = 2 dist = 2

node = 1 dist = 3
node = 6 dist = 3

node = 4 dist = 2

node = 3 dist = 3
node = 5 dist = 3

Same for all

Word ladder1

began word = "hit" end word = "cog"
wordlist = [hot, dot, dog, lot, log, cog]

hit → hot → dot → dog → cog

5

hit → 1

[ait, bit, cit, ... zit]
[hat, hbt, hct : hot ... hzt]

[hia, hib, hic ... hiz]

hot → 2

Same thing for this

→ 3

dot lot → 4

Brute force

⊙ Levelwise means BFS

set<string> wordlist

hit , 1
hot , 2

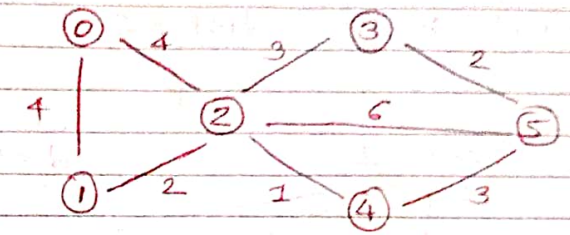
dot,3 lot,3

dog,4 log,4

cog,5 cog,5

log,4
dog,4
lot,3
dot,3
hot,2
hit,1

Dijkstra's Algorithm



0 → {1,4} {2,4}
1 → {0,4} {2,2}
2 → {0,4} {1,2} {3,3} {4,1} {5,6}
3 → {2,3} {5,2}
4 → {2,1} {5,3}
5 → {2,6} {3,2} {4,3}

dist[] =

0	1	2	3	4	5
0	4	4	7	5	10 8

Node = 0 dist = 0

+4 +4

Node = 1 dist = 4

Node = 2 dist = 4

{8,5}	7
{10,5}	6
{5,4}	4
{7,3}	5
{4,2}	3
{0,1}	2
{0,0}	1

min heap
{dist,node}

Node = 1 dist = 4

Node = 0 dist = 8

Node = 2 dist = 6

Node = 2 dist = 4

Node = 5 d = 10

Node = 0 d = 8

Node = 1 d = 6

Node = 3 d = 7

Node = 4 d = 5

Node = 4 dis = 5

+1

Node = 2 d = c

Node = 5 d = 8

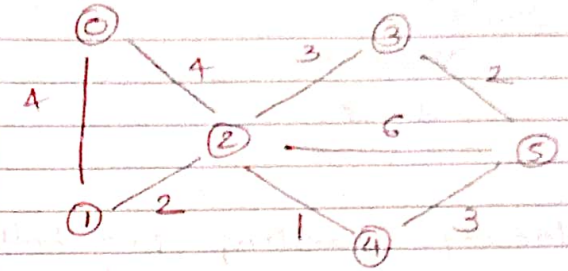
Same for all other Nodes

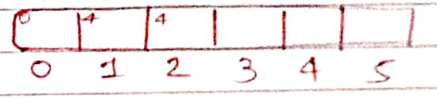
PQ

Priority_queue <Pair<int, int>, Vector<Pair<int, int>>, greater<Pair<int, int>>> pq;

T(c) → Elog v

Using Set data structure





Node = 0 d = 0

Node = 1 d = 4

Node = 2 d = 4

Node = 2 d = 4

+4 +2 +3

Node = 0 d = 8

Node = 1 d = 6

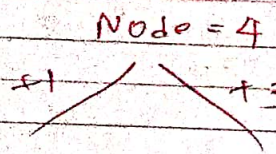
Node = 3 d = 7

+1

Node = 4 d = 5

{5,4}
{7,3}
{8,0}
{4,1}
{4,2}
{10,5} → source
{8,5} → shortest

{dist, Node}
Unique
min at top



erase already existing Non-better Paths

Theory Intution

Why Priority queue

⇒ Priority queue takes minimum distance first so all will be ~~max~~ min at first try after when taking max it won't change may get into queue

⇒ while in queue Number of unwanted occurance will be too much

$T(c) = \frac{E \log V}{\text{edges} \quad \text{Nodes}}$

```
while (! PQ.empty())                    → V
{
    dis,node = topP()                   → log heap
    for (iterat : adj(node))
    {
        if (conditional check)
            update
            push PQ                     → log (heap)
    }
}
```

no of edges = V-1 max

$O(V \times (\text{Pop Vert from min heap} + \text{no. of edges} \times \text{push into PQ}))$

$= O(V \times (\log^{heap} + ne \times \log^{heap}))$

$= O(V (\log^{heap}(ne+1)))$

$= O(V (\log^{heap}(V-1+1)))$

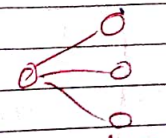
$= O(V \log^{heap} V)$

$< O(V^2 \log^{V^2})$

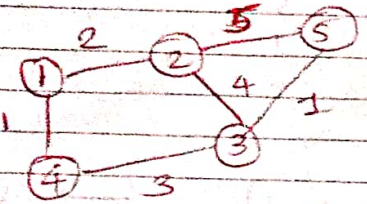
$= O(V^2 2 \log^{V})$

$= O(E 2 \log^{V})$

$= O(E \log^{V})$



Shortest Path in weighted Graph



src = 1
dest = 5

1 → 4 → 3 → 5
1 3 1
= 5

⇒ Remember where coming from

Parent

1	2	3	4	5
1	2	3	4	5

dist[]

0	2	4	1	5
1	2	3	4	5

{5,5}
{7,5}
{4,3}
{1,4}
{2,2}
{0,1}

{dist, Node}

```
while (Parent[node] != node)
{
    ans.add(node)
    node = Parent[node]
}
```

shortest distance In Binary maze

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

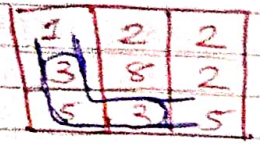
src = {0,1}
dist = {2,2}

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

{3, {2,2}} → d = 3
{3, {1,3}}
{2, {1,0}}
{2, {2,1}}
{2, {0,3}}
{1, {1,1}}
{1, {0,2}}
{0, {0,1}}

PQ → Q (already vis
{dist, {row, col}}

Path with minimum efforts



1 2 2 2 5 → max = 3

1 0 0 3

∴ ans = 3

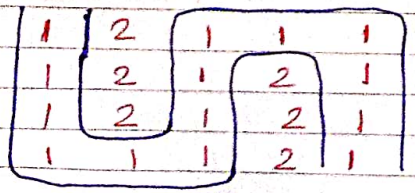
1 3 8 2 5 → max = 6

2 5 6 3

1 3 5 3 5 → max = 2

2 2 2 2

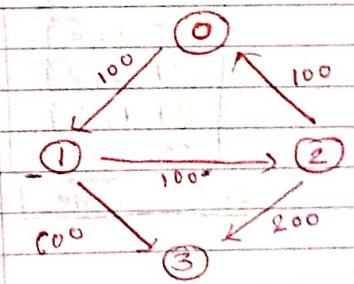
∴ ans = 2



ans = 0

(2, 2, 2)
(2, 1, 0)
(5, 1, 1)
(3, 2, 2)
(1, 1, 2)
(5, 1, 1)
(1, 0, 2)
(2, 1, 0)
(1, 0, 1)
(0, 0, 0)

Cheapest Path with K stops flights



src = 0 dest = 3

K = 1 only stops

0 → 1 → 3

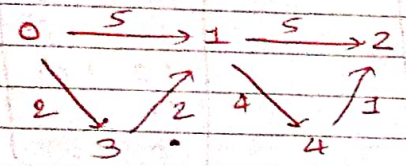
0 → 1 → 2 → 3

① ②

two stop ✗

⇒ Simple dijkstra will fail here

It will Pick minimum without Seeing steps or further Possibility to reach to destination.



src = 0
dest = 2
k = 2

dist[] =

0	1	2	3	4
	5	10 / 7	2	6

{3, 2, 7}
{2, 2, 10}
{2, 4, 6}
{1, 3, 2}
{1, 1, 5}
{0, 0, 0}

{stops, node, dist}

Minimum multiplication to reach end

[Set by Striver] GFG

Start = 3 end = 30

arr[] = {2, 5, 7}

3 → x2 → 6
3 → x5 → 15
3 → x7 → 21

[do %10^5 if n >= 10^5]

3 —x2→ 6 —x5→ 30

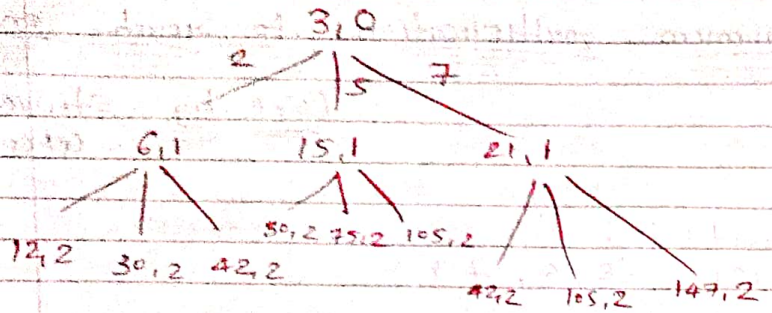
ans = 2

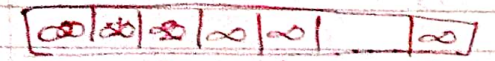
Start = 1 end = 66175

arr[] = {3, 4, 65}

7 x 3 = 21
21 x 3 = 63
63 x 65 = 4095
4095 x 65 = 266175 % 10^5
= 66175

3 step



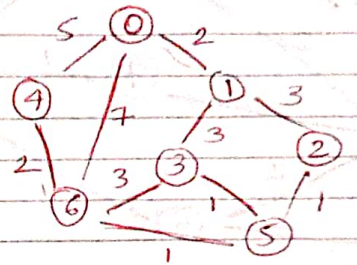


Brute force Solution

Simple Dijkstra

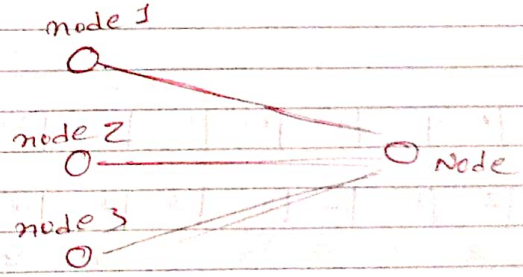
(1,42)
(1,30)
(1,12)
(1,21)
(1,15)
(1,6)
(0,3)

Number of ways to Arrive destination

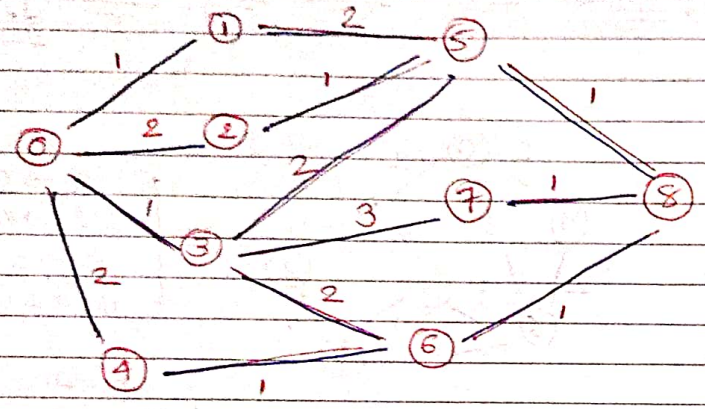


7 [
0 → 6
0 → 4 → 6
0 → 1 → 2 → 5 → 6
0 → 1 → 3 → 5 → 6

n = 7
src = 0 dest = 6



ways[node] = ways[node1] + ways[node2] + ways[node3]



ways

ways	1	1	1	1	2+1	2	1	3+2	
dist[]	1	2	1	2	3	3	4	4	
	0	1	2	3	4	5	6	7	8

{4,8}
{3,6}
{4,7}
{3,5}
{2,4}
{1,3}
{2,2}
{1,1}
{0,0}

min heap

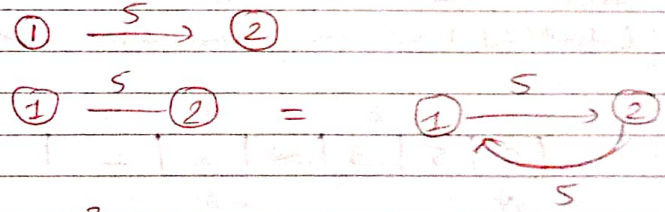
{dist, node}

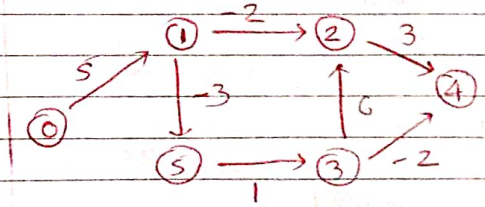
Bellmon ford Algorithm

⇒ Dijkstra fails at
- → Negative edge
- → Negative cycle

⇒ helps you to detect Negative cycle

⇒ only work in Directed graph





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

Any order

⇒ Relax all edges n-1 time Sequentially

```
if (dist[u] + wt < dist[v])
{
    dist[v] = dist[u] + wt;
}
```

	0	1	2	3	4	5
dist[]	0	∞	∞	∞	∞	∞

- first Iteration

dist[3] + 6 < dist[2]
dist[5] + 1 < dist[3]
dist[0] + 5 < dist[1] ✓ 1→5
dist[1] - 3 < dist[5] ✓ 5→2
dist[1] - 2 < dist[2] ✓ 2→3
dist[3] - 2 < dist[4]
[dist[2] +3 < dist[4] ✓ 4→6

0	1	2	3	4	5
0	5	3	∞	6	2

- Second Iteration

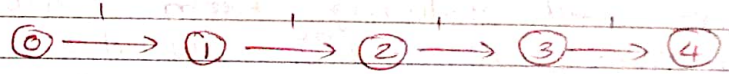
dist[3] + 6 < dist[2]
dist[5] +1 < dist[3] ✓ 3→3
dist[0] + 5 < dist[1]
dist[1] - 3 < dist[5]
dist[1] - 2 < dist[2]
dist[3] - 2 < dist[4] ✓ 4→1
dist[2] +3 < dist[4]

0	5	3	3	1	2

→ Same kind of Iteration for more 3 time

total = n-1 Iteration

Why n-1



n-1 = 4 Iteration

	u	v	wt
(	3	4	1)
(	2	3	1)
(	1	2	1)
(	0	1	-1)

- Iteration n-1

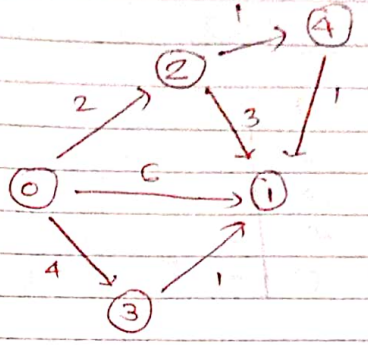
d[3] + 1 < d[4] 4th Iteration
d[2] + 1 < d[3] 3rd Iteration
d[1] +1 < d[2] 2nd Iteration
d[0] + 1 < d[1] 1st Iteration

0	1	2	3	4	
0	1	2	3	4	5

How to detect Negative Cycle

→ On n-1 iteration all edges Paths, distance will be relaxed max it can

→ If in n^{th} iteration distance is still reducing from our answers of $n-1^{th}$ iteration than there is a Negative cycle In the graph



→ go Via every Vertex / node

0 → n-1

d[0][1] = 0→1

0→2 + 2→1 = 5
2 3

0→3 + 3→1 = 5
4 1

0→4 + 4→1 = 4
3 1

Floyd Warshal Algorithm

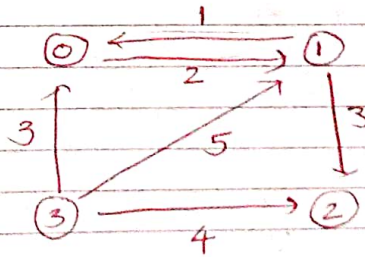
⇒ Dijkstra and Bellman ford were Single source Algorithm

⇒ floyd warshal help to find

one node to every single Node
↓
for every node

d[i][j] =

min [d[i][k] + d[k][j]]



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

Initial

→ Via Route 0

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

[1][2] = [1][0] + [0][2]
= 1 + ∞
= ∞

[1][3] = [1][0] + [0][3]
= 1 + ∞
= ∞

[2][1] = [2][0] + [0][1]
= ∞

[3][1] = [3][0] + [0][1]
= 3 + 2
= 5

[3][2] = [3][0] + [0][2]
= 3 + ∞
= ∞

→ Via 1 Route

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

[0][2] = 0[1] + [1][2]
= 2 + 3
= 5

[0][3] = [0][1] + [1][3]
= 2 + ∞
= ∞

[2][0] = [2][1] + [1][0]
= ∞ + —
= ∞

[1][3] = [2][1] + [1][3]
= ∞ + —

[3][0] = [3][1] + [1][0]
= 5 + 1
= 6 ✗3 ✗

[3][2] = [3][1] + [1][2]
= 3 + 5
= 8 ✗4 ✗

→ Same for
for via 2
via 3
↓
ultimate matrix
↓
Saving all shortest path from i to j

= matrix[i][j]

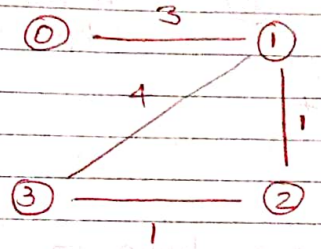
- How to detect Negative Cycle

If Costing Node from 0 → n-1

if matrix[n][n] = -Negative Value

Negative Value cycle

City with Smallest Number of Neighbours at a threshold dis



threshold = 4
city

0 → 1, 2
1 → 0, 2
2 → 0, 1, 3
3 → 1, 2

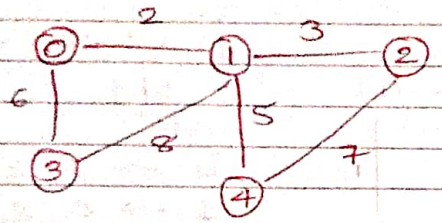
ans → 3

→ If we want to use dijkstra apply it for every Node

	0	1	2	3	
0	0	3	4	5	Cnt = 0123
1	3	0	1	2	Cnt = 03234
2	4	1	0	1	Cnt = 01234
3	5	2	1	0	Cnt = 0123

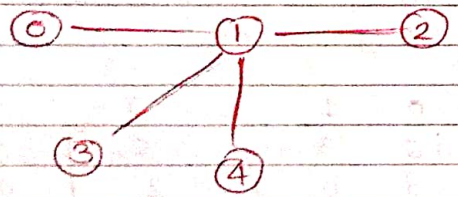
min = 543
city = 1 0 3

Minimum Spanning Tree → MST



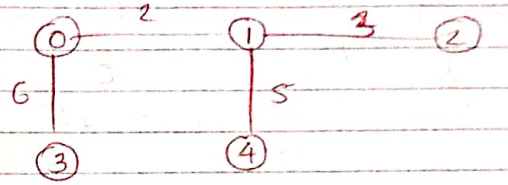
nodes = 5
edges = 6

⇒ Tree in which we have n nodes, n-1 edges and all nodes reachable from each other

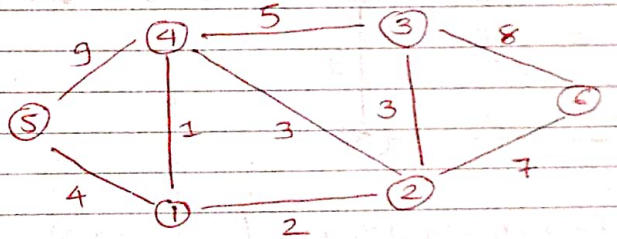


Sum of all edges of MST is it's weight

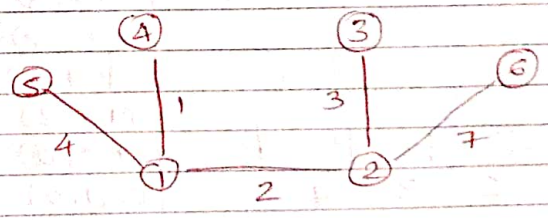
Smallst weight of Spanning Tree is called minimum Spanning Tree



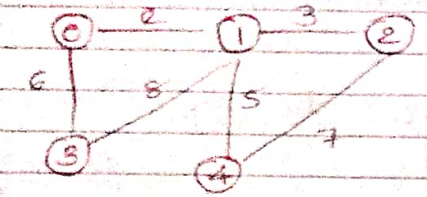
Sum = 5+6+2+3
= 16

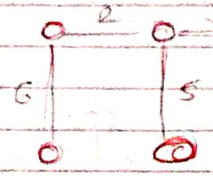


Sum = 17

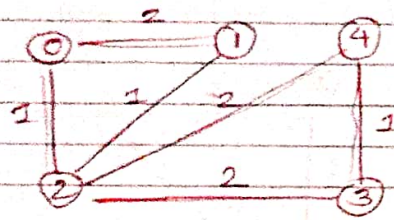


Prim's Algorithm (MST)





- 16



Sum = 0 1 2 4 5

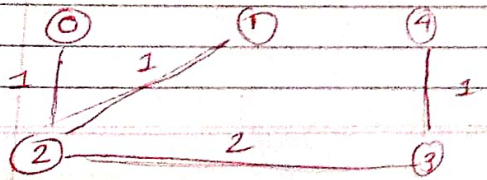
mst → [(0,2)(1,2)(2,3)(3,4)]

vis[] =

1	1	1	1	1
0	1	2	3	4

(2,2,3)	7
(1,4,3)	6
(2,3,2)	5
(2,4,2)	8
(1,1,2)	3
(1,2,0)	2
(2,1,0)	4
(0,0,-1)	1

(~~wt~~, node, Parent)
(min-heap)



Disjoint set

→ 

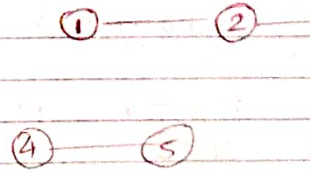
→ For determining if 1 & 5 belongs to same component

Pick 1 → do dfs if can't find 5, than it is NOT

O(N+E) time

⇒ Disjoint set helps us to tell this in O(1) time (Constant)

⇒ used in Dynamic graphs



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

Union(node1, node2) ⇒ Connect two nodes

Find Parent()

Union() — rank / size

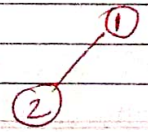
→ Union (P, Q)

1. Find ultimate Parent of u and v (Pu & Pv)
2. Find rank of Pu & Pv
3. Connect Smaller rank to Larger rank always

	1	2	3	4	5	6	7
rank	0	0	0	1	0	1	0	0
Parent	1	1	1	4	4	4	6	8

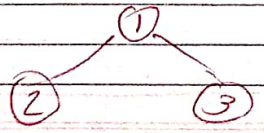
○ Union (1,2)

Pu = 1 Pv = 2
r = 0 r = 0



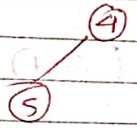
○ Union (2,3)

P(2) = 1 P(3) = 3
r = 1 r = 0



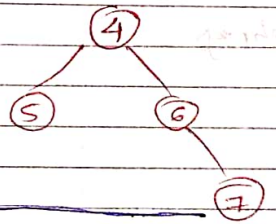
○ Union (4,5)

P(4) = 4 P(5) = 5
r = 0 r = 0



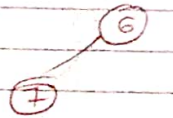
○ Union (5,6)

P(5) = 4 P(6) = 6
r = 1 r = 1



○ Union (6,7)

P(6) = 6 P(7) = 7
r = 0 r = 0

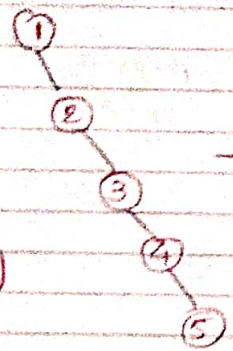


does 1 & 7 belong to same component

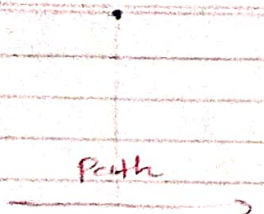
→ go to ultimate Parent of both
→ if same yes else no

→ Finding ultimate Parent takes $O(\log^N)$ time

→ Save ultimate Parents of all Node by Pre Computing to do Searching on $O(1)$ Constant time



O(log n)

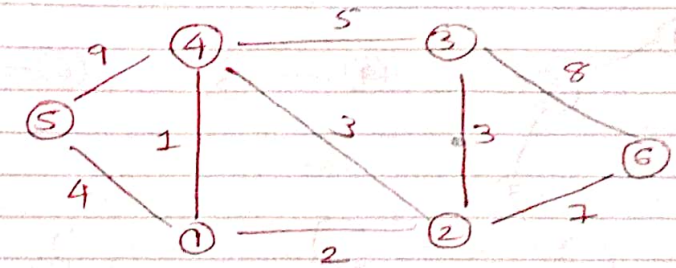


O(1)

⇒ Why Smaller to Longer

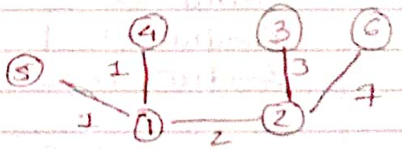
To reduce Size
→ Fast Searching

Kruskal's Algorithm [MST]



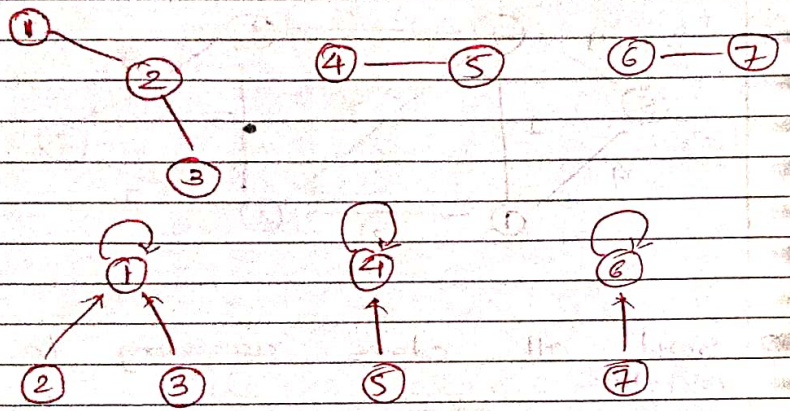
- Sort all edges according to weight

	wt	u	v
✓	1	1	4
✓	2	1	2
✓	3	2	3
X	3	2	4
✓	4	1	5
X	5	3	4
✓	7	2	6
X	8	3	6
X	9	4	5



wt = 1+2+3

Number of Provinces



findUP(1) = 1
findUP(2) = 1
findUP(3) = 1

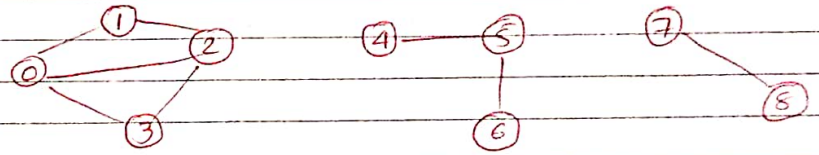
findUP(4) = 4
findUP(5) = 4

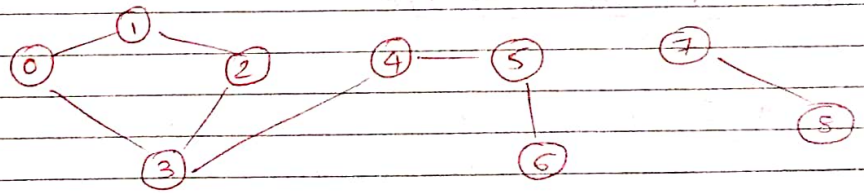
findUP(6) = 6
findUP(7) = 6

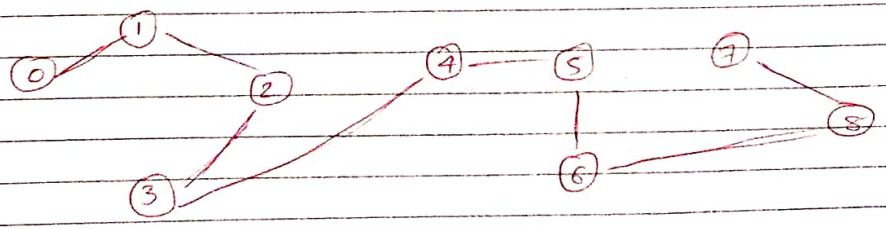
→ if (findUP(u) == u)
    count ++;

Minimum no. of operation to make a graph connected.

⇒ In each step
  ⇒ Can add edge blw two vertex
  ⇒ Can Remove edge blw two vertex
⇒ for connecting one edge have to remove one edge







2 step

- for n Component
  n-1 edges need to make fully Connected

→ no of Connected Component → nc

nc - 1 = ans

Count extra edges

if extra edges >= ans
Can be done

Account merge

0 → ["John", "j1@com", "j2@com", "j3@com"]
1 → ["John", "jh@com"]
2 → ["Has", "h1@com", "h2@com", "h3@com"]
3 → ~~["John", "j4@com", "j5@com"]~~
4 → ["Mat", "h2@com", "h3@com"]
5 → ["many", "m1@com"]

j1@com → 0
j2@com → 0
j3@com → 0
j4@com → 1
h1@com → 2
h2@com → 2
j5@com → 3
h3@com → 4
m1@com → 5

Code at
'github.com/Vmayakson/DSA-CPP'

Number of Island = II

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

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

How, col

↳ How x m + col

↓

node

ent = 0 1 2 3 2 1

cnt = 0 1 2 2 2 3 3 3 4 4

3 1 2 2 3 3

2 2

1 1

Code at
"github.com/Vinayak501/DSA-CPP/no_of_Island_2_DISJoint-CH.cpp"

Maximum Connected group

⇒ nxn 1,0

almost one cell from 0 to 1

find larget group of connected is

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

6 6

7

6+6+7+1 = 20

5 col, 6 How

30 Cell = 1 to 29

(How, col)

NodeNo = How * m + col;

most stone removed with same row and column

	0	1	2	3
0	✗		✗	
1				✓
2				
3		✓	✗	
4		✗		✗

Hashing

1	2	1	3	2

1 → 2
3 → 1
4 → 0
2 → 2
10 → 0

→ we can do this using cnt and increasing it at number appearing

→ O(Q×N)

Hash Array

	2	2	1									
0	1	2	3	4	5	6	7	8	9	10	11	12

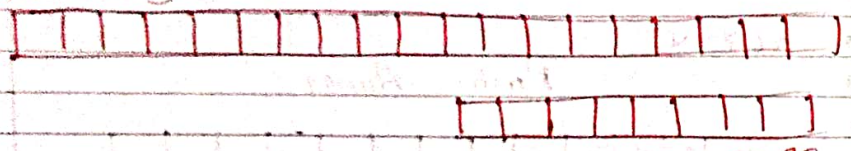
hash[1] = cnt of 1
hash[2] = cnt of 2

arr[10^6] → Inside main if max size
arr[10^7] → Globally

Chars Hashing

- Same it can be done by for loop also

s = "abcdabcefc"



int n = 'a'

n = 97

∴ int n = 'a' - 'a' } save like this
= 0
= 'b' - 'a'
= 1

→ for not doing 'a' - 'a' more memory of 256
↑
Include all

STL map, unordered_map

arr = 1, 2, 3, 1, 3, 2

map <key, Value>
↑ Number ↑ frequency

3 → 2
2 → 2
1 → 2

mpp[arr[i]]++;

→ map <char, int> for chars frequency Hashing

Time Complexity (map)

Storing } O(N) for all case
fetching }

for unordered_map

storing } O(1) best average
fetching }

O(N) worst rarely