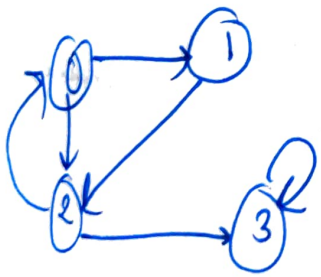
Graphs

```
public int V;
public LinkedList<Integer> adj[];

Graph(int v)                                  // constructor.
{   V = v
    adj = new LinkedList[v];
    for(int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

// To add an edge into the graph
void addEdge(int v, int w)
{   adj[v].add(w);
}

// To do the BFS structure traversal.
```



0 → 1, 2
1 → 2
2 → 0, 3
3 → 3

0 1 2 3] visited
F F F F

0 1 2 3] Rec.
F F F F

T F F F
T F F F
children: 1 2

T T F F
T T F F
children = 2

T T T F
T T T F
child 0 1 3

Directed cycle

Rat in Maze

0 — X

L, R, U, D

One can see in a path } Backtracking

- Backtrack & explore all paths from that cell.

Steps by knight

cell = (2, 2)

queue → all those at distance 1 from cell position.
all the four circles added to queue
pop the queue head & repeat
(adding 1 in any step for distance)

- BFS approach to the problem
- For the 8 directions check the conditions
- Queue operations have followed by the validations

Clone graph

Node
Val neighbours

Constructor
→ Val = 0, neighbours [ ]
→ Val = -val (update), neighbours [ ]
→ Val = -val, neighbours = -neighbours.

if (node → next)
  return

Queue q [Node → Node → Node → ...]

Hashmap map [Node Node
              Node Node
              Node Node]
              ...

nnode → node.val
        new ArrayList<>()

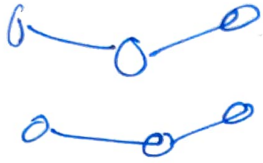
map → [node nnode]

while(q not empty)
{ temp = q.poll()
  for (Node i : temp.neighbours)
  { if (map.containsKey(i) == false)
    { q.add(i)
      map.put(i, new Node(i.val, new ArrayList<>))
                 node nnode
                 i   i.val <>
    }
    map.get(temp).neighbours.add(map.get(i))
  }
  return nnode;
}

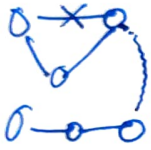
Make Network Connected

In connecting c nodes

C-1 edges are required to make them are connected (Min Span Tree Concept)



N = 6
E = 4
Here Redundant → 0



N = 6
E = 5
Red → 1

- Make adjacency list
- No - of components (c)
- Edges, Nodes, Compare } → Redundant - Edges

$$(\text{total Edges}) - [(n-1) - (\text{Components} - 1)]$$

When Red edge $\geq (c-1)$
Edges = $(c-1)$ required

N.g edg - $< (n-1)$
Red edges $< (c-1)$ } Negative Cases

To make network connected

visited = [F F F F . . .]

Hashmap(adj) [Int , <Int, Int ...>
Int , <Int ... >]

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

edgesCount = 0

n = connections[ ][ ].length

l1 = adj.get(0)
l2 = adj.get(1)

} Adjacency List Building

l1.add (1)
l2.add (0)

edges = 1

components = 0 // for components

if (node i not visited) [for i=0 to N-1]

components += 1;

DFS (adj, i, visited[ ])

}

if (edges < N-1)
return -1

redundant = edges - (N-1) - (components - 1)] the formula to calculate the redundant edges.

Now to check if components can be managed using redundant edges

if (redundant >= components - 1)
return components - 1;

return -1.

Dijkstra

```
int minDistance (int dist[], Boolean sptSet[])
{
    min = Integer.MAX_VALUE
    min_index = -1;
    for (int v=0; v<V; v++)
        if (sptSet[v] == false && dist[v] <= min)
        {
            min = dist[v];
            min_index = v;
        }
    return min_index
}
```

To find vertex with min distance value

min distance update

```
void dijkstra (int graph[][], int src)
```

adjacency matrix.

```
{
    int dist[] = new int[V];
```

shortest distance from src to i

```
    Boolean sptSet[] = new Boolean[V];
```

true if vertex i is included in shortest path tree

{Initialization}

```
    dist[i] = Integer.MAX_VALUE;
    sptSet[i] = false;
```

} i=0 to V-1

```
dist[src] = 0;
```

for (int count = 0; count < V-1; count++)
{ int u = minDistance(dist, sptSet);
(pick the min distance vertex.

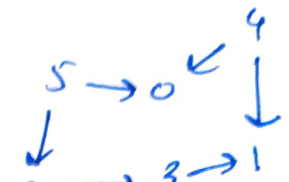
sptSet[u] = true;
for (int v = 0; v < V; v++)
if (!sptSet[v] && graph[u][v] != 0
&& dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v])
{ dist[v] = dist[u] + graph[u][v];

}
print dist → the final resultant array
}

Topological sort
Kahn's algo
BFS algo

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

→ Indegree of nodes



Step 1 → Check for 0 indegree & add to queue.

5
4

Now BFS on queue
For 4 neighbours are 0 and 1
Go there & decrement in degree by 1

Check for Step 1
Not Done for 4
Now go for 5

0 and 2 neighbours.
Decrement by 1
Check for Step 1
Yes → add to queue ...

1
3
2
0
5
4

Q is empty

4 5 0 2 3 1

→ Final Result.

→ impossible

Course Schedule [topological sort] -

[a, b] b is prerequisite for a

b → a

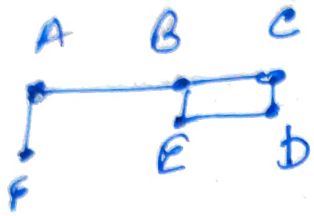
- The presence of cycle would make it impossible to complete all courses. Hence topological sorting.

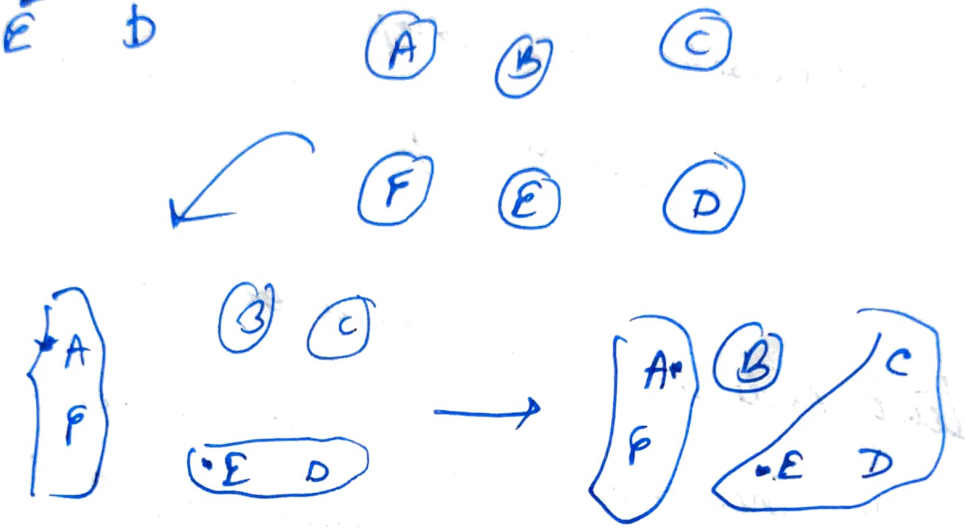
- Least node in topological ordering means no incoming edges [no prerequisite courses] Hence they can come before everyone, then after knowing them, next nodes to be processed.

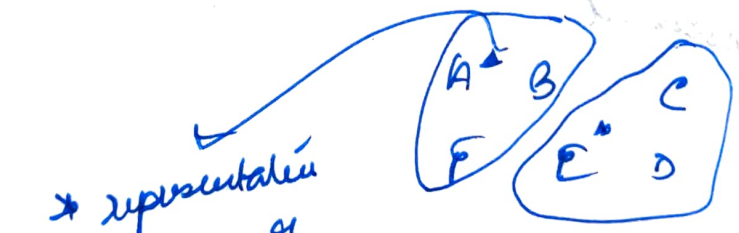
Cycle in Undirected Graph

Disjoint Set

Makeset Union findset







* representation
A is parent of the cluster

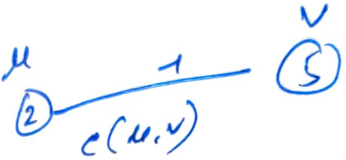
Bellman Ford

Negative edges → Dijkstra won't work

↓

Here Bellman Ford (DP approach)

- Go and relaxing all edges
- Go on for (V-1) times



if $(d[u] + c(u,v) < d[v])$

$d[v] = d[u] + c(u,v)$

$O(|E| \times |V-1|)$ → $O(n^2)$

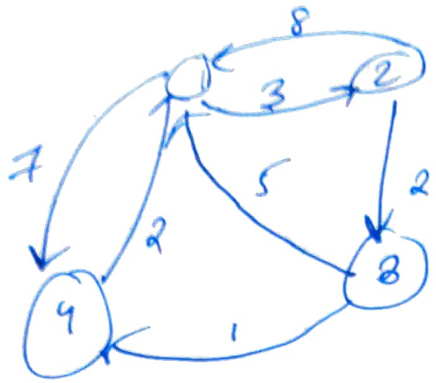
Worst (complete graph) → $O(n^3)$

If negative cycle → then a problem

↓

Then run for another iteration to detect the negative weight cycle.

Floyd Warshall



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

Using DP
↓
Decision to take whether shortest path via any middle vertex can be done.

for Vertex 1

	1	2	3	4
1	0	3	∞	7
2	8	0	2	(15)
3	5	8	0	1
4	2	8	∞	0

→ updated

Keep 1 as intermediate node

↙

$A^0[2,3] \rightarrow A^0[2,1] + A^0[1,3]$

(2) < 8 + ∞ = ∞

$A^0[2,4]$ $A^0[2,1] + A^0[1,4]$

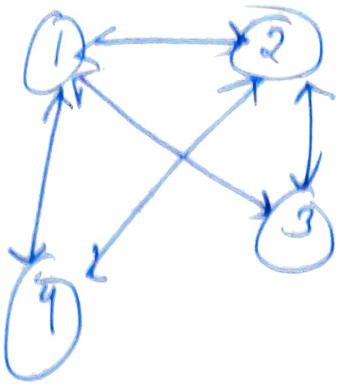
∞ > 8 + 7 = 15

$A^0[3,2]$ $A^0[3,1] + A^0[1,2]$

∞ > 5 + 3 = 8

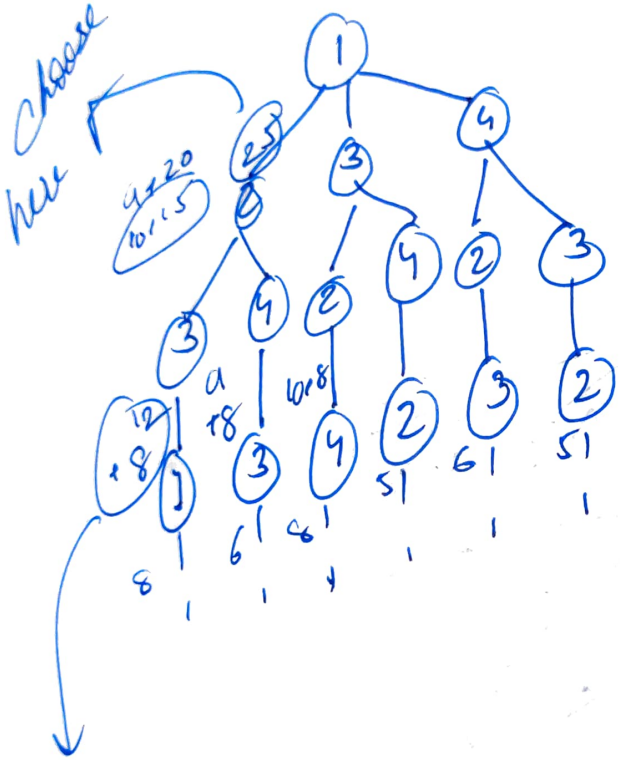
... continue.

Travelling Salesman



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

→ Go through all vertices & return to the starting point



$g(1,\{2,3,4\})$

$= \min\{c_{1k} + g(k, \{2,3,4\} - \{k\})\}$

$$g(i,S) = \min_{k \in S}\{c_{ik} + g(k, S-\{k\})\}$$

S = set of vertices

c_{ik} = cost i to k & cost of k to remaining vertices

$3 \to 4$

then $4 \to 1$

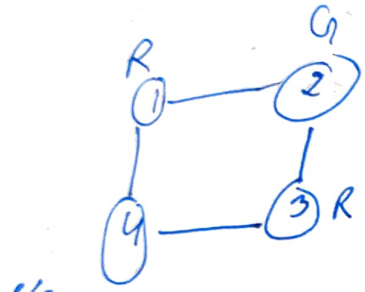
Hence $12+8=20$

Graph colouring - No 2 adjacent can have same colour

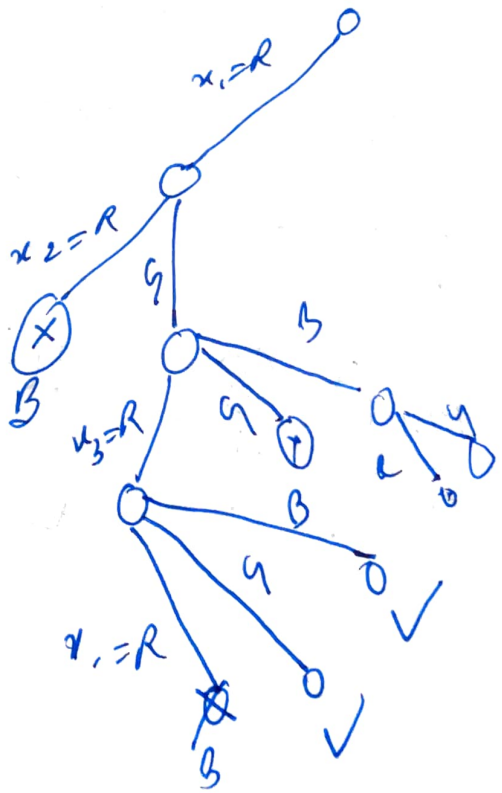
m-colouring decision problem. → whether colours are enough or not

m-colouring optimisation problem
↳ min colours

Backtracking - [State Space tree]



m = 3
{R, G, B}



- R G R G
- R G R B

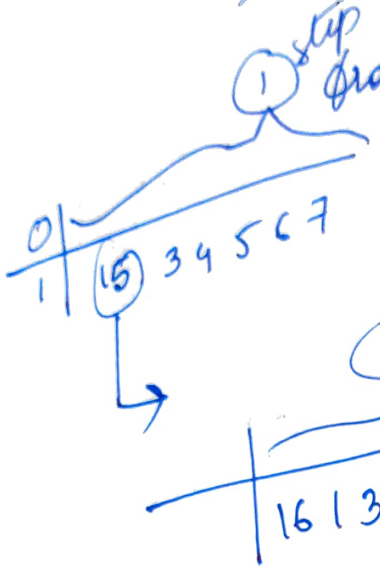
Snakes & Ladders

2 → 15
14 → 35
] Ladders

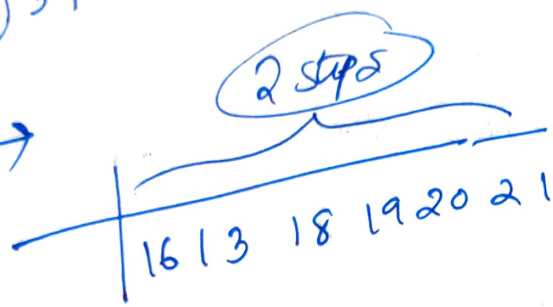
6×6 = 36
Board size.

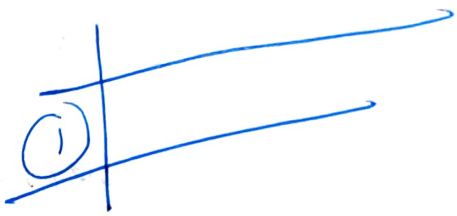
17 → 13) Snake

1 step from 1 can go the following



Simple BFS





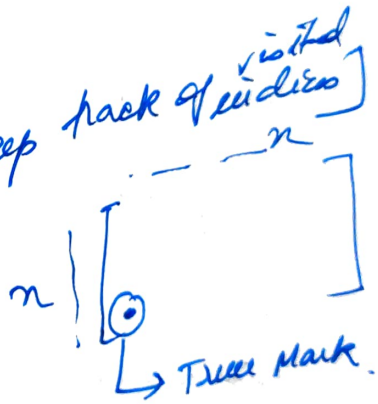
board [][]

n → board.length

steps → 0

Queue → Q

visited [][] → Boolean [to keep track of visited indices]



q.add(1);

while (q not Empty)

{ size → q.size

for (int i=0; i<size; i++)

{

x → q.poll()

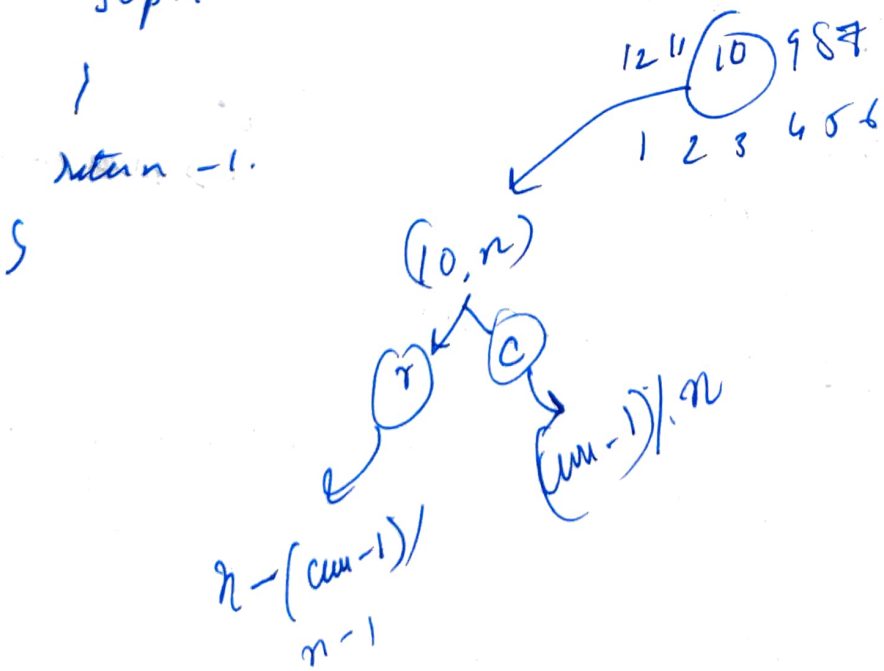
x ≠ 1, 2, 3, 4, 5, 6 [One at a time]

if (x == n²) return steps;

```
for( k=1 : k<=c : k++){
    if(k+x > n^2)  break;
    int pos[] = findCoor(k+x, n);
    int r = pos[0]
    int c = pos[1]
    if(visited[r][c] == true)
        continue
    visited[r][c] = true;
    if(board[r][c] == -1)
        q.add(k+x);
    else
        q.add(board[r][c]);
    }
}
step++;
}
return -1;
}
```

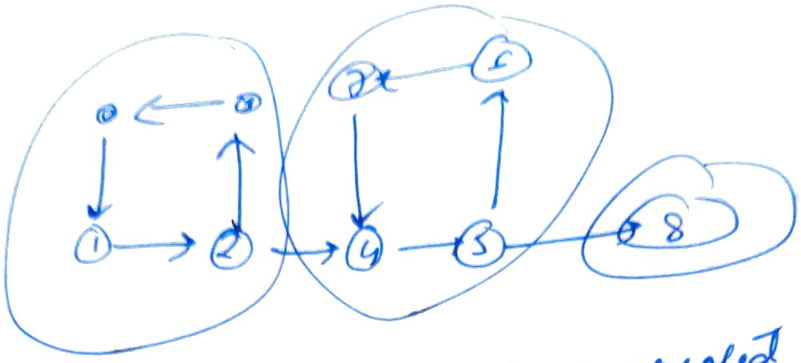
k+x	x
1+1	1
1+2	1
1+3	1
1+5	1
1+6	1

1 1 1 1 1 1
1 5 1 1 1



```
fuid Coor. {

r → n - (curr - 1)/n - 1;

c → (curr - 1) % n;

if (r%.2 == n%.2)
    return new int [] { r, n-1-c };

else
    return new int[] { r, c };

}
```

Kosaraju algo



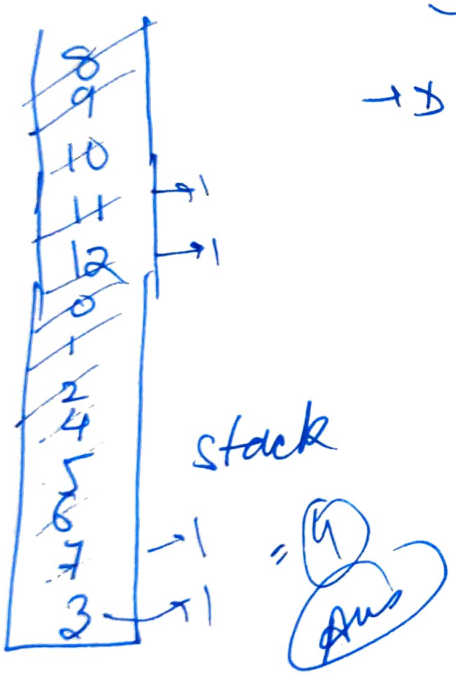
→ 3 strongly connected components

→ Directed Graph

→ No of cycles ≠ No of SCC

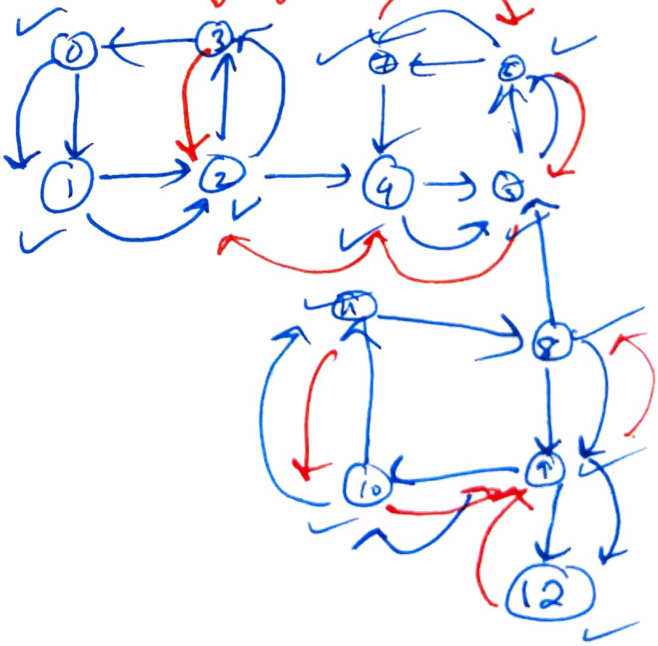
→ DFS → Specific Order.

(1)



Do DFS

Before Backtrack add to stack



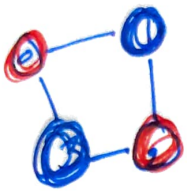
(2) New Graph with all edges reversed

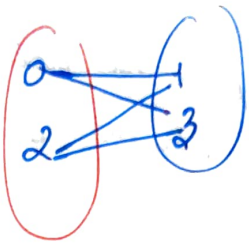
(3) In new graph, DFS start popping from stack

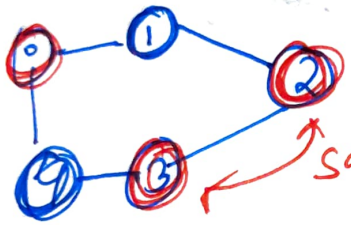
Bipartite graph.

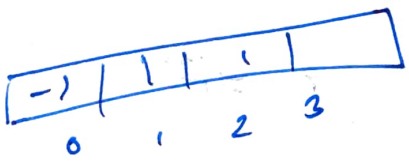
— Can we divide in 2 groups?

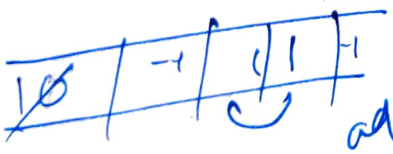
— If 2 adjacent diff colours → then we can separate











Journey to Moon

- 2 people
- Diff countries

No of Components

Given

List< ID Pair >

↓

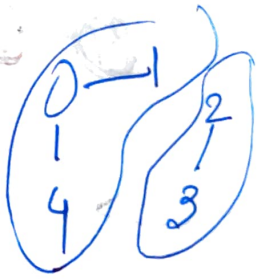
Each pair has astronauts from the same country

⇒ How many pair of astronauts from diff countries can they choose from?

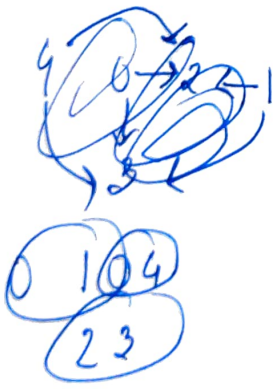
Ex $n = 4$

[1,2] [2,3]

→ 3 pairs



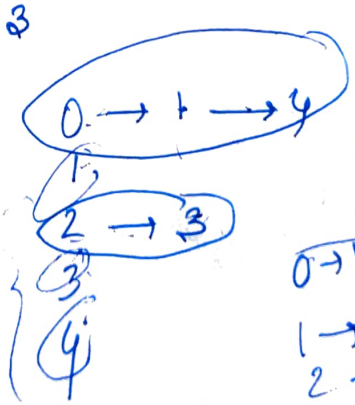
[1,2] [2,5]



1 → 2

2 → 3

1 → 2 → 3

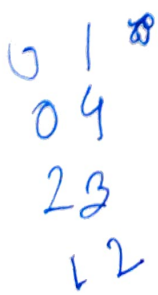


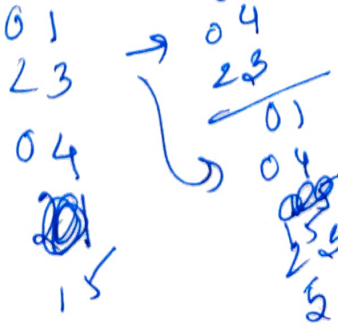
0 → 1 → 4

1 → 5

2 → 3

5 → 1





Cheapest flights within k steps

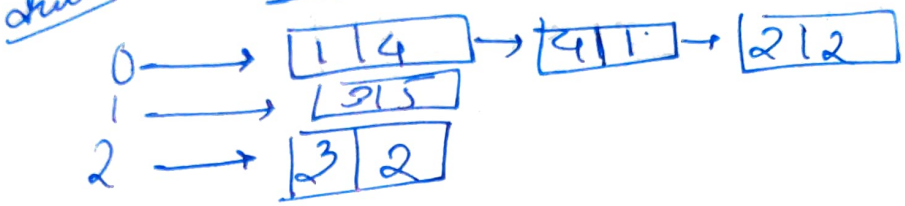
$$\begin{bmatrix} [0,1,4] \\ [0,4,1] \\ [2,3,2] \\ [1,3,5] \\ [0,2,2] \end{bmatrix}$$

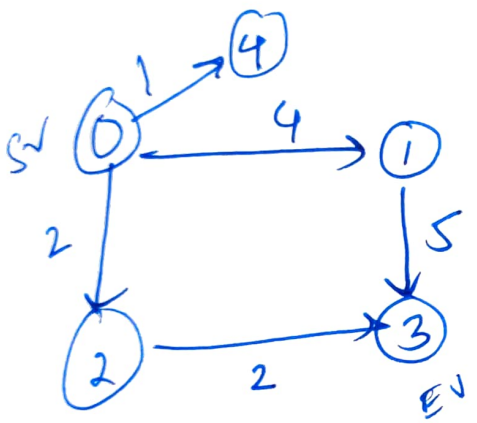
src → 0
dst → 3
k → 2

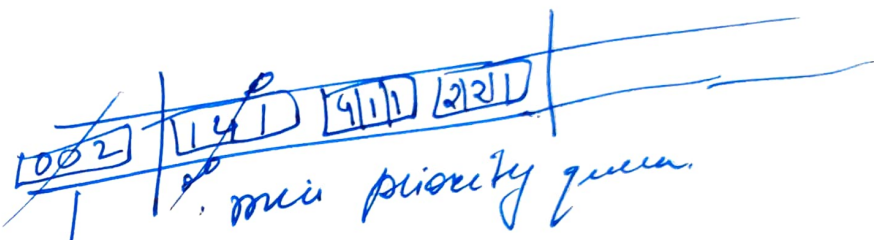
} min cost from src to dst

↓
maximum steps k

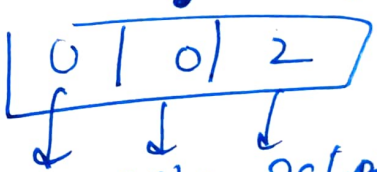
List array type





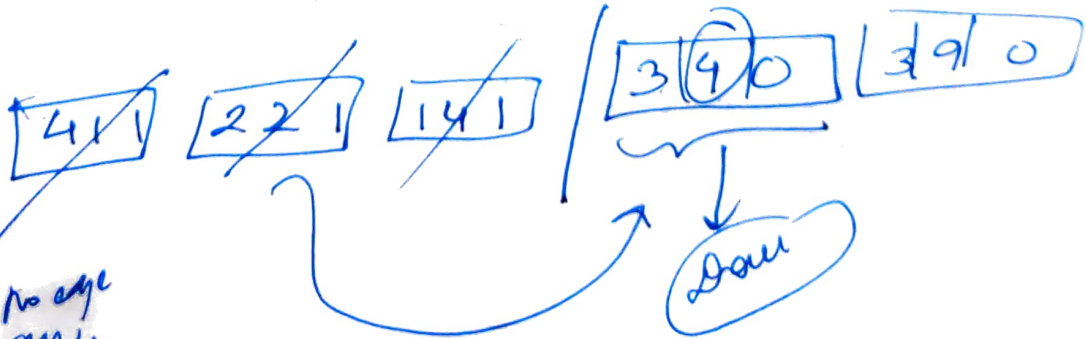


cost to be compared & sorted.



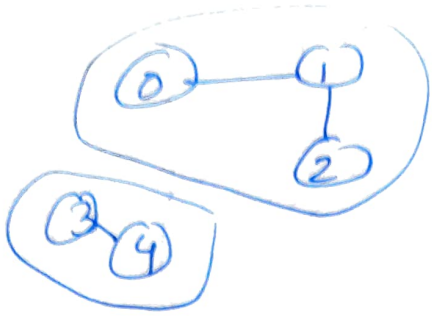
Initially all → ∞

- pop from min heap
- Ordered in terms of cost of queue.



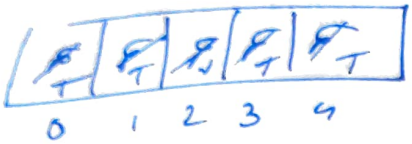
No edge from 4

Friend Circle

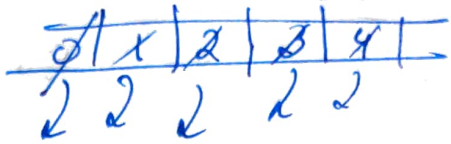


} Number of components

ans → 2



Visited

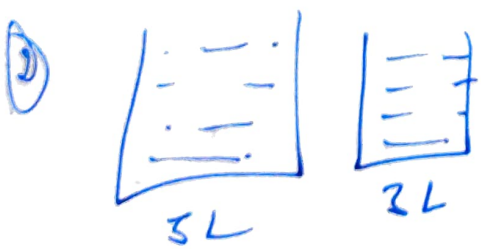


$^{12}C_2$

$\frac{9!}{2!}$

Water Jug

Can you fill the 3L with 1L of water

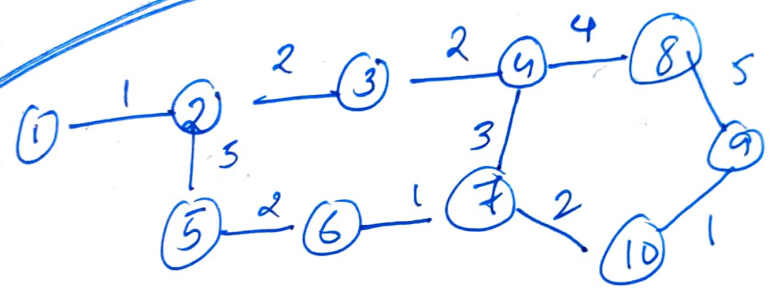


Can you fill 4L water in 5L jug

① → (0,10) → (4,0) → (1,3) → (1,0) → (0,1)
Initial Final

② → (0,10) → (0,3) → (3,0) → (3,3) → (5,1) → (0,1) → (1,0)
(4,0) ← (1,3)

Longest Path from the Initial to the chosen length atleast k



K = 23

Return TRUE

- Can't use cycle

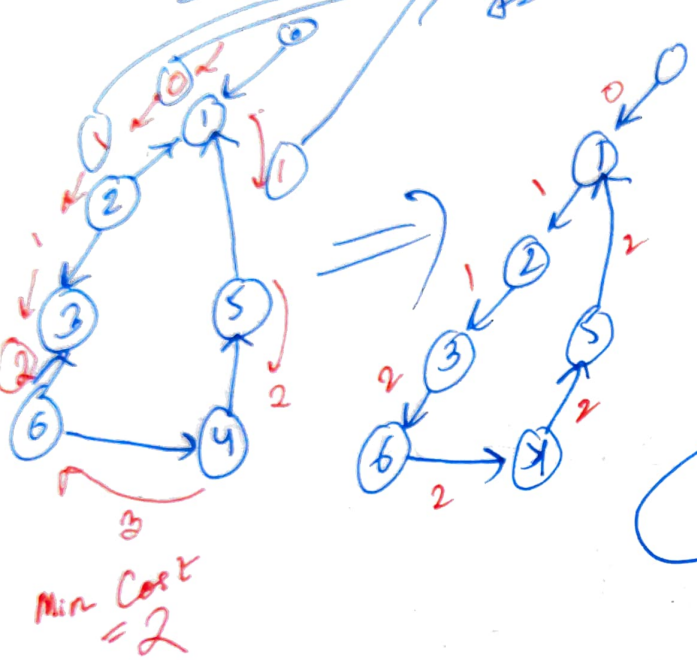
Backtracking

Concept

- Start with k
- If < k then decrement by (k - W) until k reaches 0 or W >= k
- Else Backtrack & make the node unvisited.

Min No of Edges to Reverse to make Path from Source to Destination

Will be graph weights



1-2 & 3-5 reversed

Dijkstra Problem

↳ if weight is variable [If fixed then BFS]

- first initiate graph
- Add weights by modifying graph.
- Then apply Dijkstra

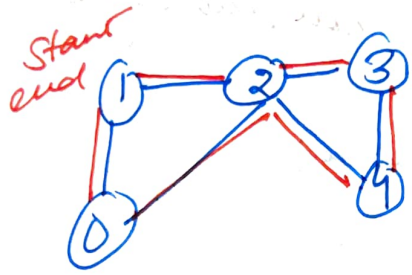
$d[u] + wt < d[v]$

Eulerian graph / Euler Circuit

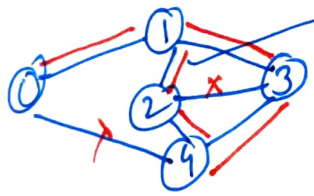
Walk → any random traversal

Trail → No edge is repeated
Vertex can be repeated

A trail which starts & ends with same vertex → Euler Circuit

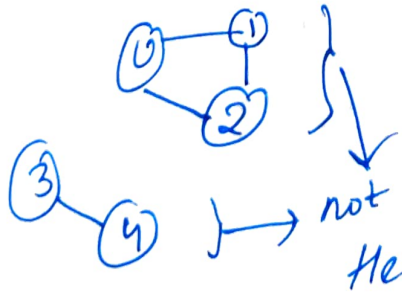


- all edges covered.
- Every edge visited only once



→ Blocked
Not Euler Circuit.

Euler graph → Graph having Euler Circuit



→ not all edges covered
Hence not Euler graph.

1. All vertices must have even degree
2. All vertices with non zero degree are connected in a component. Rest all vertices must have 0 degree

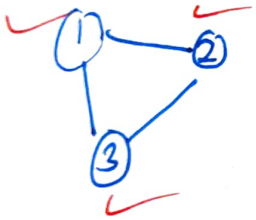
Euler Path - semi Eulerian

- Every edge once
- Start vertex ≠ End vertex

⇒ ① Exactly 2 vertices must have odd degree (start & end vertex)

② All vertices with non zero degree all connected.

→ Find node which has degree > 0



DFS call

0	1	2	3	Visited
F	T	T	T	

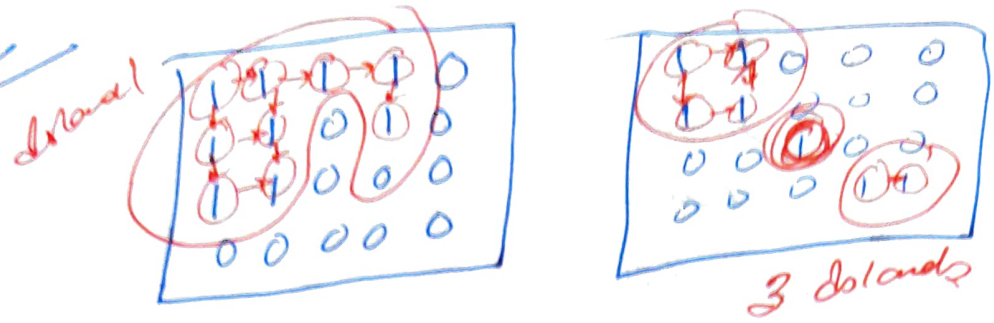
→ Node having odd degree [adjacency list]

count
- 0 → Eulerian
- 2 → semi Eulerian
- >2 → not Eulerian.

$O(V+E)$

```
int findEuler ()
{
    if (!connectedGraph ())
        return 0

    // count odd degrees
    int odd=0
    for (int i=0 ; i<V ; i++)
        if (adj[i].size() & 1)
            odd++;

    if (odd > 2)
        return 0

    return (odd == 0) ? 2 : 1 ;        1 → Semi
                                       2 → Eulerian.
}

bool connectedGraph()
{
    Boolean visited [V+1]
    for (0 → V)
    {   if (adj[i].size()>0)
        {   node = i                    // found node to start DFS
            break
        }
    }
    if node == -1
        return true

    DFS(node, visited);
    for (0 → V)
        if (visited[i] = False & adj[i].size > 0
            return false                // edges in multi component
    } return true.
```

No. of Islands



Visit all cells → can be done recursively

Make 1's 0's

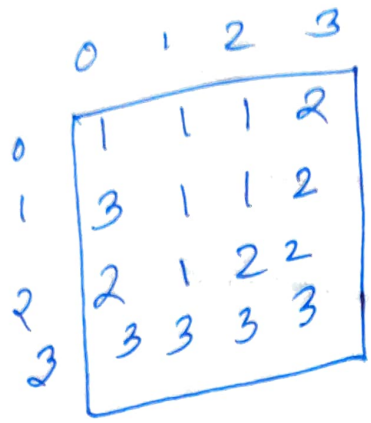
Ans ++

```
num Islands {
  (if grid[i][j] == '1')
      { dfs(grid, i, j)
        count++
      }
  return count.
}
```

Variation: Island Perimeter

```
dfs fun (grid, i, j)
{  if (i >= 0 & i < grid.length &
       j < grid[0].length & grid(i)(j) = 1)
{  grid[i][j] = 0
   dfs fun(grid, i+1, j)
   dfs fun(grid, i-1, j)
   dfs fun(grid, i, j+1)
   dfs fun(grid, i, j-1)
}  }
```

Flood fill



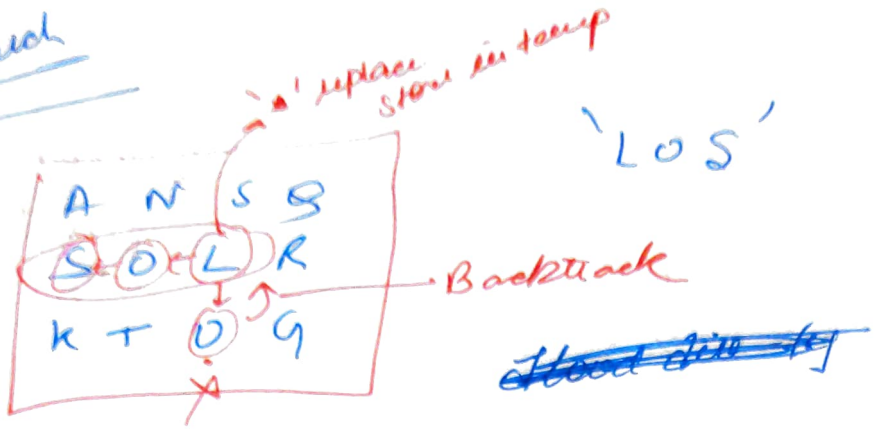
Remember Paint Bucket in Paint dost XD.

```
fill( image[][], srcRow, srcCol, newColor)
{
    if (newColor == image[srcR][srcC]
        return here

    DFS (image, src, sR, sR, image[sR][sC]);
    return image;
}

DFS ( . . . . )
{ if( row >=image.length || row < 0 || col >=image[0].length
        || col < 0 || image[R][C] != oldColor) return;
    image[r][cc] = color;
    DFS( image, row-1, col, color, oldColor)
```

4 dirs.

```
    .( . .  row+1, col . . . )
    . ( . .  row, col -1 . . . )
    . ( . .  row, col+1 . . . )
}
```

Word Search



~~Word diff by~~

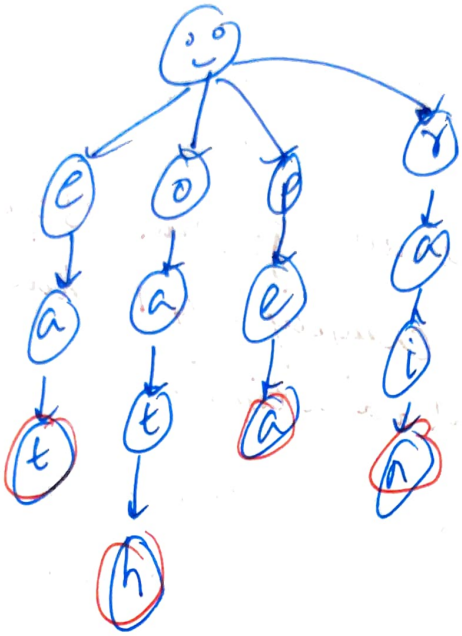
- No of board type
- Move forward
- 4 directions in the beginning
- Backtracking

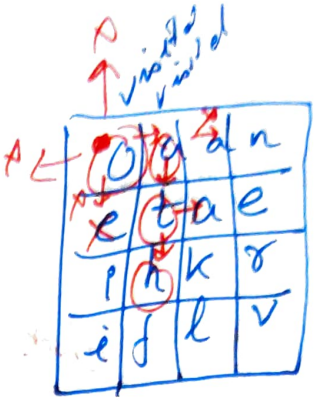
```
Boolean Exist( board [], String Word)
{
    for( i → board.length)
        for (j → board.length)
            if (board[i][j] == word.charAt(0)
                && dfs(board, i, j, 0, word)
                return true
                count

    return false
}

Boolean dfs ( . . . . . )
{   if(count == word.length)
        return true;
    if( i == -1 || i == board.length || j == -1 || j == board[0].
        length || board[i][j] != word.charAt(count))
        return false

    char temp = board[i][j]          for Backtracking
    board(i)(j) = '*';
    boolean found = dfs (board, i+1, j, count+1, word) ||
                    . . . i-1, j . . .
                    . . . (i, j+1) . . .
                    . . . (i, j-1)
    board[i][j] = temp  // if not found backtrack &
    return found.                replace by temp again
}
```

Word Search II

Using Trie

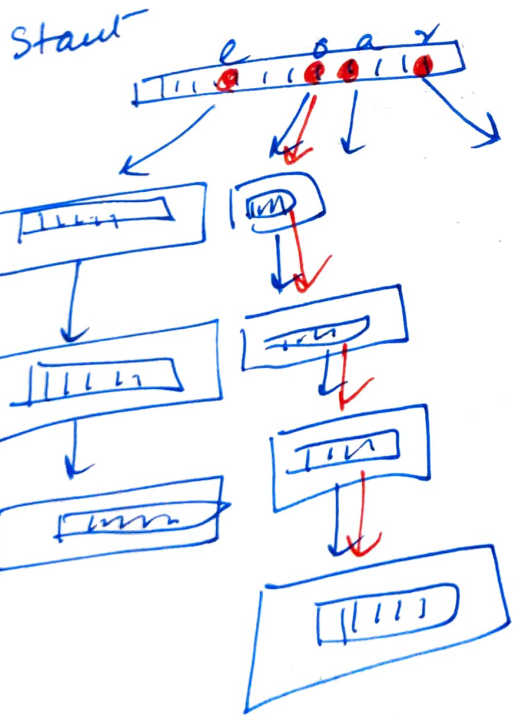
["oath", "pea", "eat", "rain"]

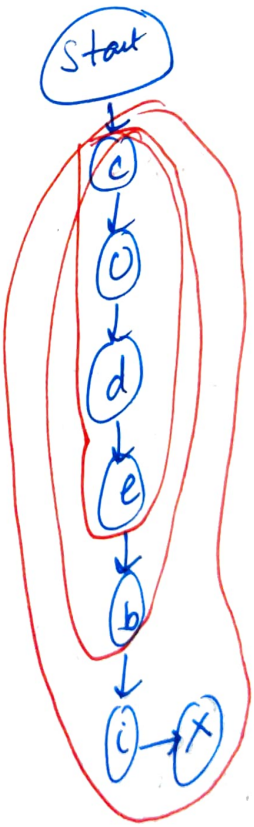




Why trie ?

["code", "codeb", "codebr", "codebia"]





All in one Heap LL TC

Word Ladder

Begin → 'hit'

End → 'cog'

wordList → ["hot", "dot", "dog", "lot", "log", "cog"]

Check if you can go from Begin to End word changing only 1 character of the word at a time.

hit → hot → dot → dog → cog

(5) → min steps return

⇒ BFS

Visited Hashmap

String	Boolean
"hit"	~~false~~ true
"hot"	false
⋮	
"lot"	false

Queue

hit

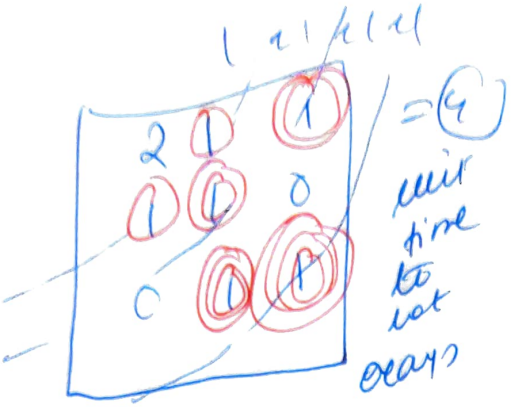
hit

abc.... z

drawn & check if changing one letter can yield a word present & unvisited in hashmap

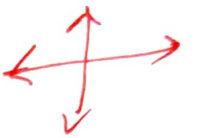
→ Terminate (level) → Return it finally

Rotten Orange

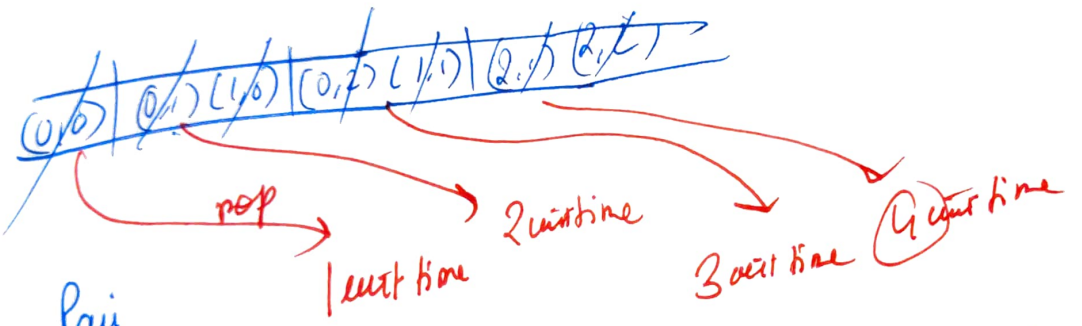


2 → Rotten
1 → fresh
0 → No Orange

- Rotten can change the surrounding fresh Oranges



- BFS



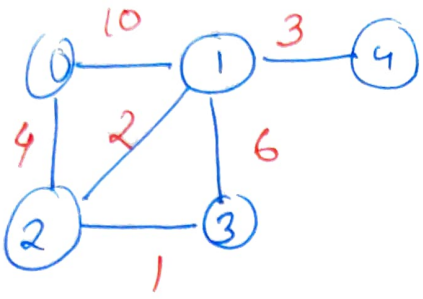
Pair

(0,0)

Rotten orange

- if more than 1 oranges already rotted, then add them together initially in queue.
- if fresh orange cannot be rotten ever, then return -1.

Prim's



- 4 edges are used
- 2 are cost more.

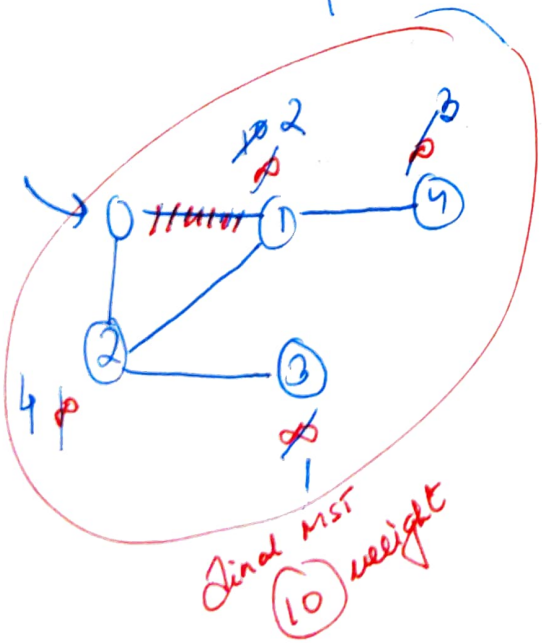
Vertex	Parent []	Weight []	Visited []
0	-1	0	✓
1	~~-1~~ 2	~~∞~~ 10 ~~2~~	✓
2	0	~~∞~~ 4	✓
3	2	~~∞~~ 1	✓
4	1	~~∞~~ 3	✓

Adj Mat $O(V^2)$
Adj list $O(E \log V)$

-1 -1 -1 -1 -1 Parent

0 ∞ ∞ ∞ ∞ Weight

F F F F F visited



Find MST weight (10)

Code in Github