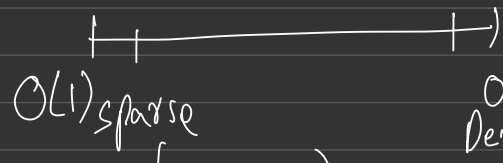
# Graph terms (N - vertices, M - edges)

**sparse, dense:** 1) For an N node graph we can have $O(N^2)$ edges right (edge b/w each node)

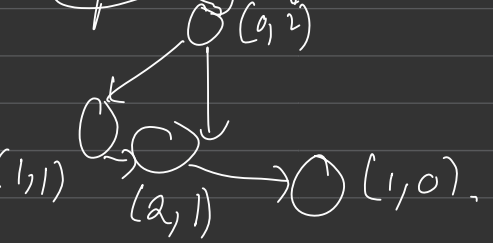
2) We can have $O(1)$ edges also.




$O(N\sqrt{N})$ → $N \leq 10^5$, $M \leq 10^5$ (sparse)

**Undirected:** Only degree.

**directed:** we have in/out degree.



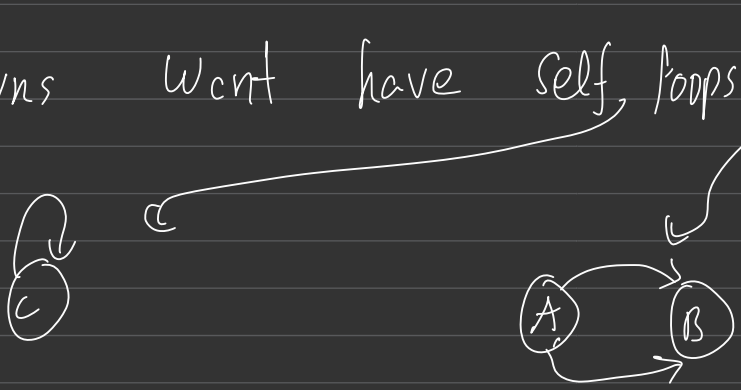

**Path:** → ordering of vertices which are reachable.

simple → u cant repeat nodes.

Isolated: No edge to any other node.

Reachable: if ∃ a path from node A → B.

Most q/ns won't have self loops / Mul. Edges.




Simple graph: Simple = graph with no self loops and no multiple edges!

Multi-graph: All graphs which allow self, Multi edges.

SUB-graph: $G(V, E)$, $G'(V', E')$

if $V' \subseteq V$, $E' \subseteq E$

# DAG : Directed acyclic graph.

No Cycles.

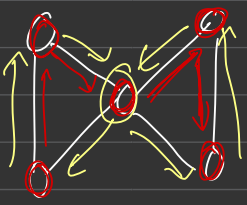
**Tree** : N nodes, N-1 edges.
∃ a path from A → B ∀ all A∈V, B∈V.

**Forest** : Collection of trees.

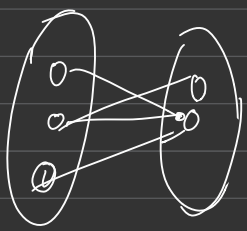
**Eulerian, Hamiltonian path** :

Eulerian: each edge visited exactly once. Yellow.

Hamiltonian: each node visited exactly once. red.



**Bipartite** :

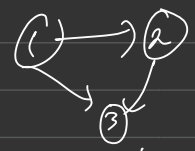


**Complete** : Every node connected to every other node!

$(m = {}^{n}C_2)$

## Representation

**Adj. Matrix** : N×N matrix, 1 to.



| M | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 |

↓ from

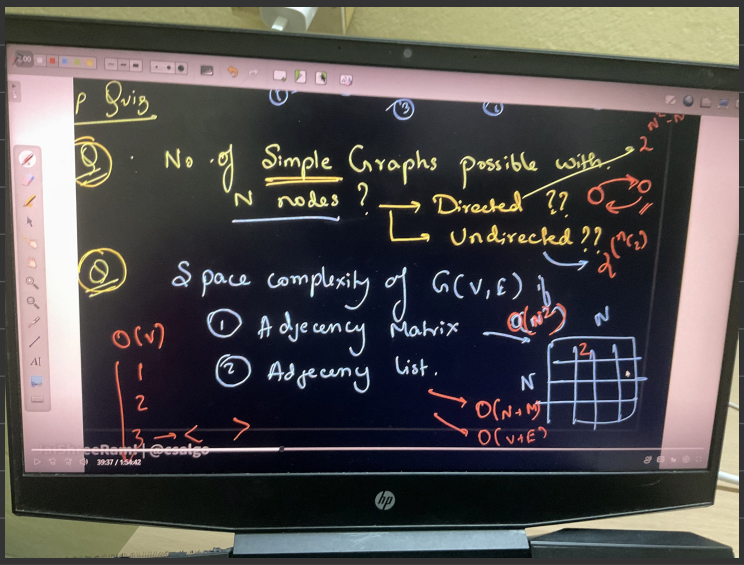
if undirected $M = M^T$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | 1 | 0 | 1 |
| 3 | 1 | 1 | 0 |

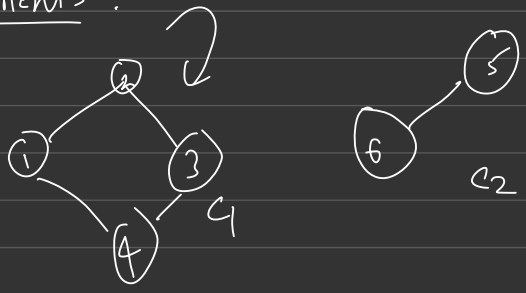
**Adj. List** : Compressed form.
→ * mostly used.

1 → <2,3>
2 → <3>
3 → < >

**Edge List** : V → {(1,2), (2,3), (1,3)} → *MST




**Components** :




$a_1a_2 + a_1a_3 + a_1 \cdot a_n + \dots$

$a_1($

```
for(i = 1 to K)
    for(j = i+1 to K)
        sum += a[i] * a[j];
```

$a_1, a_2, a_3, \dots a_n \rightarrow O(n^2)$

$O(n)$ $a_1a_2 + a_1a_3 + \cdots + a_1a_n$ using prefix sum,

↳ $+ a_2a_3 + a_2a_4 + \cdots + a_2a_n$

$+ \cdots a_{n-1}a_n.$

$a_1(a_2 + \cdots a_n) + a_2(a_3 + \cdots + a_n)$

$a_1(\text{sum} - \text{pref}[a_1]) + a_2[\text{sum} - \text{pref}[a_2]]$

$+ \cdots$

$\sum_{i=1}^{n} a_i(\text{sum}) - \text{pref}[a_i]$

No need of extra space. $O(N)$

$a_1(0) + a_2(a_1) + a_3(a_2 + a_1) +$

$\vdots$

$+ a_n(a_{n-1} + a_{n-2} + \cdots + a_1)$

Easily $O(N)$

$\text{ans} = \sum a_i(\text{sum upto } i - a_i)$

↓ easy O(N).

Used for # of edges that can be added s.t. # components of the graph can be decreased!

dfs(node, *component)

Now for dfs the parameters we give are really imp.

Q1)
Say we want what all nodes that are reachable / is there a path / no. nodes / components.

↓

grouping all nodes part of one dfs search into a component no. in visited list helps us a lot.

Q2) Say we want to find if a graph (no odd cycle) can be divided into a bipartite, then we could start dfs at node a, but we 2 colors.

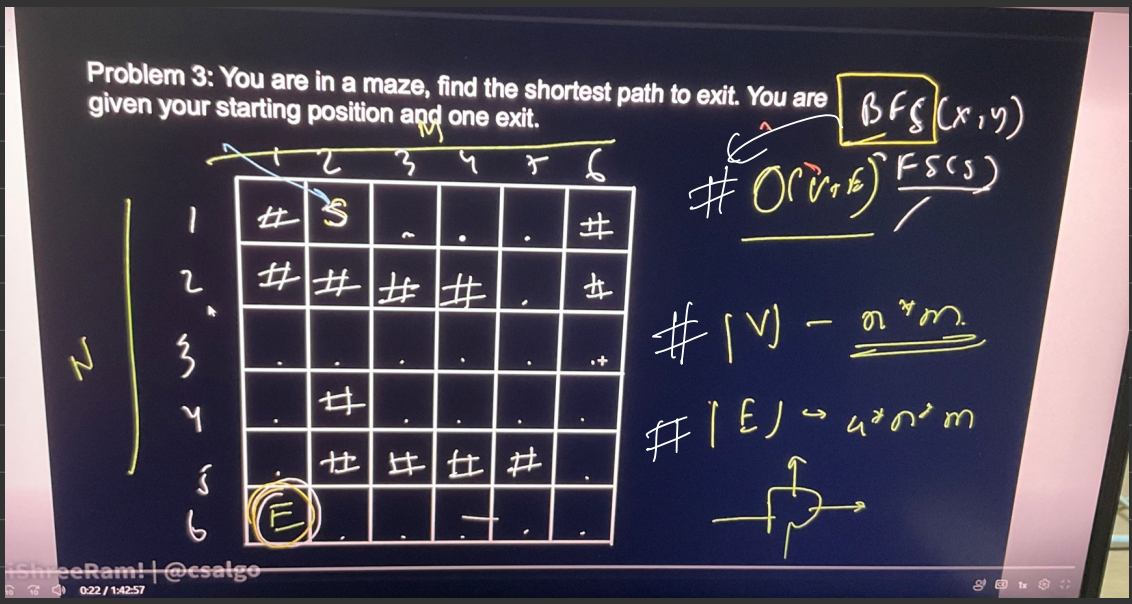
everytime we visit the uncolored neighbor we paint (what colour.) also if we encounter a visited node, and if it has same colour as the current node we stop and return false else its true.

here in dfs, only 2 colours are sufficient. ex '0' unvisited '1' or '2' one kind '3-x' another kind

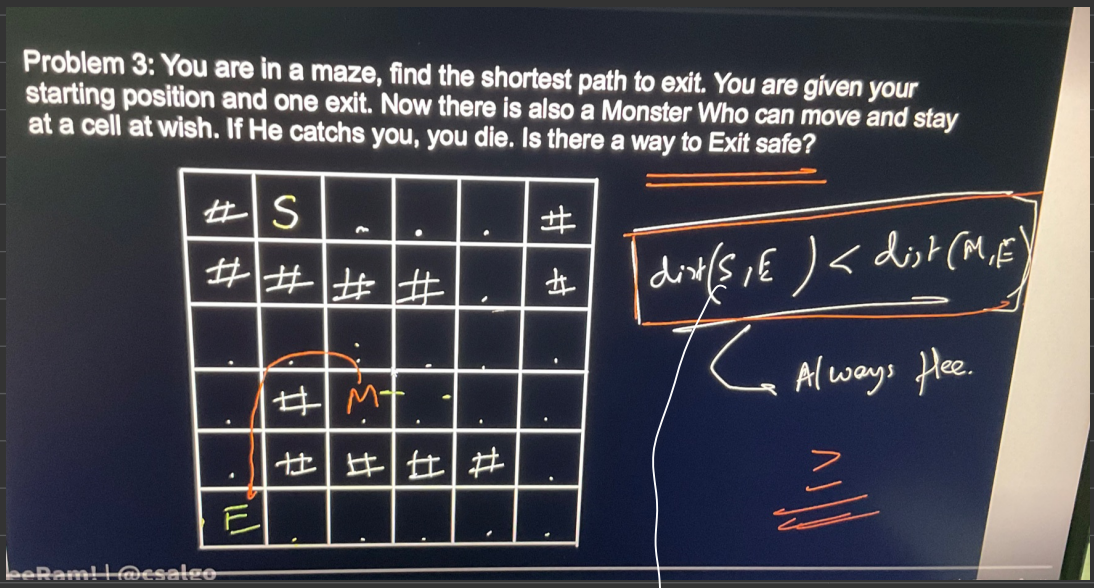
and we start with '1'

---

# MAZE Qns

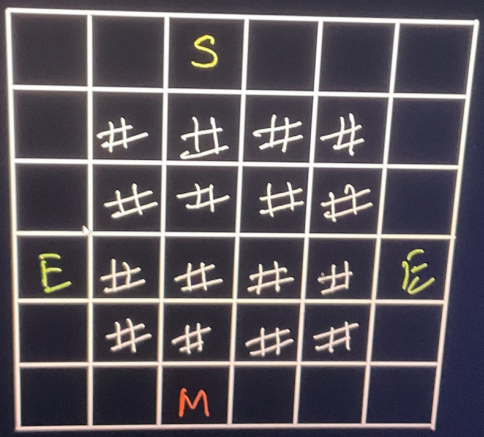



we can go only up, down, left, right.




he always survives else dies.

Will this strategy work for the case where there are multiple exits?



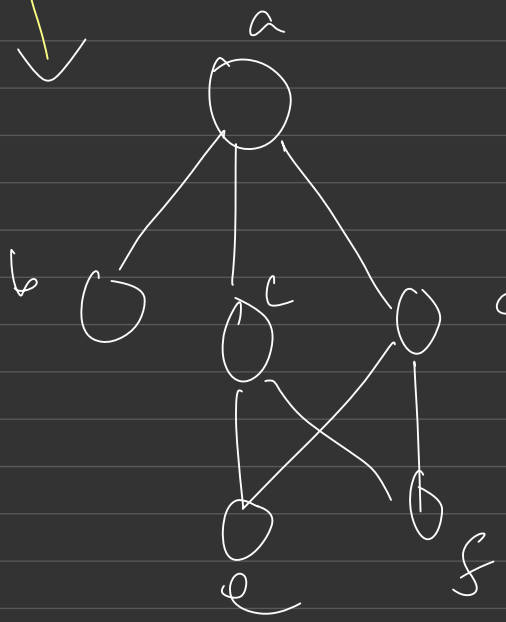

↓ No blud. if depends on real time positions of S, M!

if ∀ all $E_i$ $dist(M, E_i) < dist(S, E_i)$
then also there might be a way for the soldier to win.

# Coding BFS (Printing path also)

i) no recursion

ii) Before pushing into the queue we do all necessary things we are supposed to do. (marking / updating distance / updating path)




a

~~b c d~~

c d e f

by the time we reach d no unvisited node shld be left.

∴ A marked node is either in the queue or already visited.

iv) if we want distance, then we maintain a distance vector instead of visited

ex: vector <int> dist;
dist.clear()
dist.resize(n+1);

& also for every node we maintain a parent vector

vector <int> par;

## Gen. Settings for a grid (n x m grid)

We maintain no adj list in gen.
we maintain a 2D dist vector.

vector <vector <int>> dist.

( dist[x][y] gives distance pt (x,y) from source s.

we initialize all distances to $1e^9$.

dist = vector <vector <int>> (n, vector <int> (m, $1e^9$));

now we need a 2D vector that hold its parent as a pt.

∴ We need pairs

pair <int, int> pt.

Vector < vetor < pair <int, int>>> parent;

parent[x][y] gives us a pair which is the parent of (x, y) during bfs from s.

We maintain dx[] = {  }
dy[] = {  }
→ K possible dirn to move from a cell.

and during the inner loop for bfs,

```
curr = q.front();
q.pop();
exploring curr now.
for(i=0, i<k; i++){
condn for marking/changing dist/changing par
if Yes then we push it into the queue
if No we move to the next pt.
}
```

If no 2D grid and all.

just.

i) vector<vector<ints> graph → normal adj. list.

ii) vector<int> vis / dist also.

iii) vector<int> par.

---

NOTE: Make sure to clear all the graphs, dist, par before taking new test cases.

Main issue is how to create edges or model the graph s.t. the time complexity becomes small.

↓

goal is to make $|V|, |E|$ as less as possible

↓

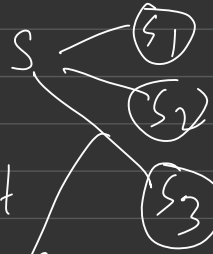
few techniques

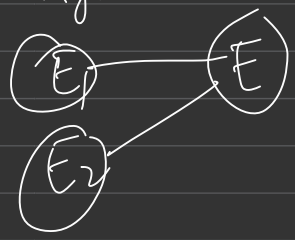
(i) adding dummy nodes so that bfs/dfs is possible

(ii) Changing weights smh. So that we can do bfs in less time.

i) multiple sources, multiple endings.

add one super S




now find shortest dist from $S \longrightarrow E$.

↓ Bruv here no need to create a new node s.

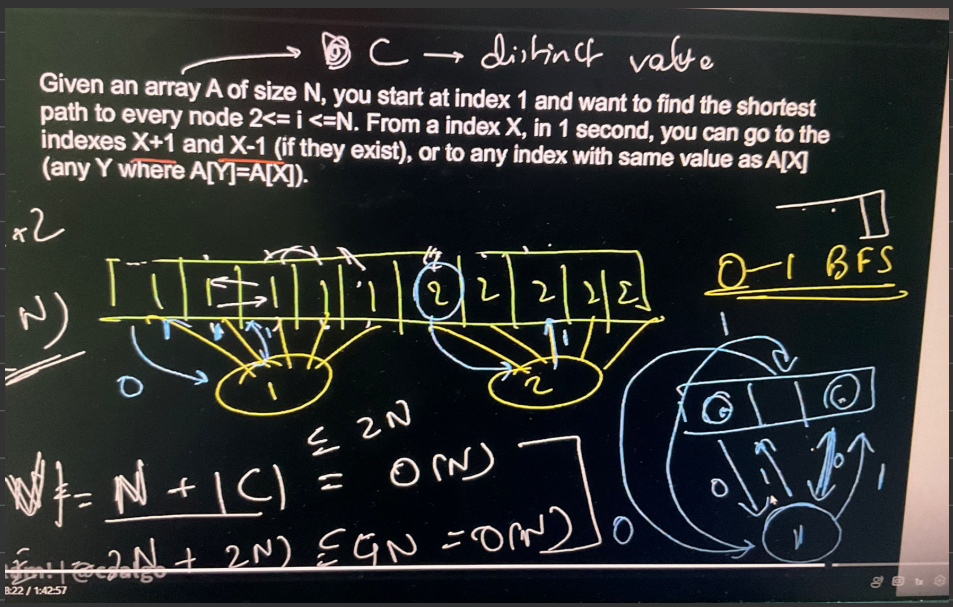
*.

→ just initialize $ds_1, ds_2, ds_3$ to zero and push them in the queue.

(all nodes effectively act as 1)

2) For intra traversal situations like moving from same colour to same colour nodes / teleportation etc instead of adding all the edges req, we could create a pseudo node that takes care of these paths.

ex:



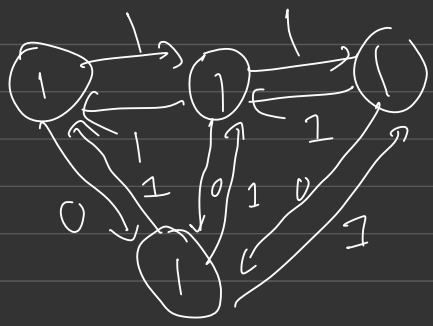

If we do brute force, then $|V| = O(n)$

$\therefore$ BFS becomes $O(n^2)$.

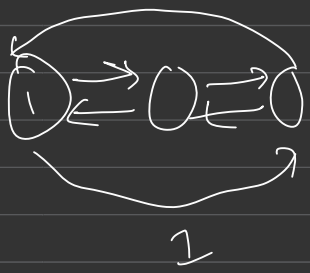
$|E| = O(n^2)$ when all nodes are same.

But we need to improve.

For that we add a pseudo node s.t. all the same values are connected to it.

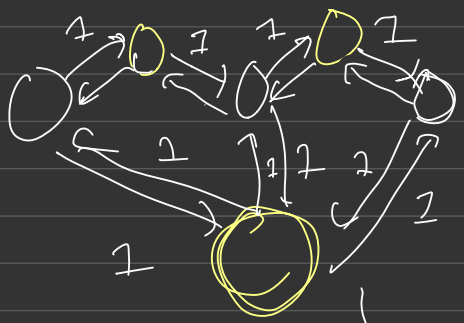


now it



→ we can apply 0-1 bfs to get sp.

If we dont want that 0 edge,



Now take 'newdist' as ans.
2

↳ apply normal bfs.

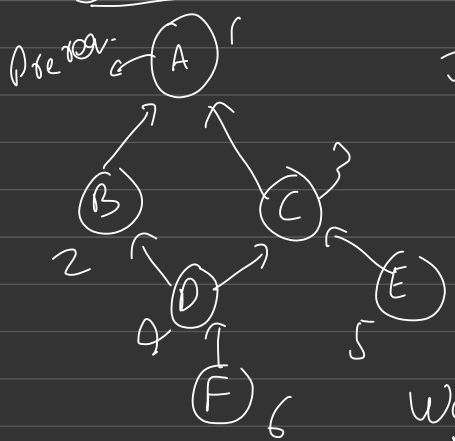
dijkastras, Floyd Welsh***

# Topological ordering

(** for directed acyclic graphs. (DAG))

if nodes have dependencies.

Courses:




for B we need to do A
for D we need B, C.

Multiple orders to do the entire thing.

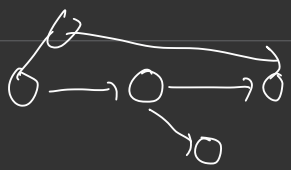
ex: A B C D E F
A C B E D F.

We need an efficient algorithm which is linear like $O(n+m)$

## 1) Using dfs:

(i) We do standard dfs but keep a topo vector.

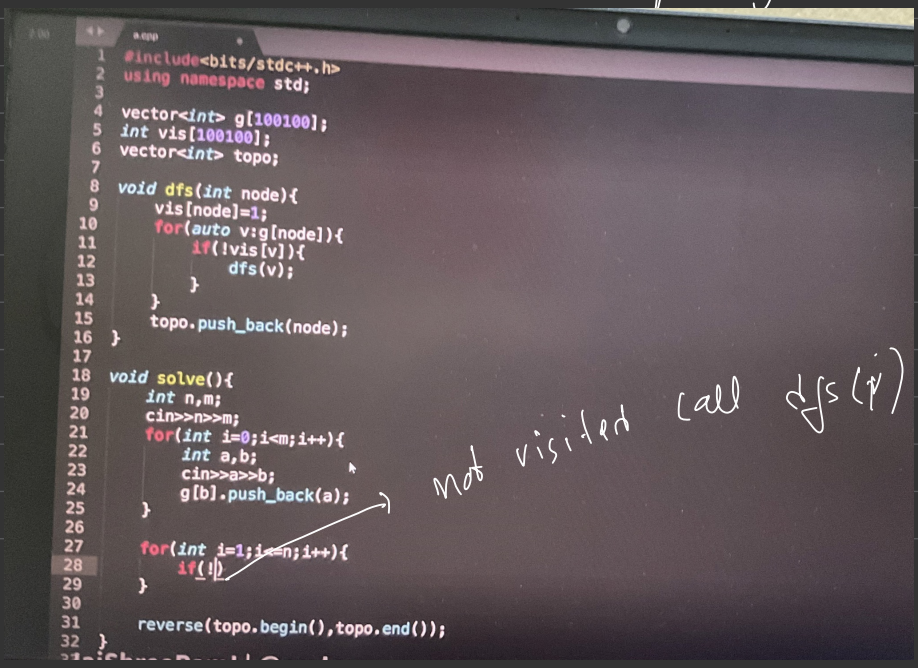
(ii) Also in our graph if B depends on A, then edge from

```
Void dfs(node){
    vis[node]=1;
    for(auto v: g[node]){
        if(!vis[v]) dfs(v);
        else (not possible as theres a
              }          cycle blud).
    topo.pushback(node);
}
```

↳ after exploring all nodes we push back.

Now reverse order of the topo vector is a valid topological ordering



## Khan's algo:

→ depends on in degree.
→ if a node has 0 indegree its mean that it isn't dependent right,

idea.
- we process those first and now make changes to its neighbours
- we go to all it neighbour decrease their indegree, add to queue if their deg becomes 0.
- after that we push them to our topo array.

## Applications:

(1) To detect cycles in directed graphs.
↓
run Khans, all vertices not in topo form a cycle.

(2) detect Longest path in DAG

dp + graph

should do again.

NOTE: In gen. Longest path is NP-complete

# SSSP

7-oct

0-1 BFS: (undirected / directed, 0,1, weights)

for weighted graphs,

neighbour → its weight.

```
vector < pair <int, int> > g[100100];
g[a].push_back({b,k});
```

When updated distance an pushing into the queue, if the weight of the edge was 0 we add it to the front of the queue not at the back.

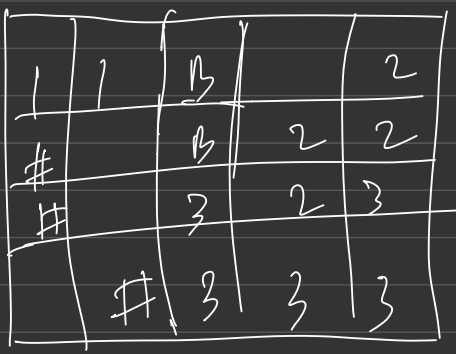
↓

y because essentially all the elements we are adding to the queue would having distance > the current level right, but if there was a zero edge, then its distance would be exactly equal.

Hence to maintain that invariant in the queue, we push it in the front.

**Problems**: Min no. of Bridges to cnct all components.

1—1: weight 0
x - empty cell - 1
# Cannot be visited




There is always a node "X" s.t.

$$\min(\text{shortest dist } X \text{ to } 1 + \text{Shortest dist } X \text{ to } 2 + \text{Shortest dist } X \text{ to } 3)$$

X Can be an empty cell

X Can be in a numbered node
↳ here just sum.

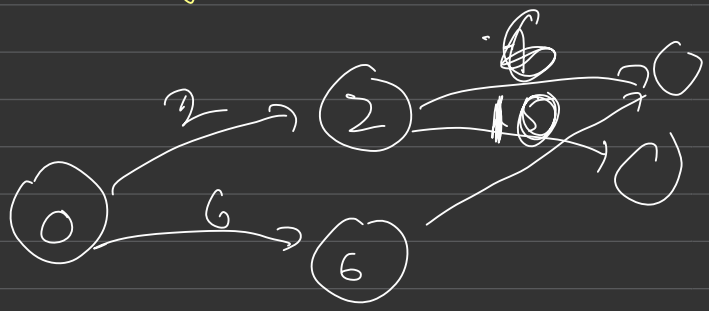
1 1 1
B
B B X B B
B B B 3 3 3

2 2 2
B
→ here we should do -2

# Dijkstra's Algo

$(E+V)\log|E|$

→ Doesn't work when there is a negative cycle.




→ a greedy algo. Every time we fix the dist to the shortest node & adjust to see if anything changes from that node.

→ We visit & check all the neighbours of node "only once" & "only when it is the current min" from the source s.

20-oct.

```cpp
void dijkstra(int start){

    // queue contains (-weight,node) pairs as we want q to be min dist sorted (hence the negative)
    priority_queue <ii> q;

    // lol we have to initialize start dist =0 else no use of applying the algo
    dist[start]=0;
    q.push({-0,start});

    while(!q.empty()){
        ii min =q.top();
        q.pop();

        // taking the node
        int node=min.S;
        if(vis[node]) continue;

        // we mark the node now, in bfs we use to mark before adding in the queue
        vis[node]=1;

        // we explore the nodes neighbour to change distances if any and if we have changed then only we push
        for (auto v: g[node]){
            int wt=v.S;
            int neigh=v.F;
            if(dist[neigh]>dist[node]+wt){
                dist[neigh]=dist[node]+wt;
                q.push({-dist[neigh],neigh});

            }
        }
        }
}
```

Problem:

$V \leq 1000$

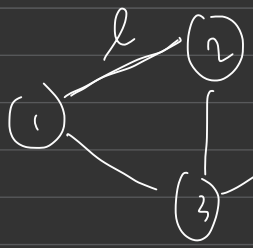
Capacity $\leftarrow$ $C \leq 100$

$E \leq 1000$

$P[i] \rightarrow$ Toll

$\downarrow$

cost/fuel.

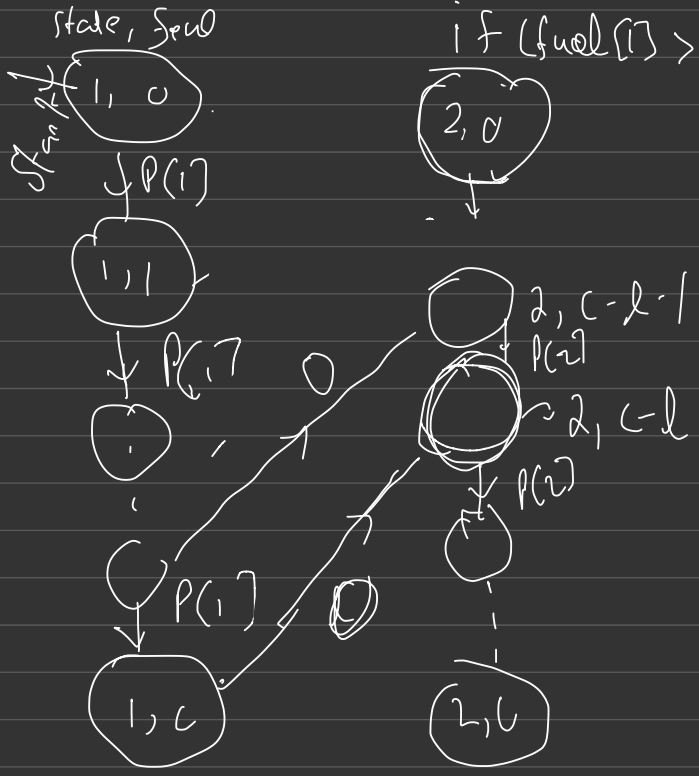



A car cannot exceed its fuel by C

k can travel when $C - l > 0$.

You have to find out the min. cost required to reach dest.

Basically u have to create a state space for the question with new nodes, new weights.




** Now run dikstras to get min cost!

But how to code?

→ Basically if dists are neg. and have a cycle, queue nevers get empty. and dikstras will not terminate

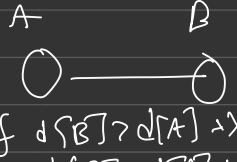
Now what to do in such cases bruv?

# Bellman - Ford → $O(V \cdot E)$ ≃ $O(V^3)$

↳ famous for finding neg cycles.
↳ not faster than dikstras.

→ Basically relax all edges |V|-1 times.

↳ check for an edge.




→ for (|V|-1):
relax all edges.

→ *Each iteration atleast 1 node will get min dist from S.

→ run 1 more time & check the nodes whose dist's change again, ⟹ they are in a negative cycle.

```cpp
int main(){
    }

    for(int i=1;i<=n;i++){
        dist[i]=1e9;
    }

    dist[1]=0;
    // we store edge list in bellman_fords algo
    // it is nothing but dp,assuming a graph has no neg cycles,max length of a path from any node to any node
    // is |v|-1;
    // therefore after |v|-1 iterations of optimising edges,the shortest distances shld come to an equilibrium
    // unless there is a neg cycle so each time we take the route through that cycle,dist decreses;
    // so therefore easy way to check a negative cycle is to perfom bellman_ford algo
    // bellmans optimization
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            if(dist[e[j].v]>dist[e[j].u]+e[j].w) dist[e[j].v]=dist[e[j].u]+e[j].w;
        }

    }

    bool isnegcycle=0;
    // checking if an edge can be optimized onece more;
    for(int i=0;i<m;i++){
        int u=e[i].u;
        int v=e[i].v;
        int w=e[i].w;
        if(dist[v]>dist[u]+w)  {
            isnegcycle=1;
            break;
        }
    }

    if(isnegcycle) cout<<-1<<endl;
    else{
        cout<<-dist[n]<<endl;
```

APSP. Floyd-Warshal → $O((V)^3)$

↳ kind of Brute force.

if we have Q queries

Q x y shortest path from x to y.

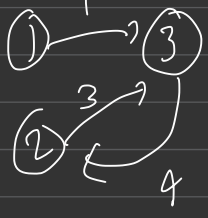
∴ If we run dijkstras, $V.((E+V)\log E)$

$\simeq V^3 \log V$ Bad!

→ Basically we use adjacency matrix,

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | ∞ | 1 |
| 2 | ∞ | 0 | 3 |
| 3 | ∞ | 4 | 0 |

→ d[i][j].




```
for K ∈ [1, N]
    for i from 1 to N
        for j from 1 to N
            dist[i][j] = min{ dist[i][j],
                              dist[i][k] + dist[k][j] }
```

Problems: 1) Shortest cycle using Floyd Warshall.

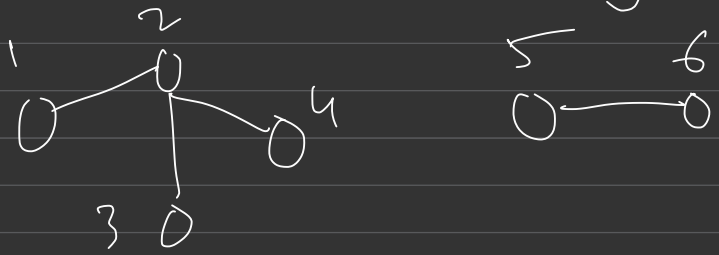
use same code but dont initialize dist[i][i] to 0.

If a cycle exists, dist[i][i] will be some non zero value & take min for all i.

30 - Nov

# Union - Find

basically a Data structure to associate a set of nodes.




1, 2, 3, 4 belong to a set of component

5, 6 belong to a different set

Now how to keep track of what all elements belong to which set?

We can use find(x) & merge(x,y) to accomplish all our problems.

So basically we start with a parent array.

initially while taking input, all nodes have themselves as parents.


par[] = [0 1 2 3 4]

0 1 2 3 4.

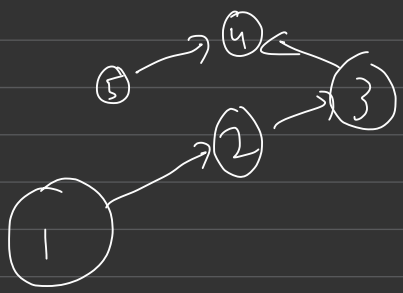
as the edges are given as input, we add them in our adj. list and also use the merge(x,y) fn.

Now what merge(x,y) does is

par[x] = y

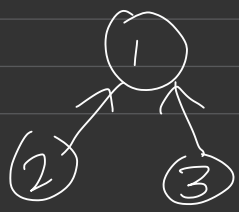

Find(x) is the root parent of the component node 'x' belongs to

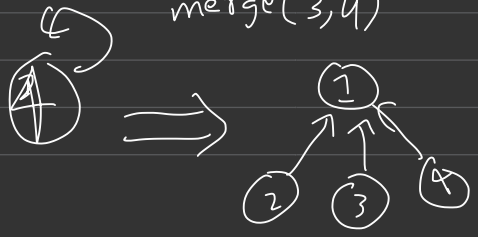
"call it the root representative"




Find(1) returns 4.

∴ Now merge(x,y) finds parents of x, y compares their size merges the lesser size parent to the larger size parent.

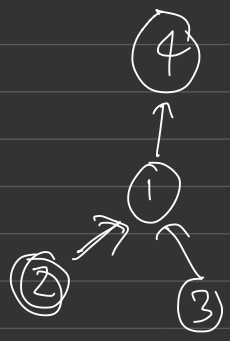


∴ We maintain a "size array for all nodes".

***

Not




also in Find(x): ⟶ complexity can be O(N)

Can we do amortization?

```
int Find(x):
    if (x == par[x]) return x;
    else { return par[x] = Find(par[x]); }
```

↓

This one line reduces complexity by soo much.

∴ After path, rank compression we can consider that

→ find(x), merge(x,y) take O(1) time.

→ also we maintain the size, each time we we merge we decrease size by 1.

gives # of components

initially to begin with,
size = n (number of nodes).

# MST

## Kruskal's Algo.

greedy.

Start with edges of small cost.

1) Sort
2) Check if x, y are already connected. i.e are part of same component
3) if not add edge / increase wt / increase edge cnt.
4) if already same skip that edge.

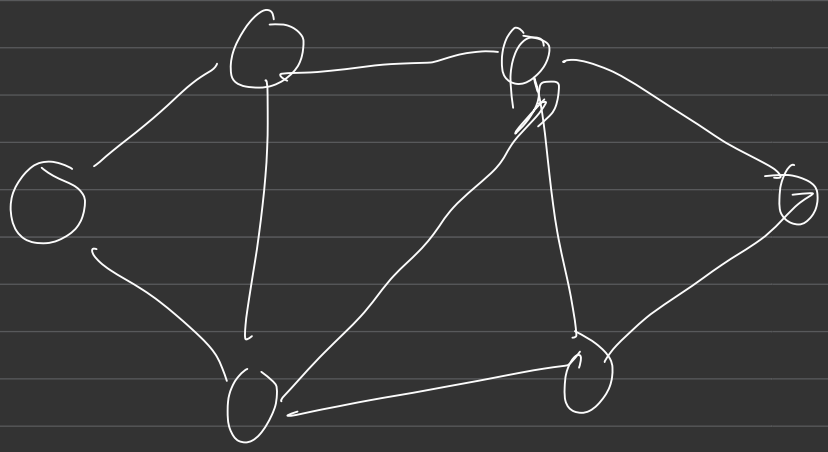
$O(E \log E + |E| \log\log |V|)$

this complexity final.

Q1) For finding Max spanning tree

- Sort reverse
- negate weights and sort.

Q2) Factories, roads.



each factorie has cost $c_i$ to produce food.
each road has cost $d_i$ to get transported.

Find optimal $c_i, d_i$ combination s.t. Total cost is min.

→ Now if u really think abt it u Cant even start a greedy strategy.

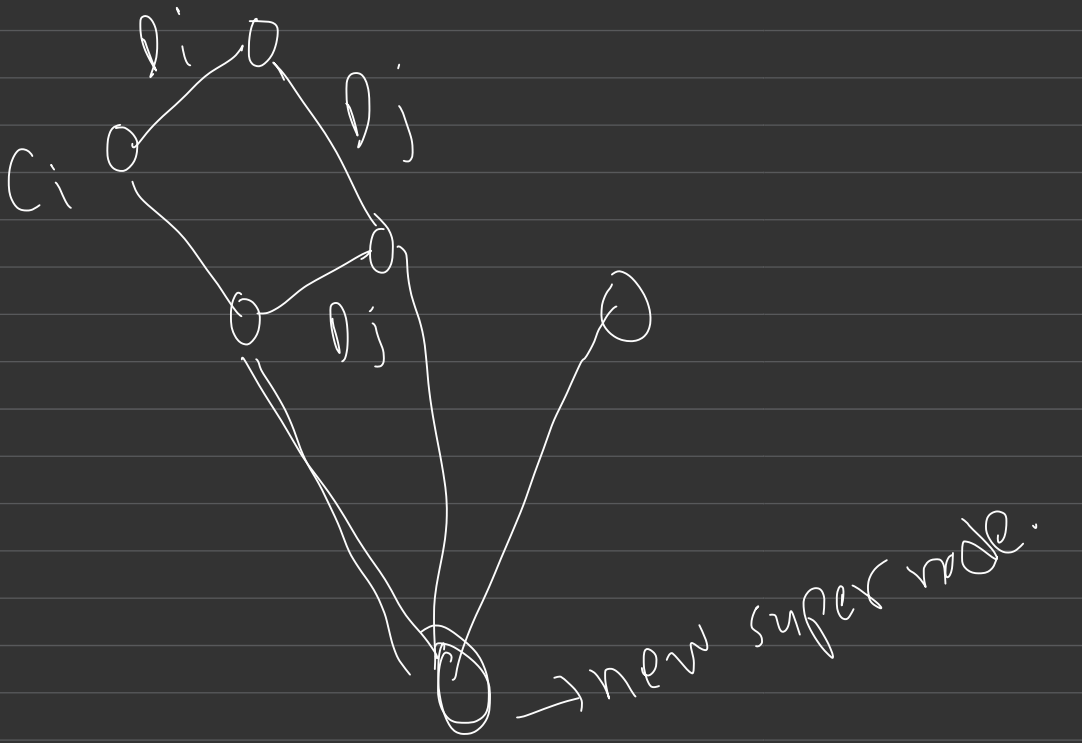
→ We know we need total of min 'n' $c_i, d_i$s to be picked.

Atleast 1 for $C_i$ & remaining $n-1$ for the spanning tree.

∴ Try adding a super node!

and connect all plants with their respective weights.

∴ Min spanning tree of new graph gives us the answer!!


$l_i$
$l_j$
$c_i$
$l_j$
new super node.