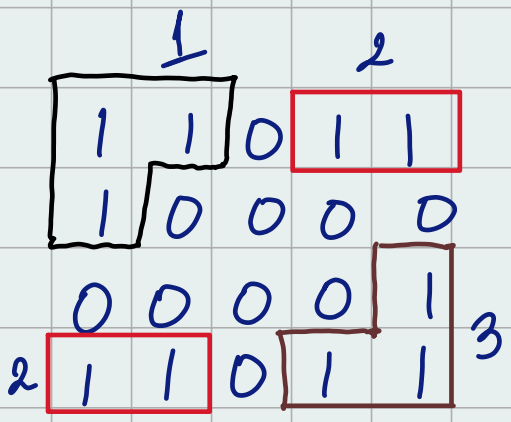
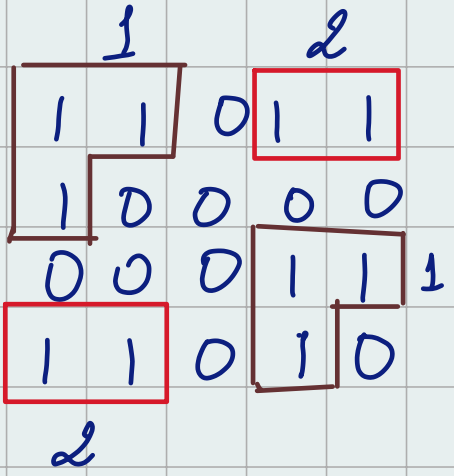
81) Number of distinct islands



Here there are 3 unique distinct islands are there



In this grid there are 2 unique distinct islands are present

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

{(0,0) (0,1) (1,0)}

(0,0) - (0,0) [(0,1) - (0,0)]

[(1,0) - (0,0)]

consider this as the base

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

(1,0) 1 → consider as a base

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

(3,3) 1

Similarly; {(0,0), (0,1), (1,0)}

[(2,3)-(2,3)] [(2,4)-(2,3)] [(3,3)-(2,3)]

Base (0,3) | (0,4) |

Base (3,0) | (3,1) |

{(0,0), (0,1)}

[(0,3)-(0,3)] [(0,4)-(0,3)]

{(0,0), (0,1)}

[(3,1)-(3,0)]

[(3,0)-(3,0)]

{(0,0), (0,1), (1,0)}

{(0,0), (0,1), (1,0)}

{(0,0), (0,1)}

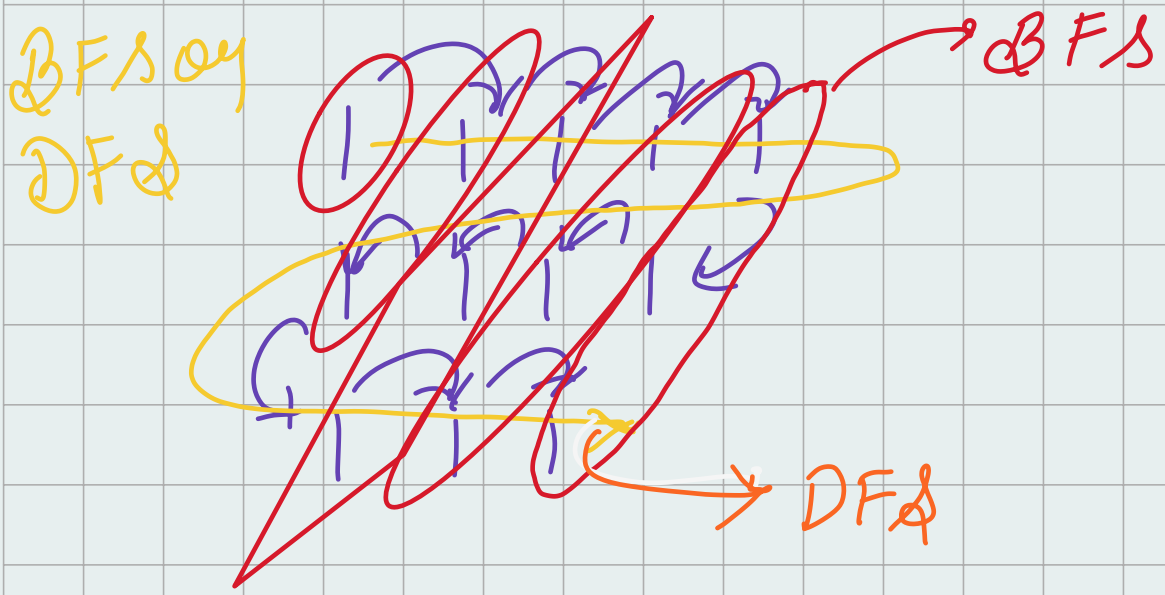
{0,0), (0,1)}

* Subtract the Base to store coordinates.

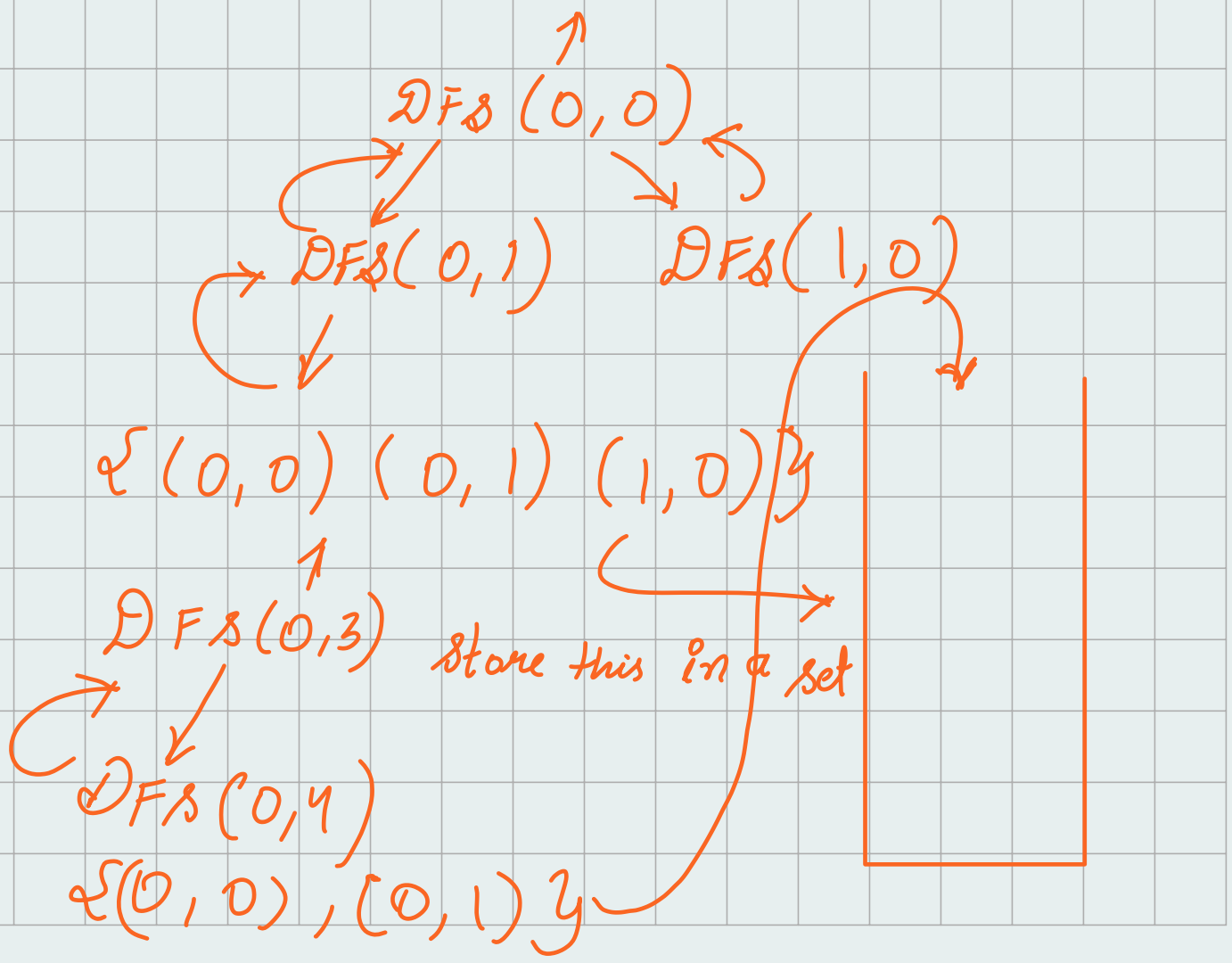
So apparently if we store them in a set data structure. Only 2 of them will get stored and that is our ans.

* what if the islands are almost completely filled with 1's than follow a particular

order to store the list. List ordering should stay same for everyone



→ Now we just need to store the islands for that we will use DFS.



T.C = n x m x (log(set.size()) + n x m x 4

Code :-

```
public int numDistinctIslands(int[][] grid)

    int n = grid length;

    int m = grid[0].length;

    int[][] vis = new int[n][m];

    HashSet<ArrayList<String>> set = new
                        HashSet<>();

    for (int i=0; i<n; i++){

        for(int j=0; j<m; j++){

            if (vis[i][j] == 0 && grid
                [i][j] == 1){

                ArrayList<String> vec = new
                            ArrayList<>();
//Vector is used here to store the path of
  the Island.

//  n x m x 4
```

```
dfs(i, j, vis, vect, grid,
        i, j );

// Here i, j are the base coordinates

                }
            }
        }
    // n x m x log(set.size()) + n x m x 4
    return set.size();
}

private void dfs(int row, int col,
int[][] vis, int[][] grid,
ArrayList<String> vec, int row0,
int col0){

    vis[row][col] = 1;

    int n = grid.length;

    int m = grid[0].length;
```

// Now, carry the base coordinates, to subtract the base coordinates from current co-ordinates.

```
vec.add((row - rowO) + " " +
        (col - colO));
```

// The string (" ") is used here to convert the integer to string

```
int[] delRow = {-1, 0, +1, 0};
int[] delCol = { 0, 1, 0, -1};

for(int i=0 ; i<4; i++){
    int nRow = row + delRow[i];
    int nCol = col + delCol[i];
    if (nRow >=0 && nRow<n &&
        nCol >=0 && nCol<m &&
        vis[nRow][nCol] == 0 &&
        grid[nRow][nCol] == 1){
```

dfs(nRow, nCol, vis, grid, vec, row 0, col 0);

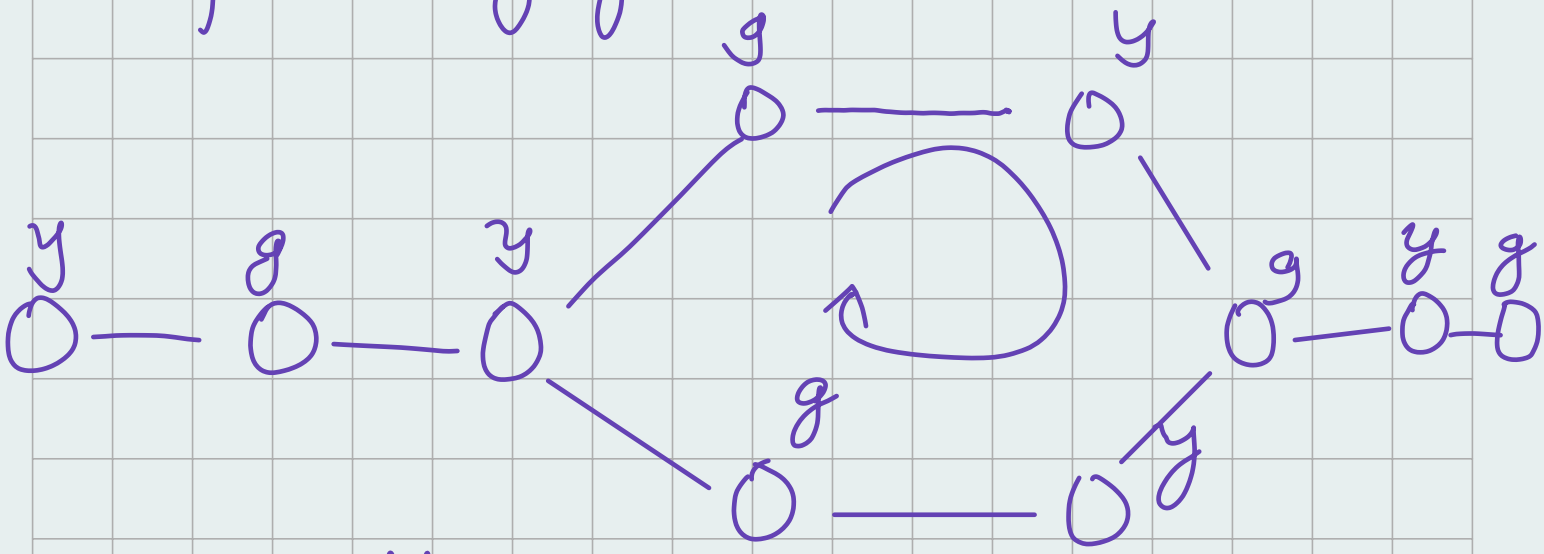
}

}

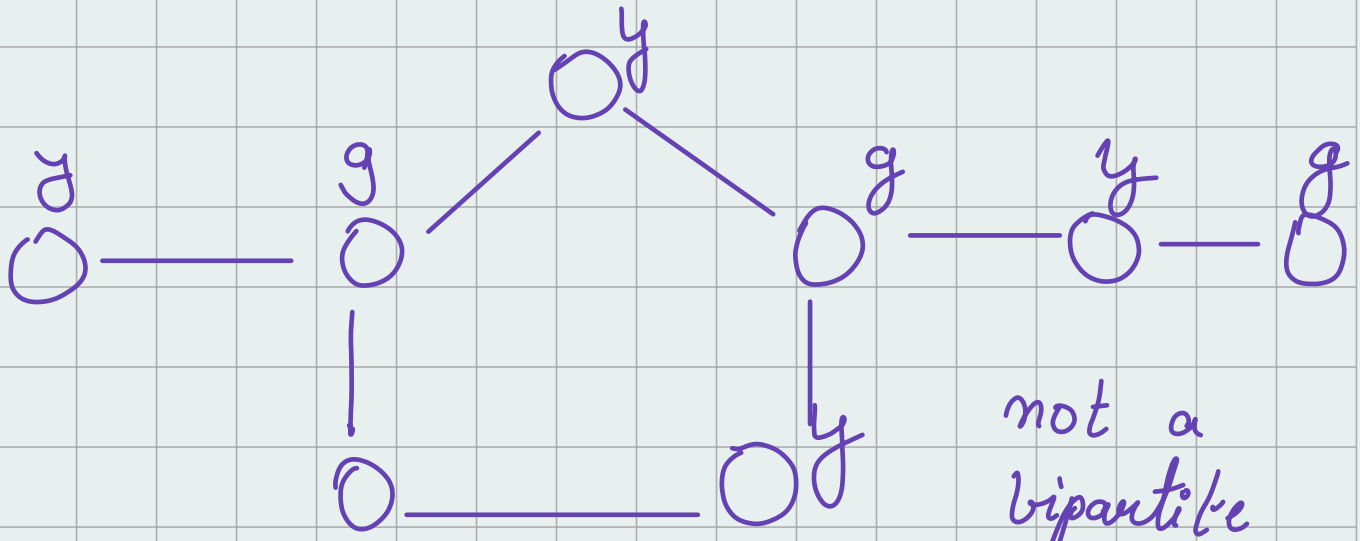
}

}

* Bipartite graph.



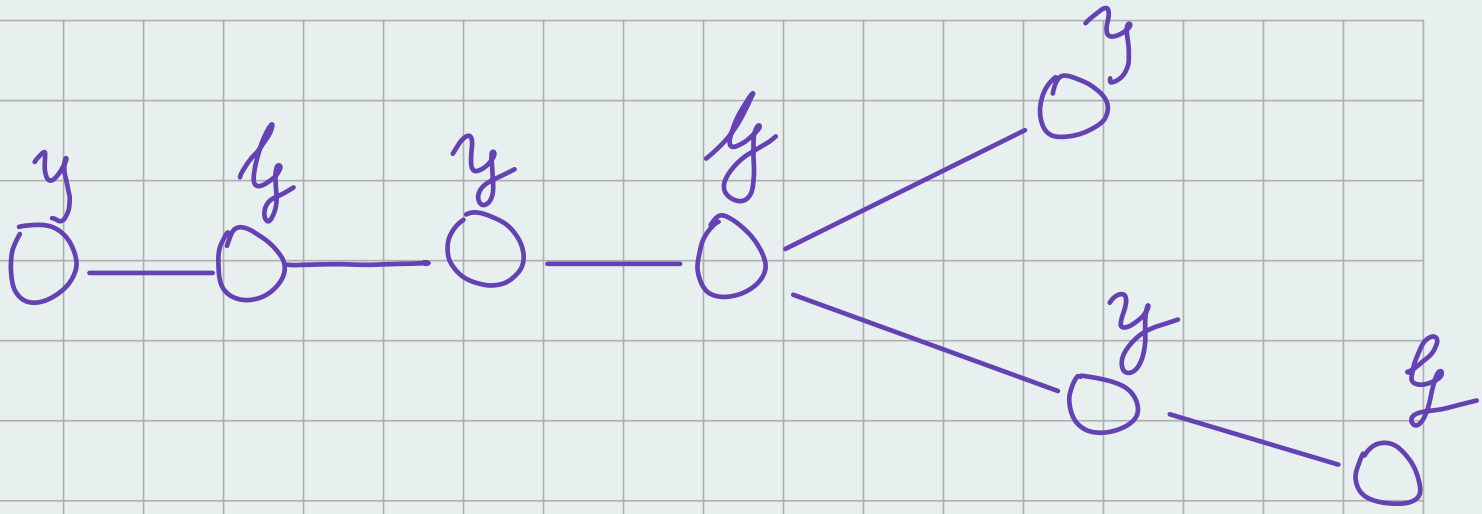
y = yellow g = green.



not a bipartite graph

we cannot color any of the 2 colors

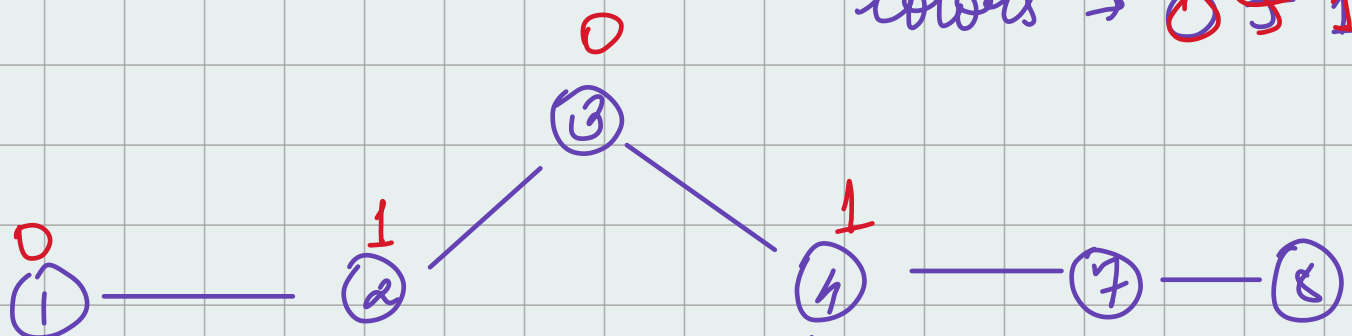
* linear graph with no cycles are always bipartite



* Any graph with even cycle length can also be bipartite.

* Any graph with odd length cycle can never be bipartite.

colors → 0 ~~1~~ 1



1 → {2}

2 → {1, 3, 6}

3 → {2, 4}

4 → {3, 5, 7}

~~node = 1~~

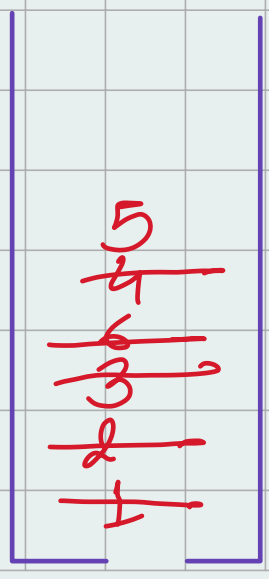
~~node = 2~~

~~node = 3~~

~~node = 6~~

node = 4

R



F

Queue

5 → { 4, 6 }

6 → { 2, 5 }

7 → { 4, 8 }

8 → { 7 }

node = 4 → Go back not a bipartite graph.

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

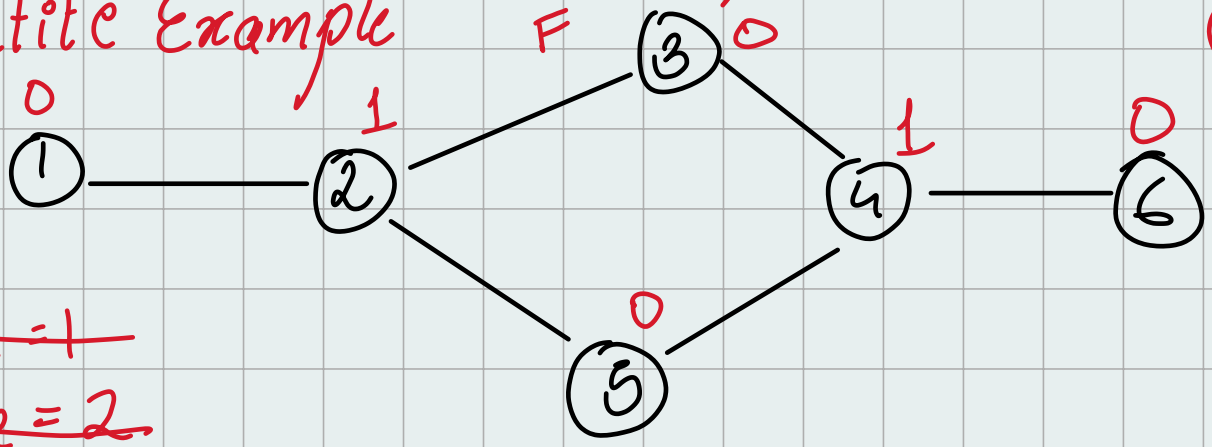
color

Queue

1 2 3 5 4 6

F R

Bipartite Example



~~node = 1~~

~~node = 2~~

~~node = 3~~

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

color

~~node = 5~~

~~node = 4~~

~~node = 6~~

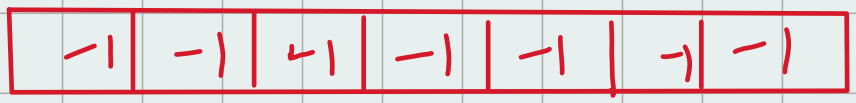
Code:

Class Solution {

```
private boolean check (int start, int V,
  ArrayList < ArrayList <Integer >> adj, int
[] color) {

    Queue <Integer > q = new LinkedList<>;

    q.add(start);

    color [start] = 0;

    while (!q.isEmpty()) {

        int node = q.peek();

        q.poll(),

        for (int adjacentNode ! adj.get(node)) {

            if (color [adjacentNode ] = -1) {

                color [adjacentNode] = 1 -
                                    color [node],

                q.add (adjacentNode),

            }

            else if (color [adjacentNode] ==
                                    color [node]) {
```

```
                return false;
            }
        }
    }
    return true;
}
```

```
public boolean isBipartite(int V,
    ArrayList<ArrayList<Integer>> adj){
    int[] color = new int[V];
    for(int i=0; i<V; i++){
        color[i] = -1;
    }
```




```
    for(int i=0; i<V; i++){
        if(!check(i, V, adj, color)){
```

return false;

}

}

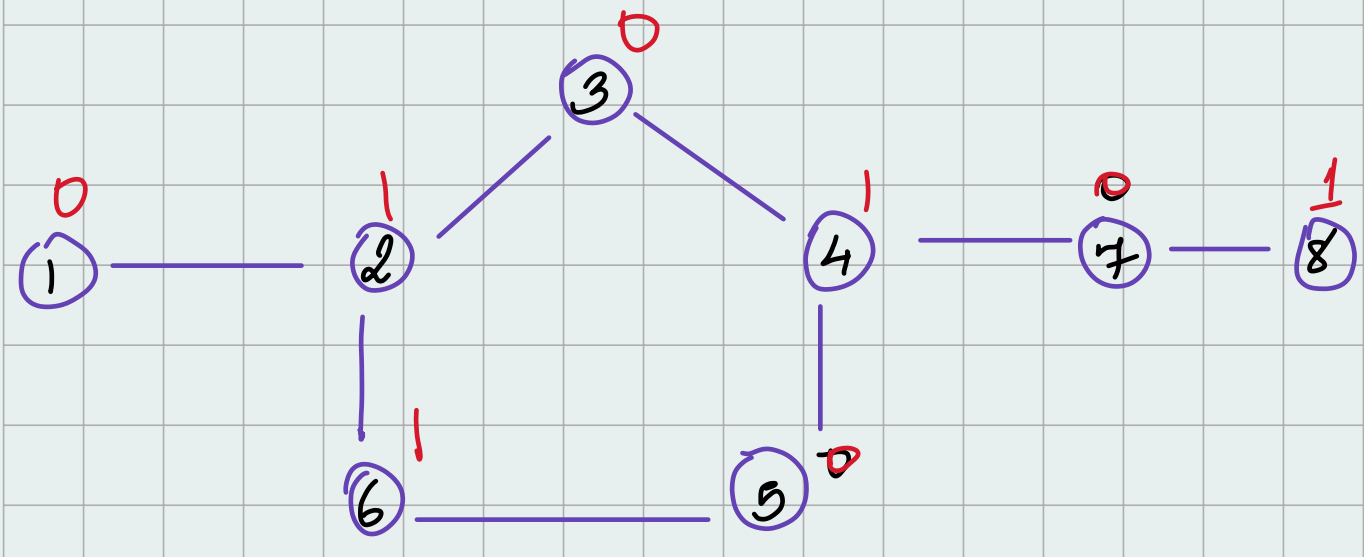
}

return true;

}

}

* Bipartite graph using DFS :-



Now make an adjacency list

1 → {2}

5 → {4, 6}

2 → {1, 3, 6}

3 → {2, 4}

4 → {3, 5, 7}

6 → {2, 5}

7 → {4, 8}

8 → {7}

start = 1

color[ ]

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

dfs(1, 0) → false

dfs(2, 1) → false

dfs(3, 0) → false

dfs(4, 1) → false

dfs(5, 0) → false

dfs(6, 1)

dfs(7, 0) and dfs(8, 1) → true

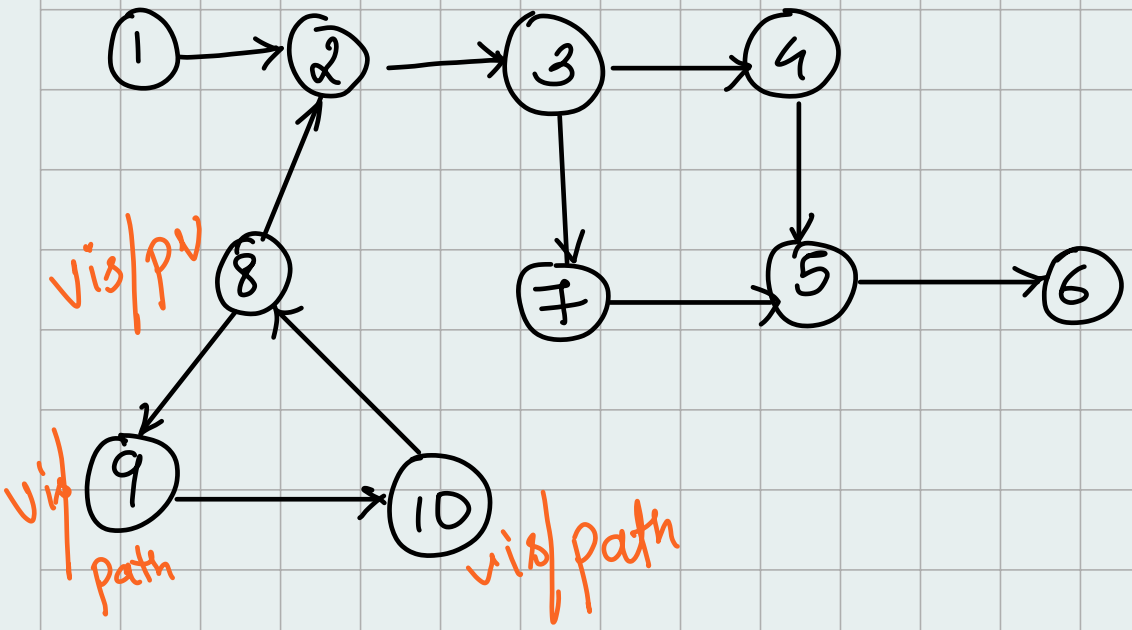
Time Complexity : O(V + 2E)

Code :-

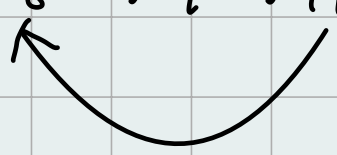
```
public boolean isBipartite(int V, ArrayList<
  ArrayList<Integer>> adj){
    int[] color = new int[V];
    for(int i = 0; i<V; i++){
        color[i] = -1;
    }
    for(int i = 0; i<V; i++){
        if(color[i] == -1){
            if(!dfs(i, 0, adj, color)){
                return false;
            }
        }
    }
    return true;
}
private boolean dfs(int node, int col,
  ArrayList<ArrayList<Integer>> adj, int[] color){
```

```
color[node] = col;

for(int adjacentNode : adj.get(node)){
    // Check for self loops

    if(color[adjacentNode] == col){

        return false;
    }
    if (color[adjacentNode]==-1){

        if(!dfs(adjacentNode,
            1- col, adj, color)){
            return false;
        }
        else if(color[adjacentNode]=col){

            return false;

        }

    }

}

return true;

}
```

* Detect cycle in a Directed graph (DFS)



* On the same path node has to be visited again then we can say it's a cycle. For ex: 8 → 9 → 10



1 → {2}

2 → {3}

3 → {4, 7}

4 → {5}

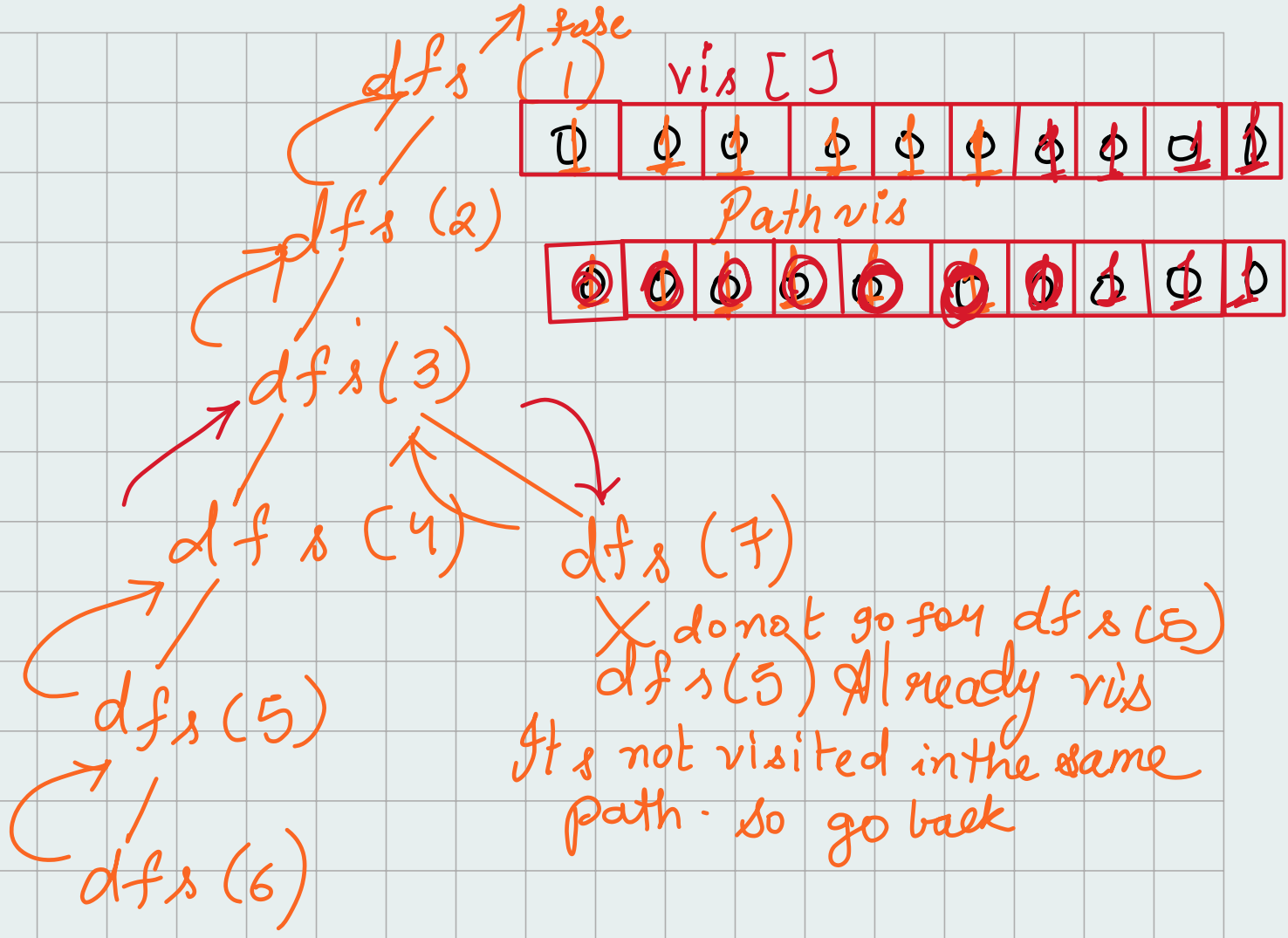
6 → { }

7 → {5}

8 → {2, 9}

9 → {10}

10 → {8}



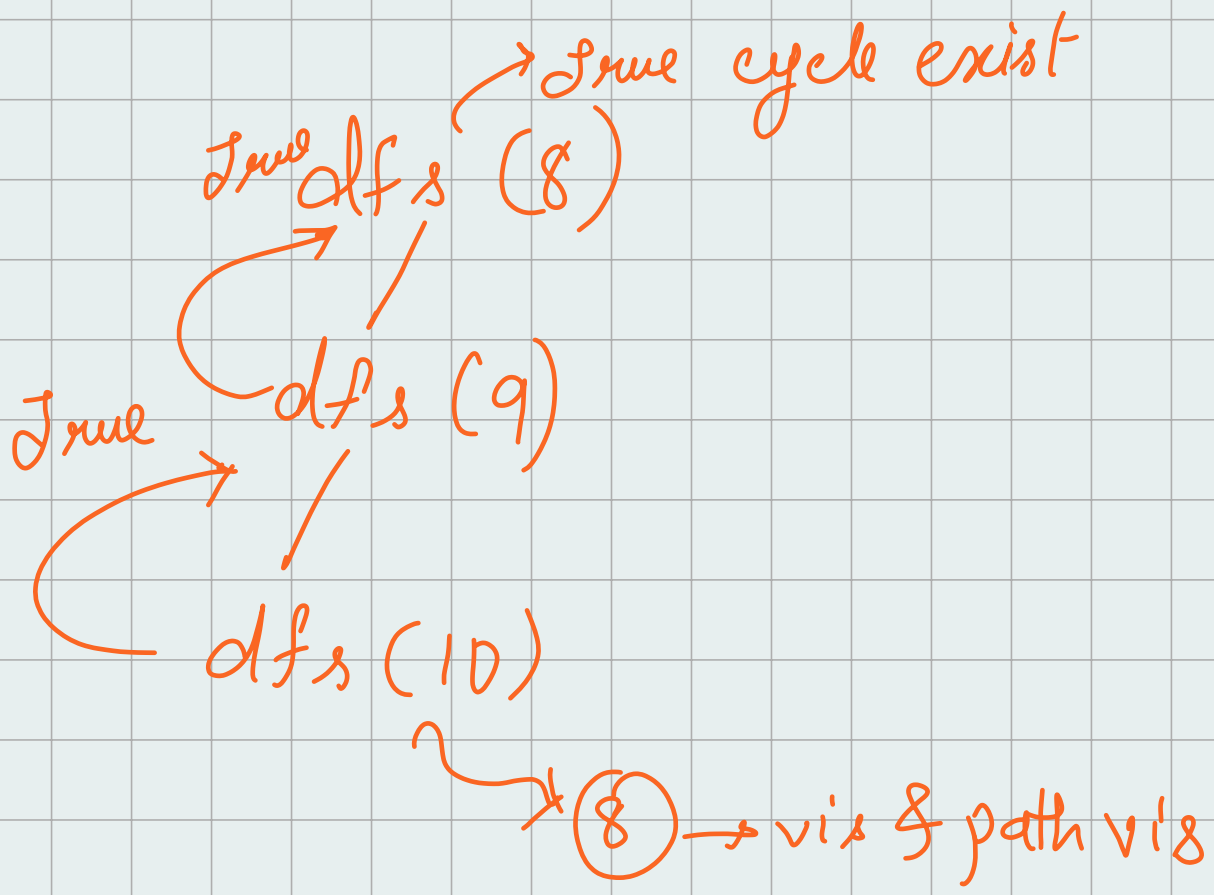
↳ when we go back from 6 we keep the vis marked as 1 but omit the path visited index [6] = 0
now go back to 5 omit the path visited to 0
go back to 4 omit the path visited to 0.
3 will not be omitted because 7 is still remaining

After going back from calls if every one in the path vis will mark themselves as zero.

```
for (i = 1 2 3 ... 8 → v){
    if (! vis[v]){
        if (dfs(i) == true)
```

return true;

}



Code :-

```
class Solution {

    public boolean isCyclic(int V, ArrayList
    <ArrayList<Integer>> adj){

        int[] visited = new int[V];
        // It is initialized with 0 in the vis[]

        int[] pathVis = new int[V];
        // It is initialized with 0.
```

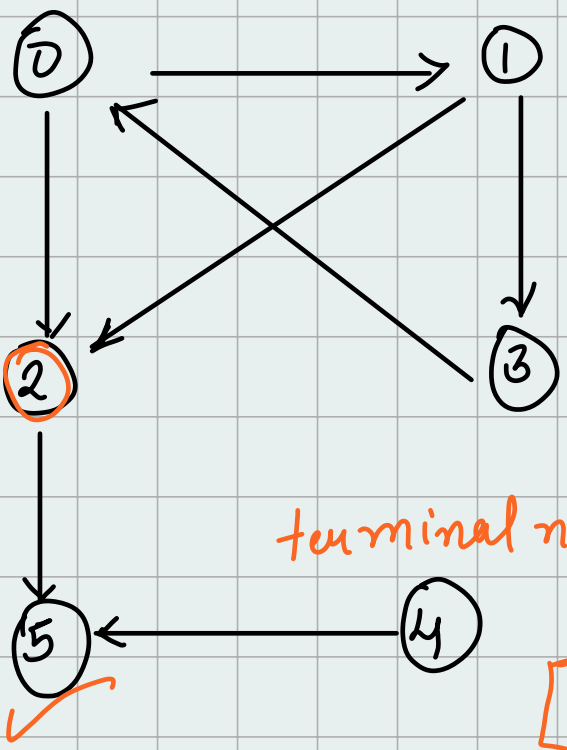
```
for( int i=0 , i<v; i++){
    if (visited[i] ==0){
        if (dfsCheck( i, adj, visited,
                      pathVis ) ){
            return true;
        }
    }
}
return false;
}

private boolean dfsCheck (int node,
ArrayList < ArrayList <Integer> adj
int [] visited, int [] pathVis){

    visited [node] = 1; if it is visited then
                        mark it as 1
    pathVis [node ] = 1 ;

    // traverse for adjacent nodes.

    for (int adjNode : adj.get(node)){
```

```
// if the node has a self loop then it
   is a cycle

if (adjNode == node){

    return true;
}

// check for immediate back edge
  (not including the current node).

if (visited[adjNode] == 1 &&
    pathVis[adjNode] == 1 &&
      adjNode != node){

    return true;
}

//when the node is not visited

if (visited[adjNode] == 0){

    if (dfsCheck(adjNode, adj,
          visited, pathVis)){

        return true;
```

```
        }
        // if the node has been previously
           visited
        // But it has to be visited on the
           same path
        else if (pathVis[adjNode] == 1){
            return true;
        }
    }
}
// remove the marked as 1 from the
   current path Vis array
pathVis[node] = 0;
return false;
}
}
```

Time Complexity :~ $O(V+E)$

9) Find Eventual Safe states



Safe nodes → ?

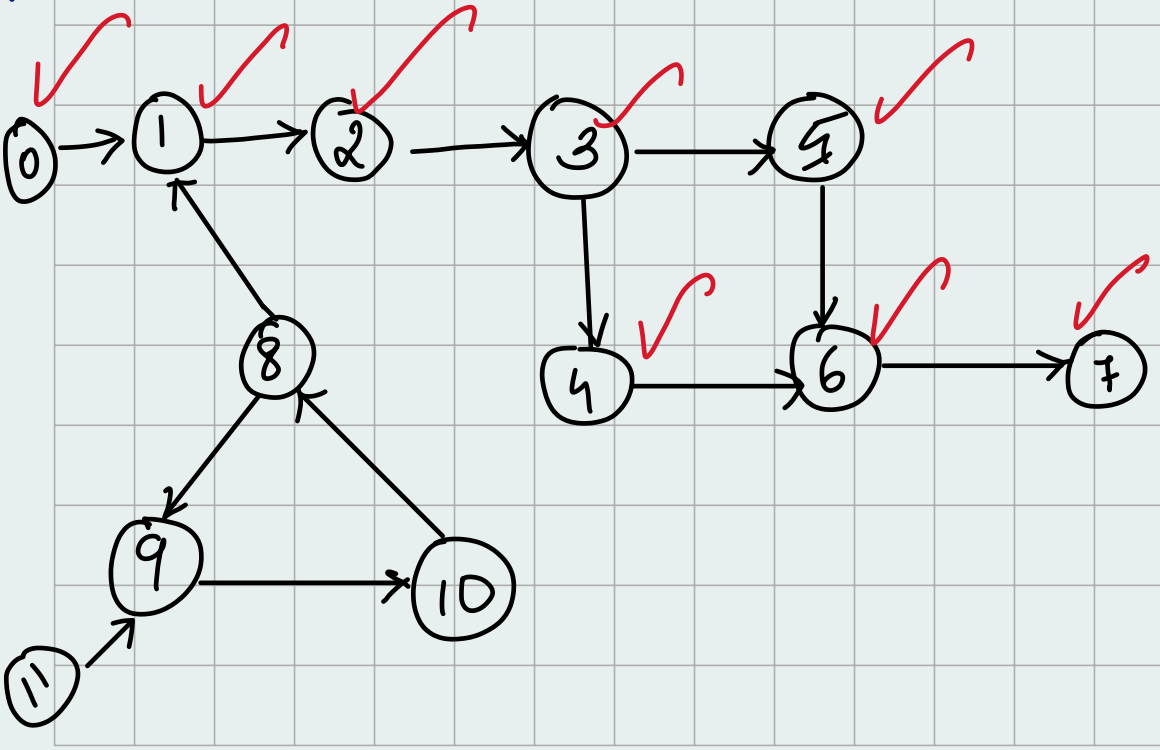
terminal nodes

↳ outdegree → 0

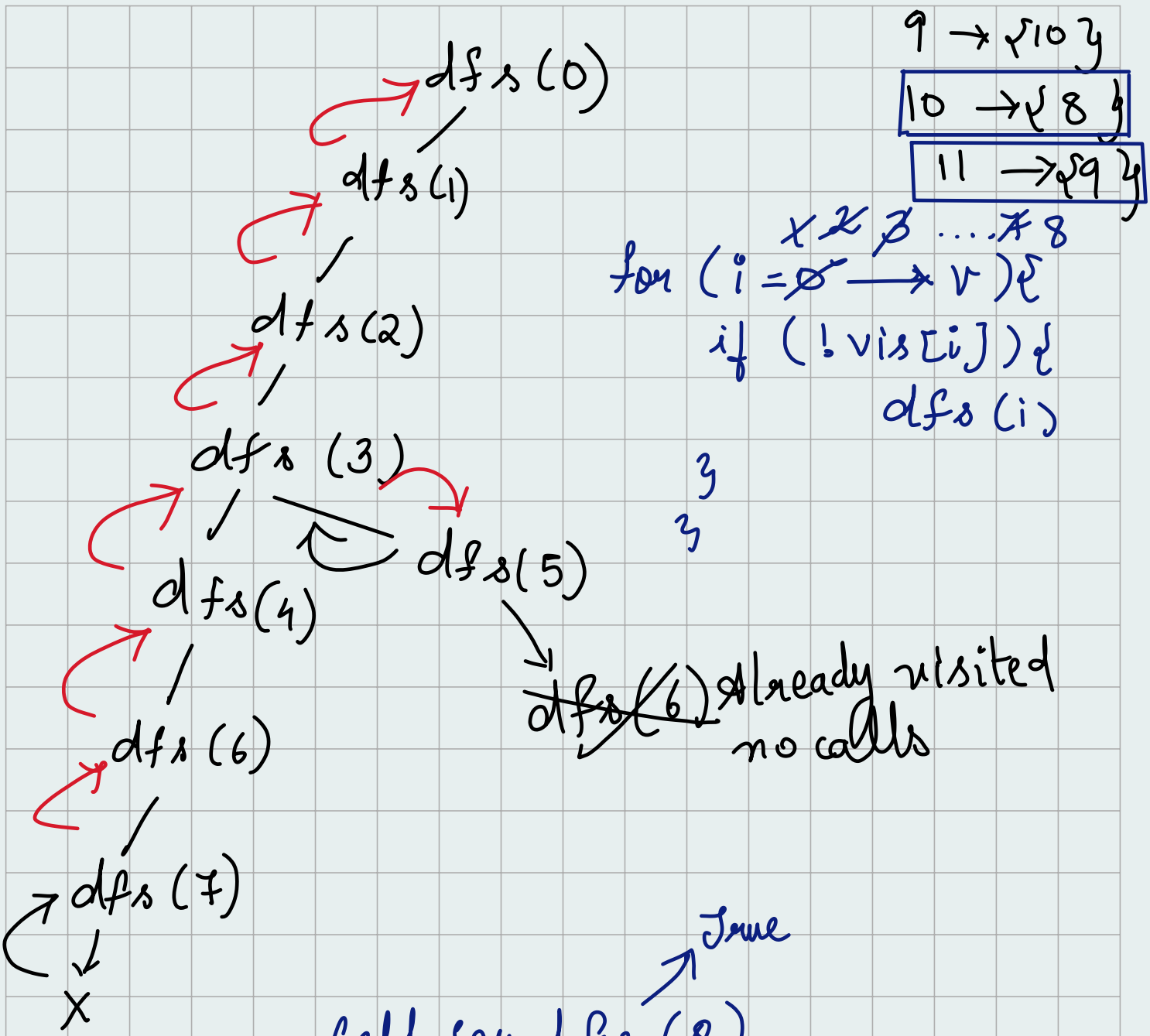
6

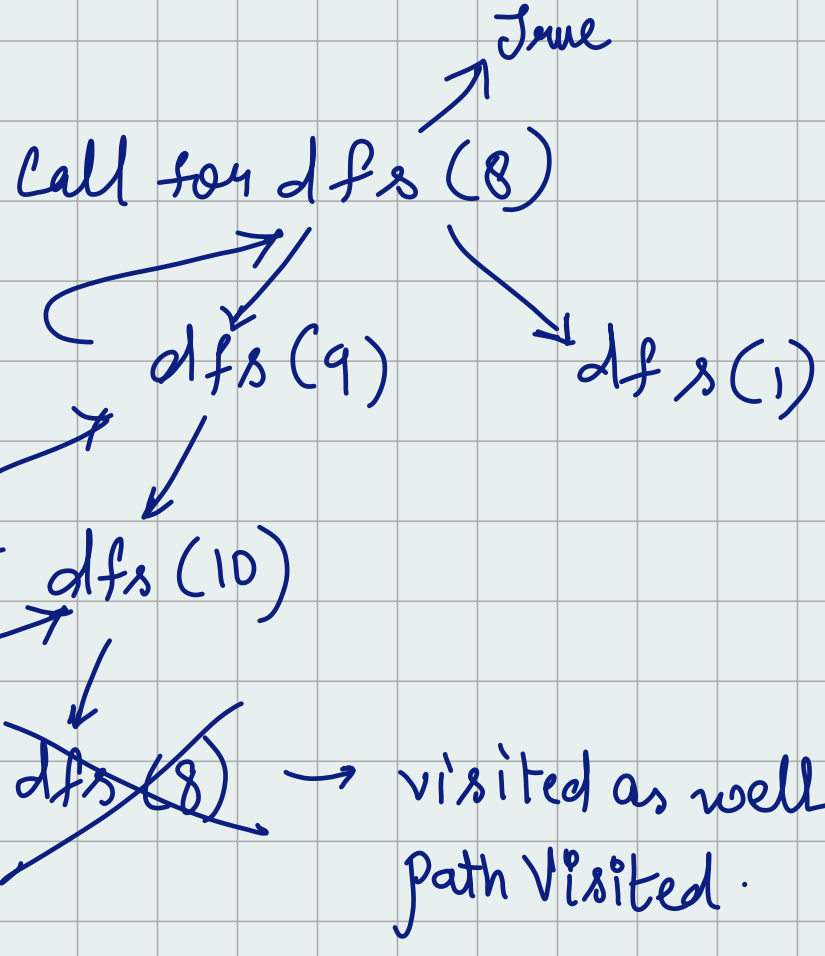
terminal nodes: arr[2 4 5 6]

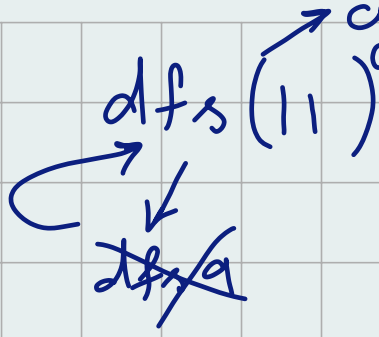
[0, 1, &3 cannot be Safe nodes as there a cycle exist]



0 → {1}
1 → {2}
2 → {3}
3 → {4, 5}
4 → {6}
5 → {6}
6 → {7}
7 → { }
8 → {1, 9}


dfs(0)
dfs(1)
dfs(2)
dfs(3)
dfs(5)
dfs(4)
dfs(6)
dfs(7)
X
dfs(6) already visited
no calls
9 → {10}
10 → {8}
11 → {9}
for (i = 0 → V) {
if (!vis[i]) {
dfs(i)
}
}


True
Call for dfs(8)
dfs(9)
dfs(1)
dfs(10)
True
cycle
exist
dfs(8)
visited as well as
path visited.



Code :-

```
class Solution{

    public List<Integer> eventualSafeNodes( int V,
        List<List<Integer>> adj ){

        int[] vis = new int[V];

        int[] pathVis = new int[V];

        int[] check = new int[V];

        List<Integer> safeNodes = new ArrayList<>();

    for(int i=0 ; i<V ; i++){

            if (vis[i] == 0){
                dfsCheck (i, adj, vis, pathVis, check);
            } // check for all the nodes in the graph
    }   // We want all the nodes that are safe
```

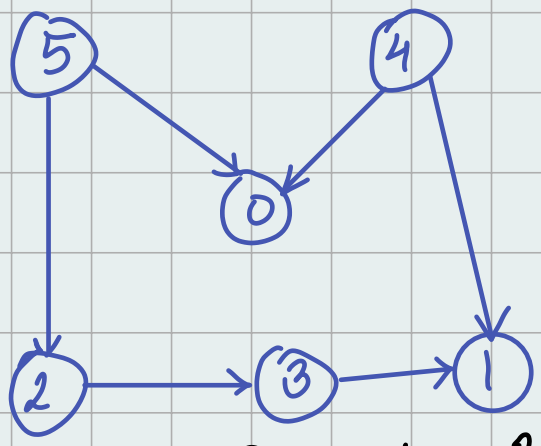
```
for (int i=0 ; i < V ; i++){
    if (check[i] == 1){
        safeNodes.add(i);
    }
}
return safeNodes(i);
}
```

This will check in ascending order and store them in a list

```
private boolean dfsCheck (int node, List<
    List<Integer>> adj, int[] vis, int[] pathVis,
    int[] check){

    vis[node] = 1;
    pathVis[node] = 1;
    check[node] = 0; // Initially mark all
                        the safe nodes as 0

    // traverse for adjacent nodes.

    for(int adjNodes : adj.get(node)){

        if (adjacentNode == node){
            check[node] = 0;
            return true;
        }
```

```
if(vis[adjNode] == 1 && pathVis[adjNode] == 1){
        check[node] = 0;
        return true;
    }

if(vis[adjNode] == 0){
    if(dfsCheck(adjNode, adj, vis, pathVis,
                check)){
// dfs Check for adjNode, if it is a
   cycle then return true.

        check[node] = 0;
        return true;
    }
}
}

check[node] = 1; // if the node is safe
                    mark it as 1.

pathVis[node] = 0; // mark 0 to all
                      the position in pathVis
                      arrays.

return false;
}
}
```

* Topological sort :- Directed Acyclic graph



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

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

```
for (i = 0 → v)
    if (!vis[i]){
        dfs(i)
    }
}
```

dfs(0)
dfs(1)
dfs(2)
dfs(3)
~~dfs(4)~~ vis[i]
dfs(4)
dfs(5)

stack: 5, 4, 2, 3, 1, 0

ans = { 5 4 2 3 1 0 }

→ one of the linear orderings

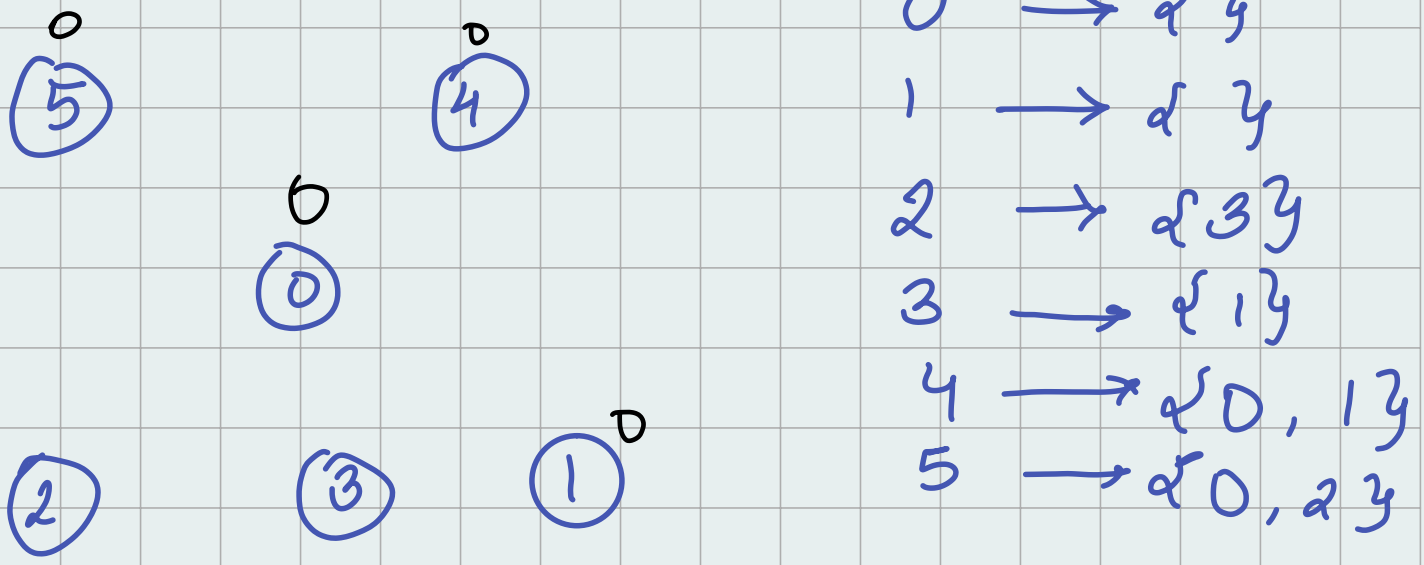
Code :-

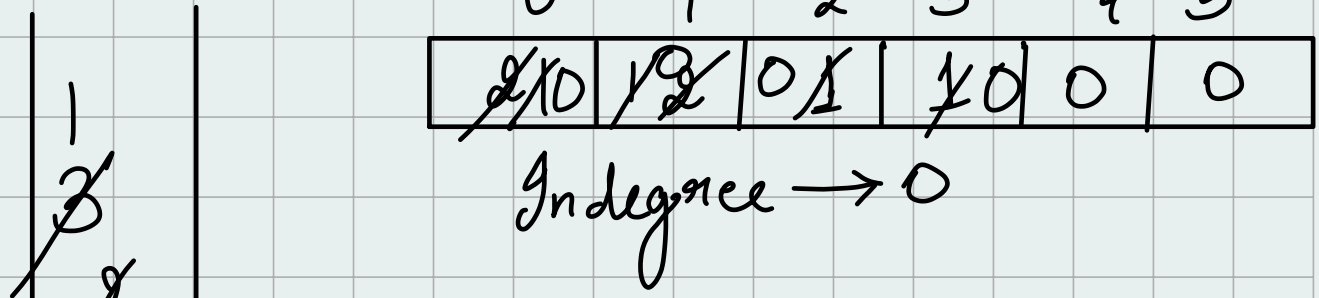
```
public static int[] topoSort(int V, ArrayList<
  ArrayList<Integer>> adj ){

      int[] vis = new int[V];

      Stack<Integer> stack = new Stack<>();

      for( i = 0 ; i < V ; i++){

            if (vis[i] == 0){

                  topodfs( i, V, vis, adj );

            }

      }

      int[] ans = new int[V]

      int i = 0 ;

      while (! stack.isEmpty ()){

            ans[i++] = stack.peek();
            stack pop ();
```

```
        }

        return ans;
    }
    private static void topodfs(int node,
ArrayList<ArrayList<Integer>> adj, int[] vis,
 stack<Integer> stack){

        vis[node] = 1;

        for(int adjNode : adj.get(node)){

            if(vis[i] == 0){

                topodfs(adjNode, adj, vis, stack);

            }
        }

        stack.push(node);

    }
}
```

* Topological Sorting (Kahn's Algorithm / BFS)

→ Linear ordering of vertices such that if there is an edge between u & v, u appears before v in the ordering





Queue

i) Insert all nodes in queue with indegree 0.

ii) Take them out of the Queue and just reduce the indegree of adjacent nodes. If the indegree is 0 then go to the Queue and print out

Topo Sort → {4, 5, 0, 2, 3, 1}

Code :-

```
public class KahnAlgo{

    public static int[] toposort(int V,
ArrayList<ArrayList<Integer>> adj){

        int[] indegree = new int[V];

    for(int i = 0; i<V; i++){

        for (int adjnode : adj.get(node)){

            indegree[adjnode]++;

    }

}

    Queue<Integer> queue = new LinkedList<>();

    for ( int i = 0 ; i < V ; i++){

        if (indegree[i] == 0){

            queue.add[i];

        }
```

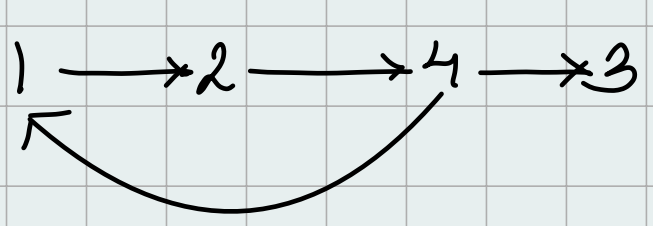
```
}
int[] topo = new int[V];
int i = 0,
while (! queue.isEmpty()){
    int node = queue.peek<>();
    queue.remove();
    topo[i++] = node;
    //node is in your toposort, now
     remove all the edges.
    for (int adjNode : adj.get(node)){
        indegree[adjNode]--;
        if (indegree[adjNode]==0){
            queue.add(adjNode);
        }
    }
}
```

```
    }
    return topo;
  }
}
```

* Course Schedule I and II

→ If the task can be performed with a yes or no

1	2
4	3
2	4
4	1



✓ ✓ ✓ X No, cause there forms a cycle so task cannot be performed.

Topo Sort → Yes (DAG)
(Directed graph)
(Acyclic graph)

u → v

possible → yes
→ No

Code:-

```
class Solution{

    public boolean isPossible(int V,
      int[][] prerequisites ){

        //Form a graph

        ArrayList<ArrayList<Integer>> adj = new
                                ArrayList<>();

        for(int i=0; i<V; i++){
            adj.add(new ArrayList<>());
        }
        int m = prerequisites.length;

        for(int i=0; i<m; i++){
            adj.get(prerequisites[i][0]).add
                                (prerequisites[i][1]);
        }
```

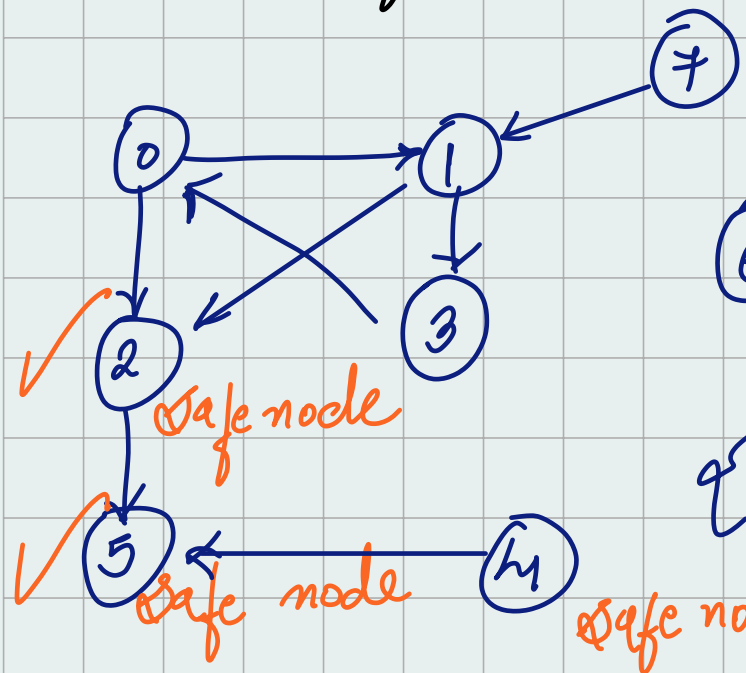
```
int[] indegree = new int[v];
for (int i=0; i<v; i++){
    for (int adjnodes : adj.get(i)){
        indegree[adjnodes]++;
    }
}

Queue<Integer> q = new LinkedList<Integer>();
for (int i = 0; i<v ; i++){
    if (indegree[i] == 0){
        q.add(i);
    }
}

List<Integer> topo = new ArrayList<>();

while (!q.isEmpty()){
    int node = q.peek();
```

```
q.remove();
topo.add(node);
for (int adjNodes : adj.get(node)){
    indegree[adjNodes]--;
    if (indegree[adjNodes]==0){
        q.add(adjNodes);
    }
}
if (topo.size()==v) return true;
return false
}
}
```

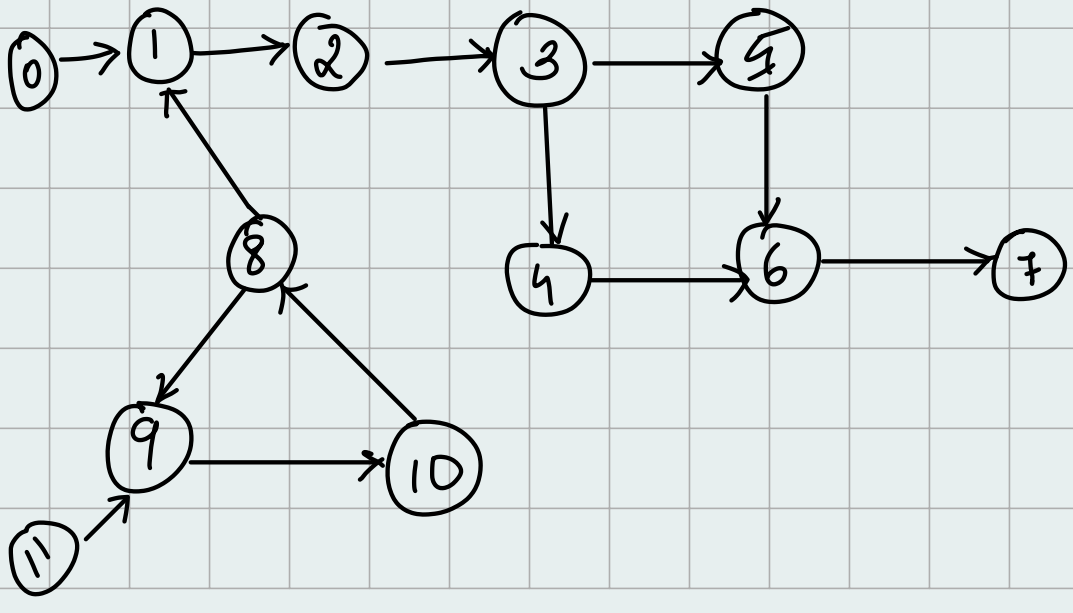
* Find Eventual Safe nodes.

* Safe nodes are all the nodes that end up at terminal node (without any cycle present in their path).

Safe nodes → terminal nodes.



{ terminal nodes are those node which doesn't have any outgoing edges like 5, 6 }



Safe Nodes

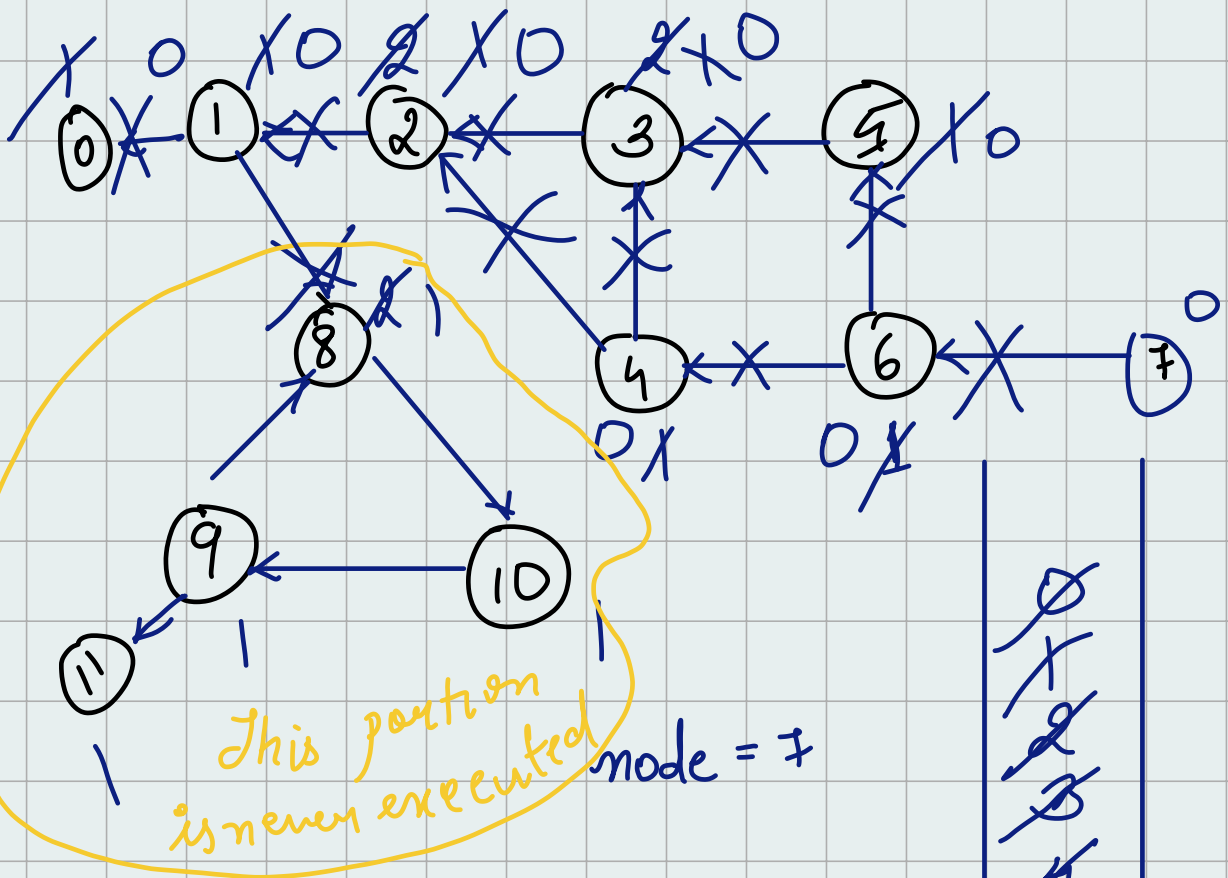
↓

terminal nodes.

→ reverse all the edges.

→ get all the nodes with indegree of 0.

→ do a removal of edges on adjacent Nodes.



Safe Node 2

7 6 5 4 3 2 1 0

Sort it

Code:-

```
class Solution {

    List<Integer> eventualSafeNodes(
            int V, List<List<Integer>> adj) {

        List<List<Integer>> adjRev = new
                                ArrayList<>();

        int[] inDegree = new int[V];

        for(int i = 0; i<V; i++){

            adjRev.add(new ArrayList<>());
        }

        for(int i=0; i<V; i++){

            for(int it: adj.get(i)){

                adjRev.get(it).add(i);

                inDegree[i]++;
            }
```

```
}

    Queue<Integer> queue = new LinkedList<>();
List<Integer> safeNodes = new ArrayList<>();

for (int i=0; i<V; i++){
        if (indegree[i] ==0){
            queue.add(i);
}}
while (!queue.isEmpty()){
 int node = queue.poll();

 safeNode.add(node);
  for( int adjNode : adjRev.get(node)){
      inDegree[adjNode]--;
      if (inDegree[adjNode] ==0){
          queue.add(adjNode);
      }
}
  Collections.sort(safeNodes);
  return safeNodes;
```

}
}

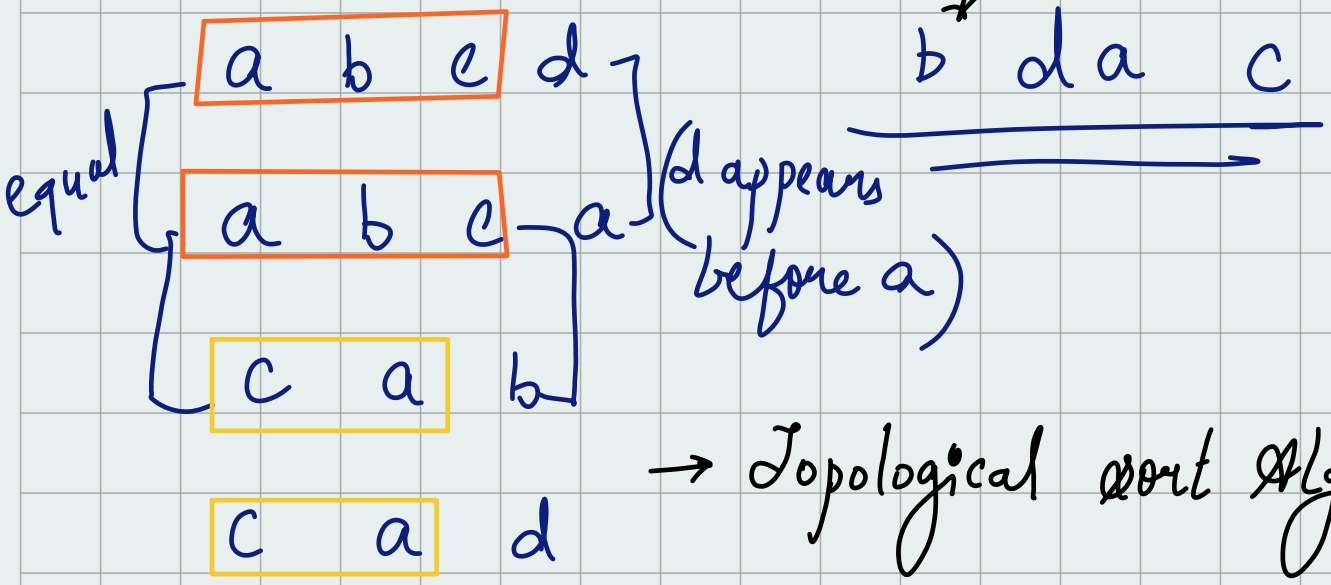
Q) Alien Dictionary

→ find out the alien order?

b appears before a (b a)

b a a

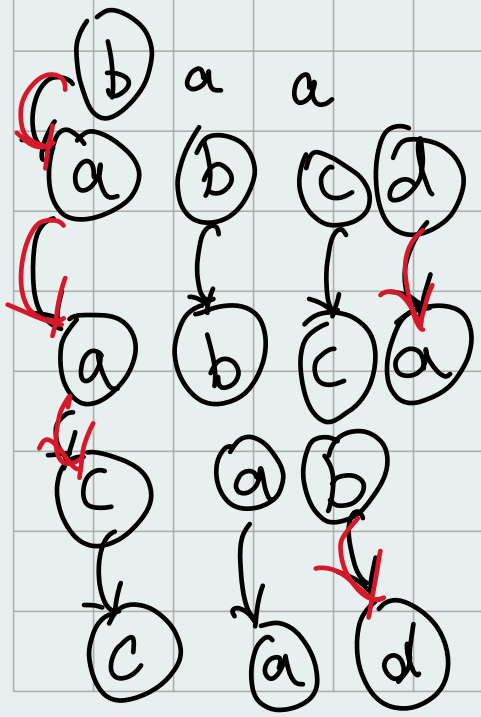
Ans ?

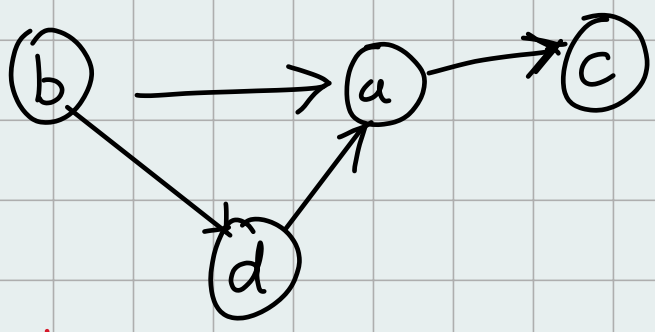


→ Topological sort Algorithm

u → v

u appears before v intuition





DG

b d a c → return

Suppose there are nodes which are not connected to any of the Directed Graph (DG) than it can be added in the return list in the start or in the end of the list ex: b d a c e or e b d a c

i =				
0	b	a	a	
1	a	b	c	d
2	a	b	c	a
3	c	a	b	
4	c	a	d	

i = 0

S_1 = arr[i] = b a a

S_2 = arr[i+1] = a b c d

The moment we find non equality we won't proceed further. Just add an edge between b to a

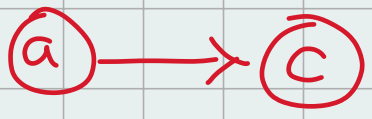
i = 1

S_1 = arr[i] = a b c d

S_2 = arr[i+1] = a b c a

i = 2 s_1 = arr[i] = a b c a

s_2 = arr[i+1] = c a b



i = 3 s_1 = arr[i] = c a b^x

s_2 = arr[i+1] = c a d

Code :-

```
public class AlienDictionary {

    public String findOrder(String[] dict,
        int N, int K) {

        List<List<Integer>> adj = new ArrayList<>();

        for(int i=0 , i< K; i++){
            adj.add(new ArrayList<>());
        }

        for(int i = 0; i< N-1; i++){
// N-1 because we are comparing till the second
  last with the last
```

```
String s1 = dict[i];
String s2 = dict[i+1];
for(int ptr=0; ptr < Math.min(
    s1.length(), s2.length()); ptr++){
    if(s1.charAt(ptr) != s2.charAt
                          (ptr)){
        adj.get(s1.charAt(ptr) - 'a').
        add(s2.charAt(ptr) - 'a');
        break;
    }
}
}
List<Integer> topo = toposort(k, adj);
String ans = "";
for(int it : topo){
```

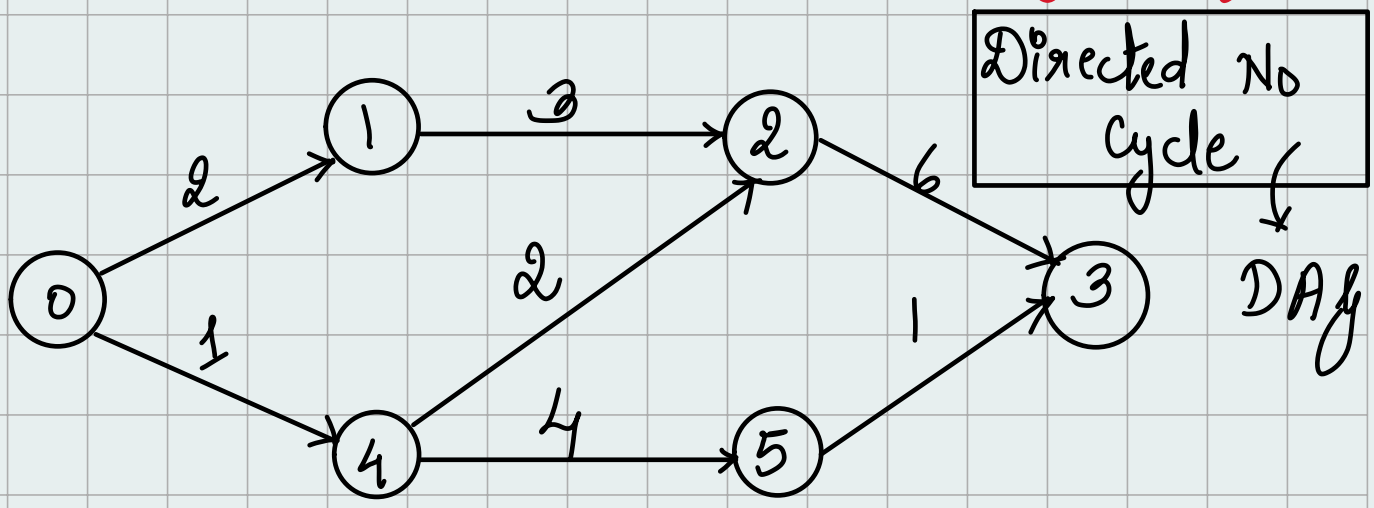
```
            ans = ans + (char)(it + (int)('a'));
        }
        return ans;
    }

    private List<Integer> topoSort(int V,
        List<List<Integer>> adj) {
        int[] inDegree = new int[V];
        for(int i = 0; i < V; i++) {
            for (int adjNode : adj.get(i)) {
                inDegree[adjNode]++;
            }
        }
        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < V; i++) {
            if (inDegree[i] == 0) {
```

```
            q.add(i);
        }
    }
    List<Integer> topo = new ArrayList<>();
    while(!q.isEmpty()){
        int node = q.poll();
        topo.add(node);
        for(int adjNode: adj.get(node)){
            inDegree[adjNode]--;
            if(inDegree[adjNode]==0){
                q.add(adjNode);
            }
        }
    }
}
```

return topo;

}

* Shortest path in Directed Acyclic graph

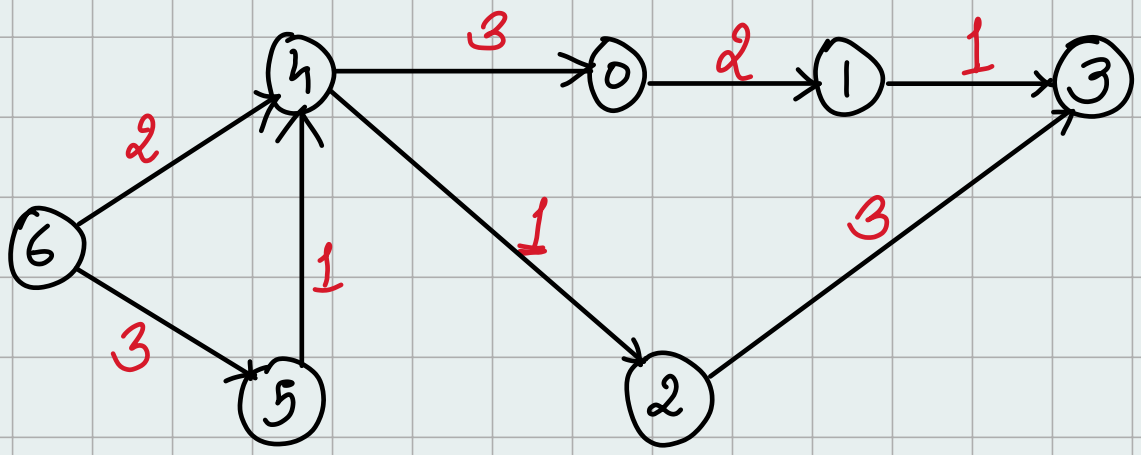


	0	1	2	3	4	5
dist →	0	2	3	6	1	5

src = 0 → 0, 1, 2, .. (n-1) node

↓ src to itself dist is 0

Shortest distance to travel:



* Make an adjacency List : pairs of nodes & list.

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

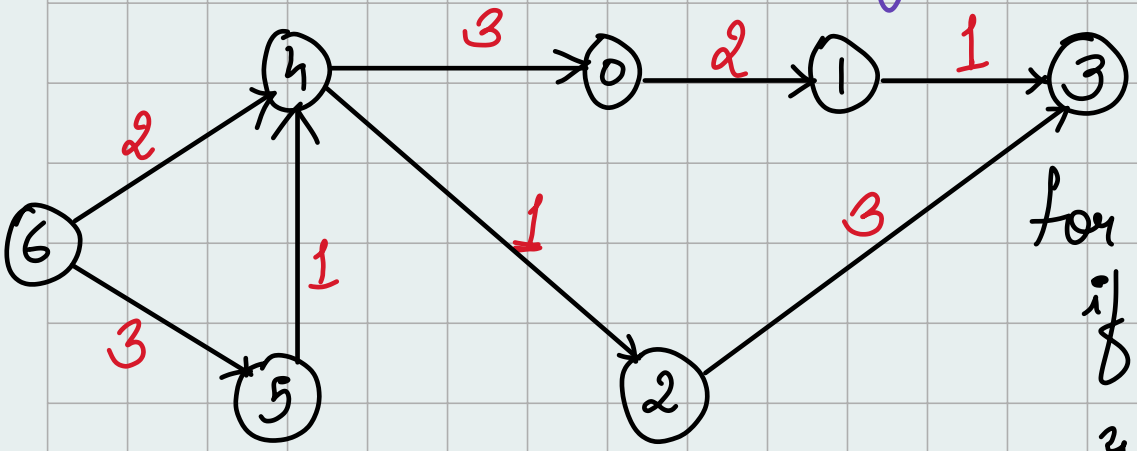
→ Algorith
→ Code
→ Intuition at last.

ArrayList < ArrayList < pairs > >

For this graph assume the src = 6.

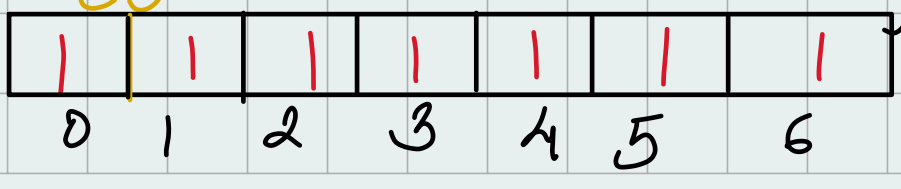
Step 1 = Do a TopoSort on this DAG. (DFS method)

Step 2 = Take the nodes out of the stack and relax the edges.




```
0 + 1 2 3 4
for (i = 0 → 6){
  if (!vis[i]){
    dfs(i)
  }
}
```

vis[ ]



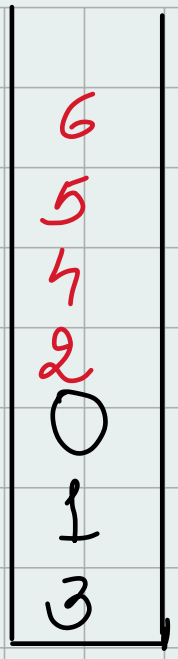
dfs(0)

dfs(1)

dfs(3)

callonen put it into the stack

similarly for all nodes



dfs(2)

X

dfs(4)

X

dfs(5)

callonen put it into the stack

dfs(6)

X

callonen put it into the stack

* Now this stack will contain the topological sorted nodes

Now step 2: In order to relax the edges we need to declare a distance array and it should be marked with infinity.

dist[]

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

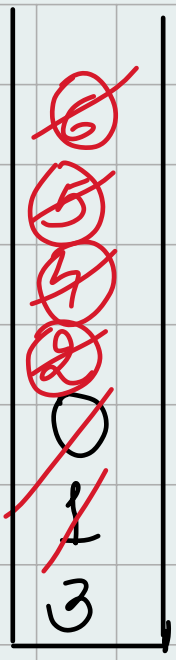
Src = 6 so make node 6 dist ∞ to 0

start from 6 end at 6

node=6, dist=0

+3 ↙ node=5, dist=3

+2 → node=4, dist=2



node = 5, dist = 3

+1 ↙

node = 4, dist = ~~1~~ already have a better dist 2 don't take omit it :)

node = 4, dist = 2

+3 ↙ node = 0, dist = 5

+1 ↘ node = 2, dist = 3

node = 2, dist = 3

+3 ↙

node = 3, dist = 6

node = 0, dist = 5

+2 ↙

node = 1, dist = 7

node = 1, dist = 7

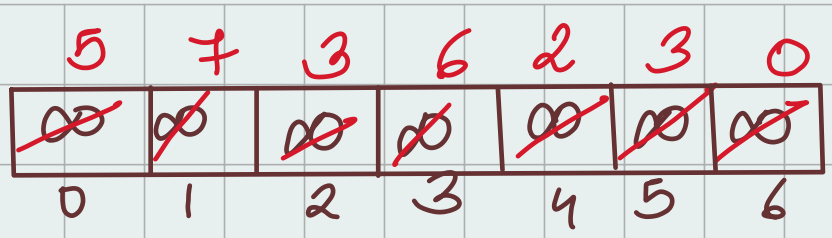
+1 ↙

node = 3, ~~dist = 8~~ already have 6

Similarly for node = 3, dist = 6 ↙ X

Ans :-

dist[ ]



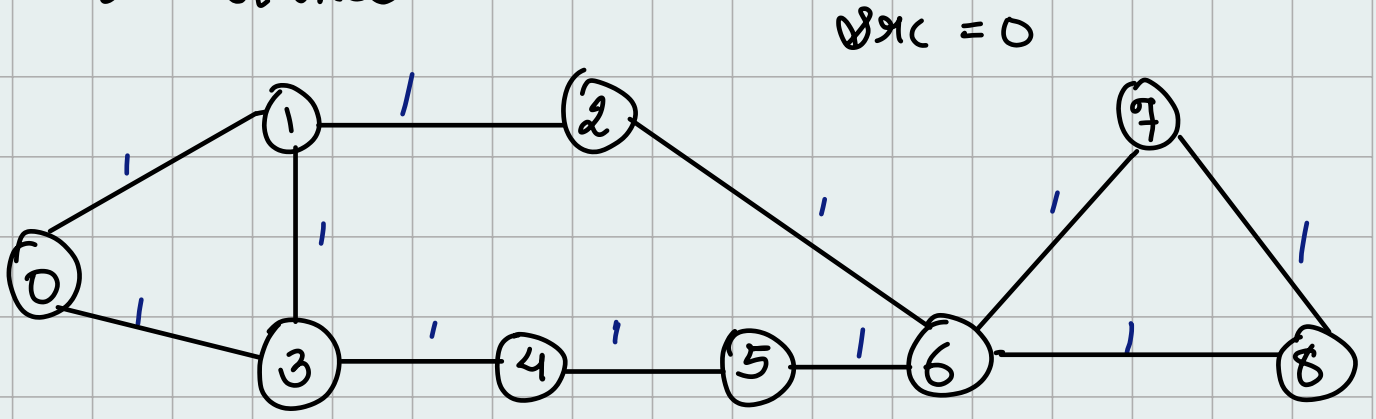
Code :-

```
class Solution{

    public int[] shortestPath (int N, int M,
      int[][] edges){

        Arraylist <Arraylist <pair>> adj =
                                new Arraylist<>();

    for(int i=0; i<N, i++){
        int u = edges[i][0];
        int v = edges[i][1]
        int wt = edges[i][2];
        adj.get(u).add(new pair (v, wt));
    }
    // find the Topological sort

    int[] visited = new int[N];
    Stack <Integer> stack = new Stack<>();
    for (int i=0; i<N; i++){
        if (visited[i] == 0){
```

```
            }   topoSort(i, adj, visited, stack);
        }
        // do the distance thing
        int[] dist = new int[V];
        Arrays.fill(dist, -1);
        dist[0] = 0;

        while(!stack.isEmpty()){
            int node = stack.pop();
            for(pair it : adj.get(node)){
                int v = it.first;
                int wt = it.second;
                if(dist[node] != -1 && (dist[v]
                    == -1 || dist[node]+wt<dist[v])){
                        dist[v] = dist[node] + wt
                }
            }
        }
        return dist;
    }
    private void topoSort(int node, ArrayList<
    ArrayList<pair>> adj, int[] visited, Stack
    <Integer> stack){
        visited[node] = 1;
        for(pair it: adj.get(node)){
            int v = it.first;
```

```
if (visited[v] == 0) {
    toposort(v, adj, visited, stack);
}
}
stack.push(node)
}
```

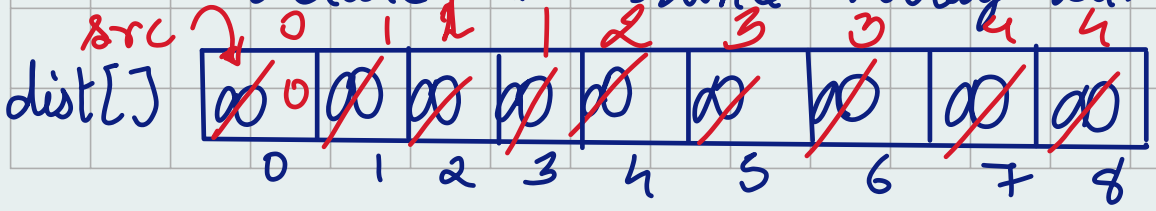
8) Shortest path in undirected graph having unit distance.



make an adjacency list :-

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

then declare a distance array with infinity init.



using BFS

node = 0, dist = 0

node = 1, dist = 1

node = 3, dist = 1

node = 1, dist = 1

node = 0, dist = 2 ✗

node = 3, dist = 2 ✗

node = 2, dist = 2

shortest path is already 1.

Queue
{node, distance}

node = 3, dist = 1

node = 0, dist = 2 ✗

node = 1, dist = 2 ✗

node = 4, dist = 2

---

node = 4, dist = 2

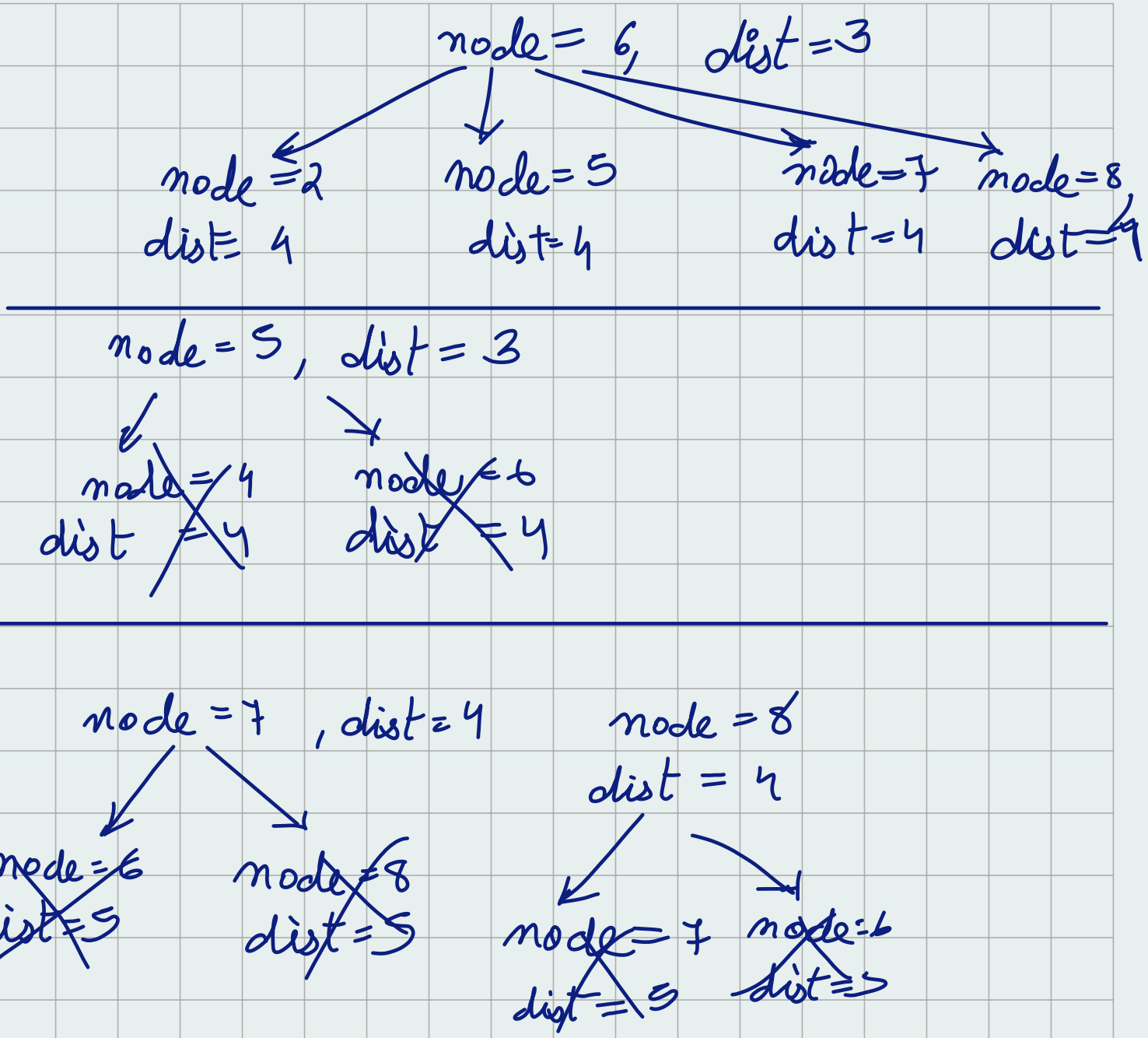
node = 3, dist = 3 ✗

node = 5, dist = 3

node = 2, dist = 2

node = 1, dist = 3 ✗

node = 6, dist = 3



Code :-

```
class Solution {

    public int[] shortestPath (int[][] edges, int n,
    int m, int src ){

        ArrayList<ArrayList<Integer>> adj = new ArrayList<>();
```

```
for (int i = 0 ; i < n ; i++){
    adj.add (new ArrayList <>());
}
for (int i = 0 ; i<m ; i++){
    adj.get (edges[i][0]).add(edges[i][1]);
    adj.get (edges[i][1]).add(edges[i][0]);
}

int[] dist = new int[n];
Arrays.fill (dist, Integer.MAX_VALUE);
dist[src] = 0;

Queue<Integer> q = new LinkedList<>();
q.add(src);

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

    for(int adjNode : adj.get(node)){
        if (dist[node]+1 < dist[adjNode]){
            dist[adjNode] = dist[node]+1;
            q.add[adjNode];
        }
```

```
        }
    }
    // suppose we couldn't reach the node then
    return it with -1
    for(int i=0; i<n ; i++){
        if(dist[i] == Integer.MAX_VALUE){
            dist[i] = -1;
        }
    }
    return dist;
}
```

Time Complexity is same as BFS Algorithm i.e. $V+2E$

8) Word Ladder (I)

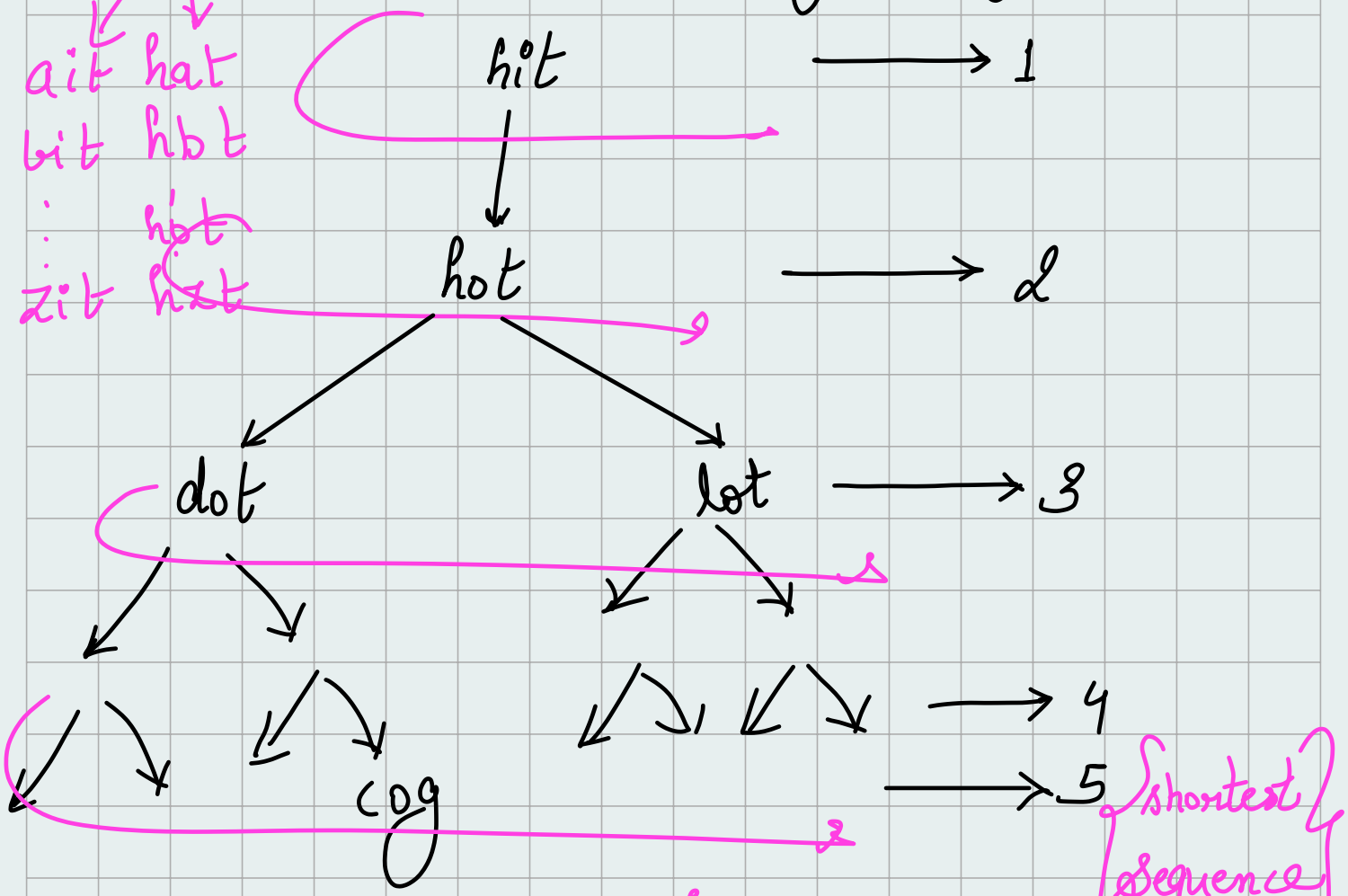
beginWord = "hit"

wordList = [hot, dot, dog, lot, log, cog]

Intuition

hit → hot → dot → dog → cog

ait hat
bit hbt
:
zit hzt

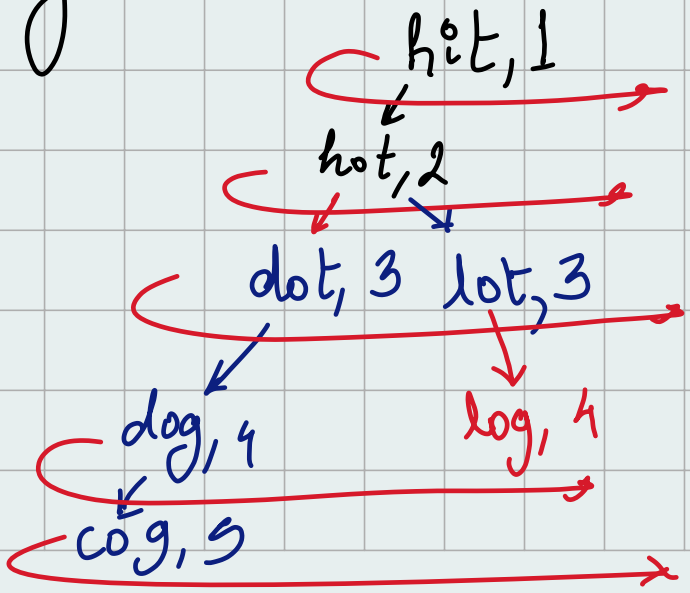


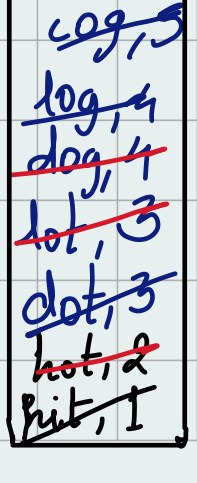
BFS Traversal

Set = { ~~hot~~, ~~dot~~, ~~dog~~, ~~lot~~, ~~log~~, ~~cog~~ }

wordList = [hot, dot, dog, lot, log, cog]

beginWord = hit endWord = cog





ans = 5

else return 0, make a Pair class

Queue
{string, Integer}

If we do linear search in wordList list it will take a lot of time. We create a set data structure and make sure that exact same wordlist is in the set It have a very less TC. If we find the words in set data structure we will delete it from the set & We are not using any visiting arrays as we mark them as visited once visited. So removing the words from the list will make sure of that the word list is not getting repeated and it gives us the correct word length. }

Code :-

```
class Pair{

    String word;
    int length;

    Pair ( String word, int length){
        this word = word,
        this . length = length,
    }
```

```
}

public class wordLadder{
    public int wordLadderLength( String startWord,
        String targetWord, String[] wordList ){

        Queue<Pair> q = new LinkedList<>();

        q.add( new Pair( startword, 1 ) );

        Set< String> set = new HashSet<> ();

        for( String word : wordList){
                set. add (word);
        }

        while (!q.isEmpty())

            String word = q.peek(). word;

            int steps = q. peek (). length;

            q. poll ();

            if (word . equals( targetWord) ){
                        return steps;
            }
```

```
// word = hot → cog

for(int i=0; i<word.length(); i++){

    for(char ch='a'; ch<='z', ch++){

        char[] replacedCharArray = word.
                                   toCharArray();

        replacedCharArray[i] = ch;

    // if it exist in the set then add it
       in the queue.

        String replacedWord = new String
                              (replacedCharArray);

        if(set.contains(replacedword)){
            set.remove(replaced word);
            q.add(new Pair(replacedWord,
                           steps+1));

}}}}
```

```
// if we could not find the target word,
// then return 0.
return 0;
}
}
```

Q) Word Ladder - II

bat → pat → pot → poz → coz

bat → bot → pot → poz → coz

wordlist = { ~~pat~~, ~~bot~~, ~~pot~~, ~~poz~~, ~~coz~~ }

beginWord = bat endWord = coz

{bat} word = bat

pat bot

After the level is completed then only delete the words from wordlist

{bat, bot, pot, poz, coz}] level 5
{bat, pat, pot, poz, coz}
~~{bat, bot, pot, poz}~~] level 4
~~{bat, pat, pot, poz}~~
~~{bat, bot, pot}~~] level 3
~~{bat, pat, pot}~~
~~{bat, bot}~~
~~{bat, pat}~~] level 2
~~{bat}~~] level 1

{bat, pat} word = pat → pot

We cannot erase pot from wordlist as the entire level needs to be completed.

{bat, bot} word = bot → pot

Now we can erase the pot from wordlist

{bat, pat, pot} word = pot → poz

{bat, bot, pot} word = pot → poz

{bat, pat, pot, poz} word = poz → coz

{bat, bot, pot, poz} word = poz → coz

{ bat, pat, pot, poz, coz }

↳ got last word (ending sequence 1)

{ bat, bot, pot, poz, coz }

↳ ending sequence-2

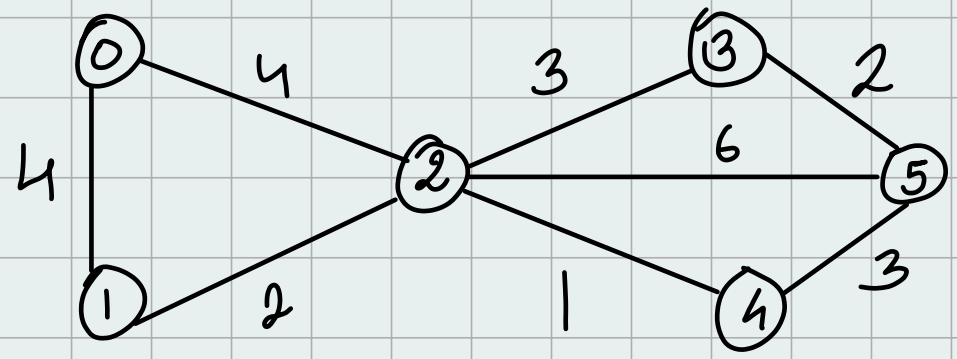
store it in your ans arraylist

store it in arraylist

{ ans }

length word > 5 { break }

* Dijkstra's Algorithm



Adjacency List

0 → {1, 4} {2, 4}

1 → {0, 4} {2, 2}

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

3 → {2, 3} {5, 2}

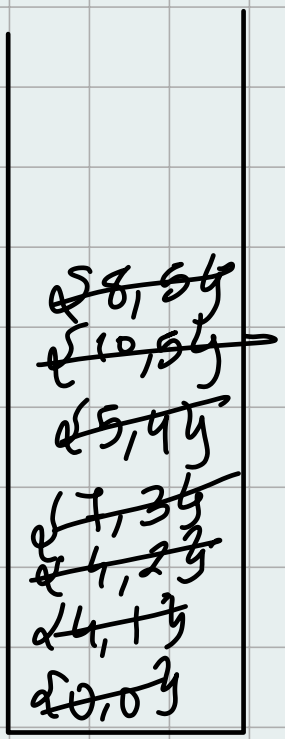
4 → {2, 1} {5, 3}

5 → {2, 6} {3, 2} {4, 3}

d = 0,

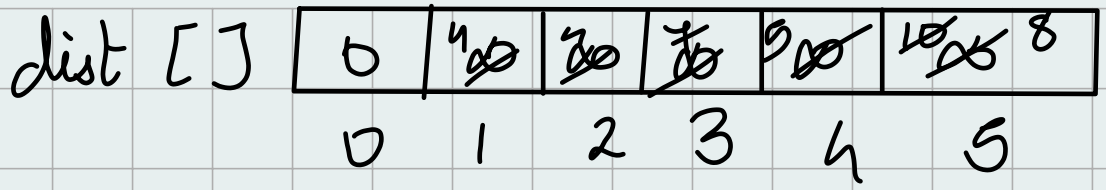
node = 0 —(+4)→ node = 1, dist = 4

node = 0 → node = 2, dist = 4



min-Heap (dist, node)

taking out lesser distance node first in ascending order (min heap)



d=4

node=1

d=4 → node=0, dist=8

dist=2 → node=2, dist=6

---

dist=4

node=2

+4 → node=0, dist=8

+2 → node=1, dist=6

+3 → node=3, dist=7

+1 → node=4, dist=9

+6 → node=5, dist=10

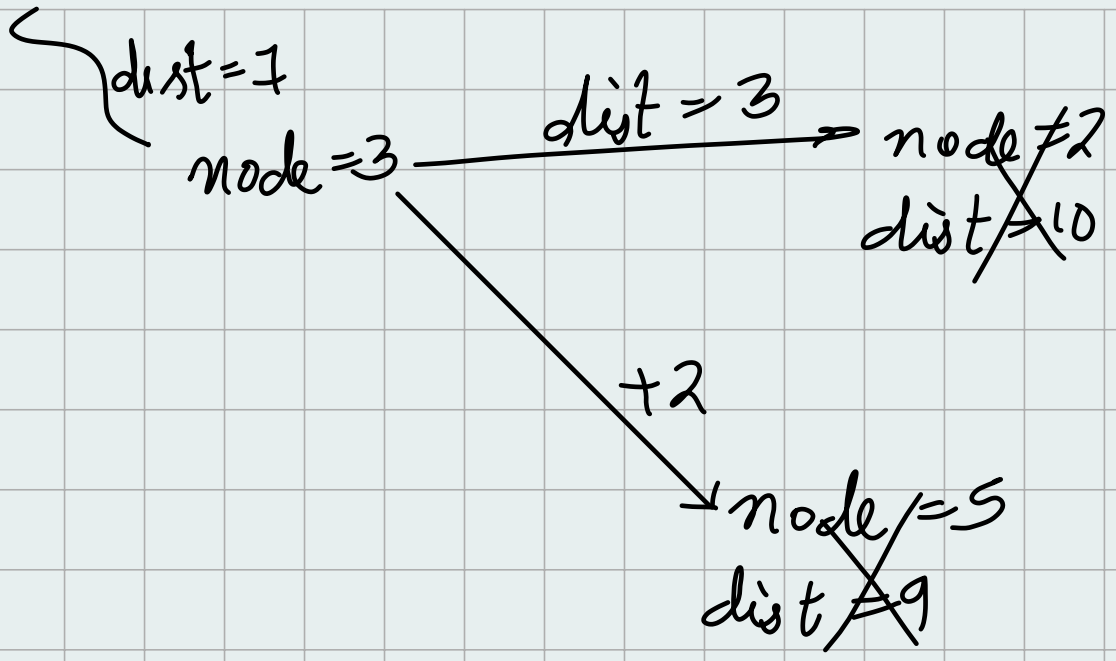
---

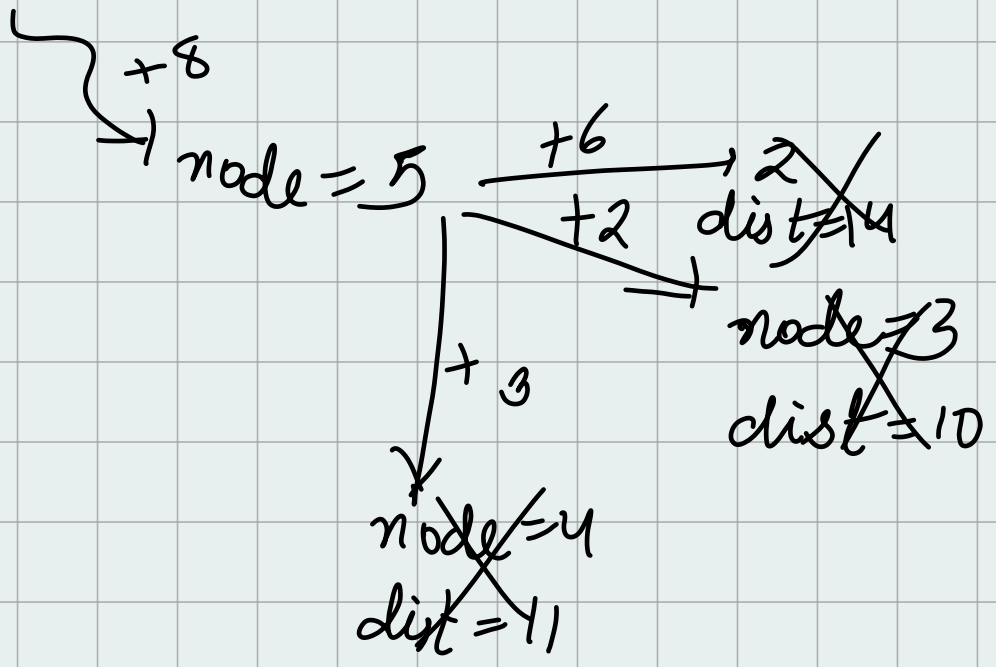
d=5

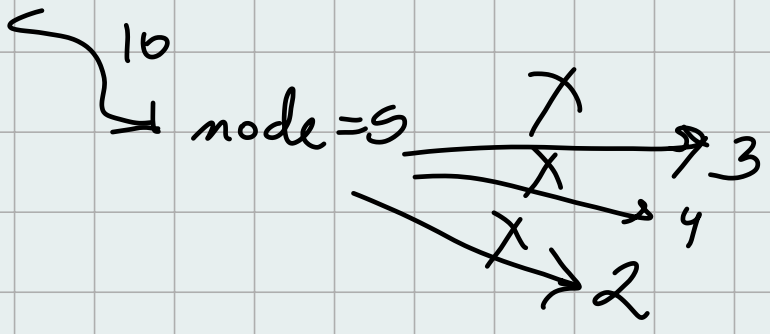
node=4

+1 → node=2, dist=6

node=5, dist=8







class solution {

static int [] dijkstra (int V, ArrayList<
ArrayList< ArrayList <Integer>>> adj, int S) {

```
// create a priority Queue for storing the
  nodes as a pair { distance, node }
//where distance is the dist from src to node

PriorityQueue<Pair> pq = new PriorityQueue
                          <>((x,y) -> x.distance - y.
                                      distance));

int[] dist  = new int[V];

Arrays.fill(dist, Integer.MAX_VALUE);

dist[S] = 0

pq.add(new Pair(0,S));

while(!pq.isEmpty()){

    int dis = pq.peek().distance;
    int node = pq.peek().node;
    pq.remove();

  // check for all adjacent Nodes that popped
      out element whether the prev dist is
  // larger than current or not

  for(int i=0; i<adj.get(node).size();i++){
```

```
int edgeWeight = adj.get(node).get(i).get(1);
```

List<List<List<Integer>>>

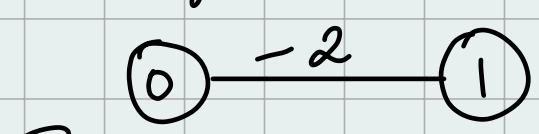
get(i) get(j) get(1)

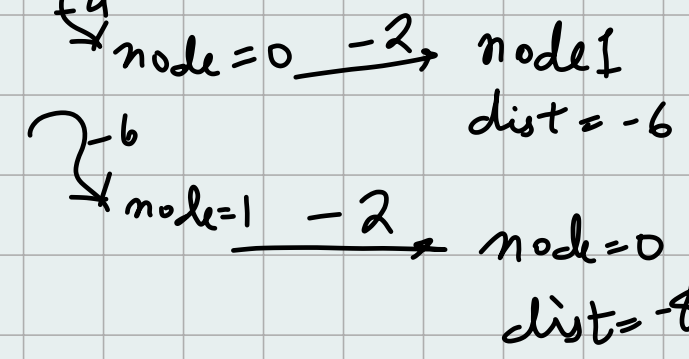
```
int node = adj.get(node).get(i).get(0);

// if current distance is smaller than put it
into the queue

if (dist[adjNode] > dis + edgeWeight)
    dist[adjNode] = dis + edgeWeight;
    pq.add(new Pair(dist[adjNode],
                    adjNode));
}
}
}

return dist;
}
}

Class Pair {
    int node;
    int distance;
```

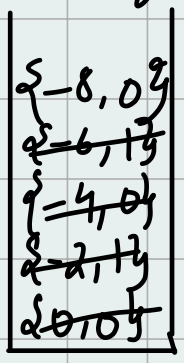
```
public pair (int distance, int node){
    this.node = node;
    this.distance = distance.
}
};
```

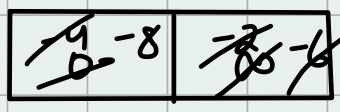
Note: The dijsktra's algorithm won't work for negative weight cycle.

* Because of this it will fall in infinite loop.









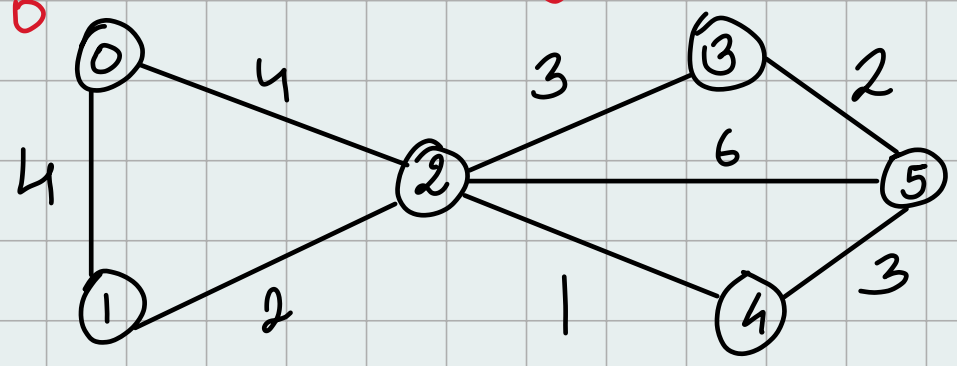
Time complexity : $E \log V$

Total number of edges

Total number of nodes

*Q) Dijkstra's Algorithm - Using set

src = 0



dist[] =

0	∞ 4	∞ 4	∞ 7	∞ 5	∞ 10
0	1	2	3	4	5

Set DS stores the element in ascending order The smallest would be at the top.

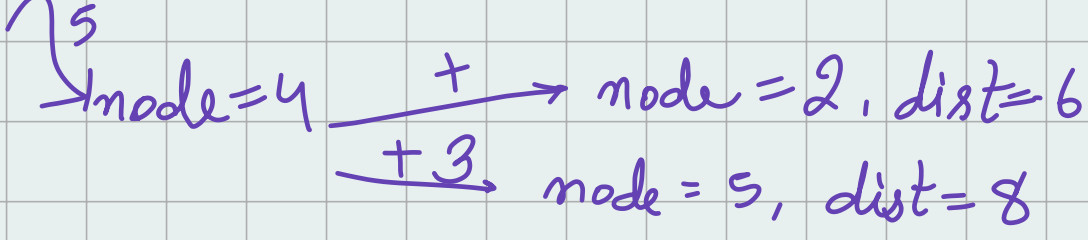
Set DS
{dist, node}

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

node = 0 —(+4)→ node = 1, dist = 4
node = 0 —(+4)→ node = 2, dist = 4

node = 1 —(+4)→ node = 0, dist = 8 ✗
node = 1 —(+2)→ node = 2, dist = 6

node = 2 —(+4)→ node = 0, node = 8
node = 2 —(+2)→ node = 1, node = 6
node = 2 —(+3)→ node = 3, dist = 7
node = 2 —(+6)→ node 5, dist = 10
node = 2 —(+1)→ node = 4, dist = 5



Now {10,5} is already in the set and dist 10 is there in node 5 It means someone already reached 5 with a distance 10. But we got a better distance than 10 which is 8 in node 5 so set will erase the ~~{10,5}~~ and keep the {8,5} in the set D.S This is what was missing in Priority Queue data structure. An extra (not needed) iteration using extra time & space complexity. Set will erase already existing path But it will take a logorathimic time to erase inside the set. So a minute difference in T.C.

* In Java using TreeSet or a HashSet is not possible.

So no Set code

---

* PQ Time complexity derivation

```
while (!pq.isEmpty()) ⟶ V
{
```

dis, node = pop() → log(heap size)

for(iterate on adjacentNodes)

{

if (conditional check)

{

update dist,

y push/add in pq

y y

$O(V \times (\text{pop vertex from min heap} + \text{no of edges on each vertex} \times \text{push onto pq}))$

$O(V \times (\log(\text{heap size}) + ne \times \log(\text{heap size}))$ (ne: no. of edges)

$O(V \times (\log(\text{heap size}) \times (\cancel{ne}^{(V-1)} + X))$

$O(V \times V \log(\text{heap size}))$

$O(V^2 \times \log(\text{heap size}))$

$O(V^2 \times \log(V^2))$

$O(E \times 2 \times \log V) \approx O(E \log V)$

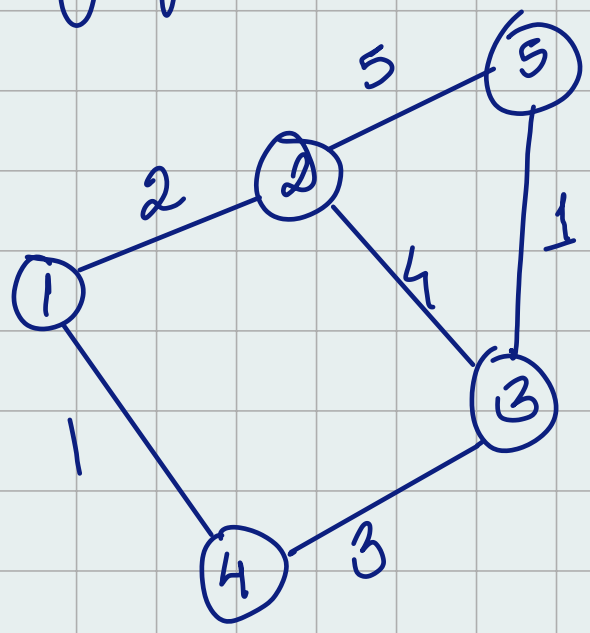
one vertex can have $(V-1)$ edges.

$E = V^2$

Total eges

$V \times (V-1) \approx V^2 \longrightarrow E$ (Total number of edges)

Q) Shortest Path in weighted undirected graph (Print the shortest path)



src = 1

destination = 5

shortest path :-

{1 → 4 → 3 → 5}

or else

{-1} [if not possible shortest path then return -1.]

To remember path we have to maintain a parent[] and backtrack it

parent

1	1	2	4	1	4	3

src=1

dist[]

0	2	4	1	5
1	2	3	4	5

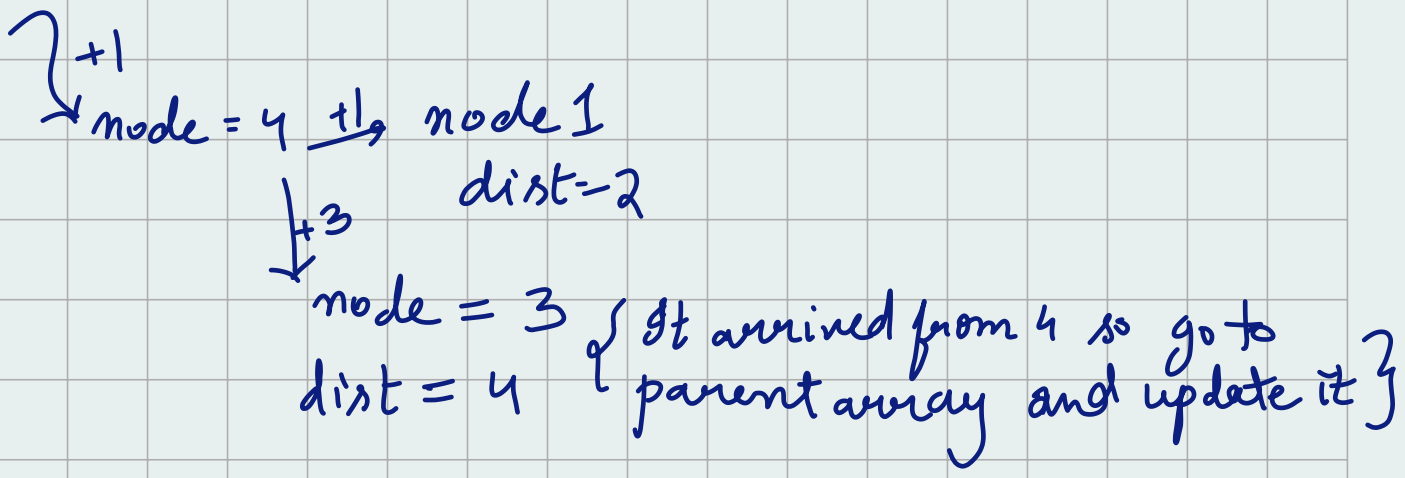
0 → node 1 —+2→ node 2 dist=2 ({2 came in from 1} so update 2 to 1)

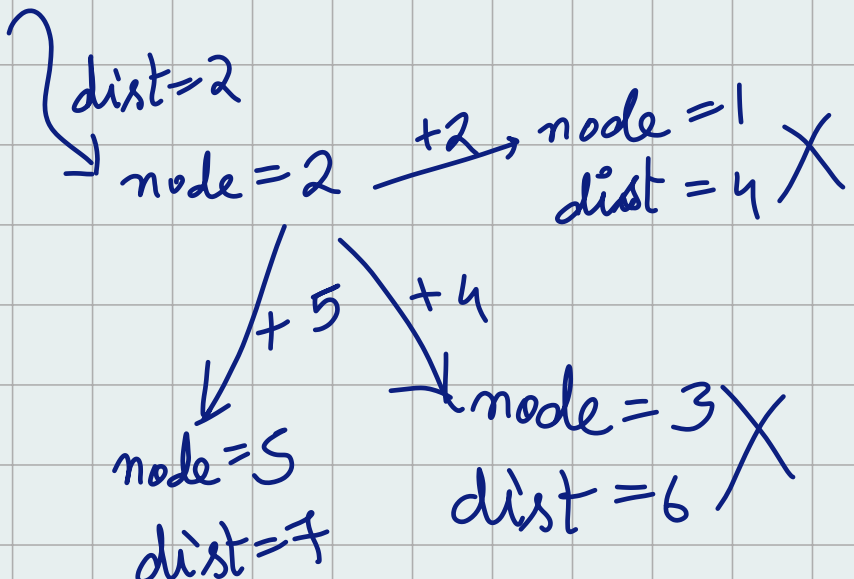
node 1 —+1→ node 4 dist=1

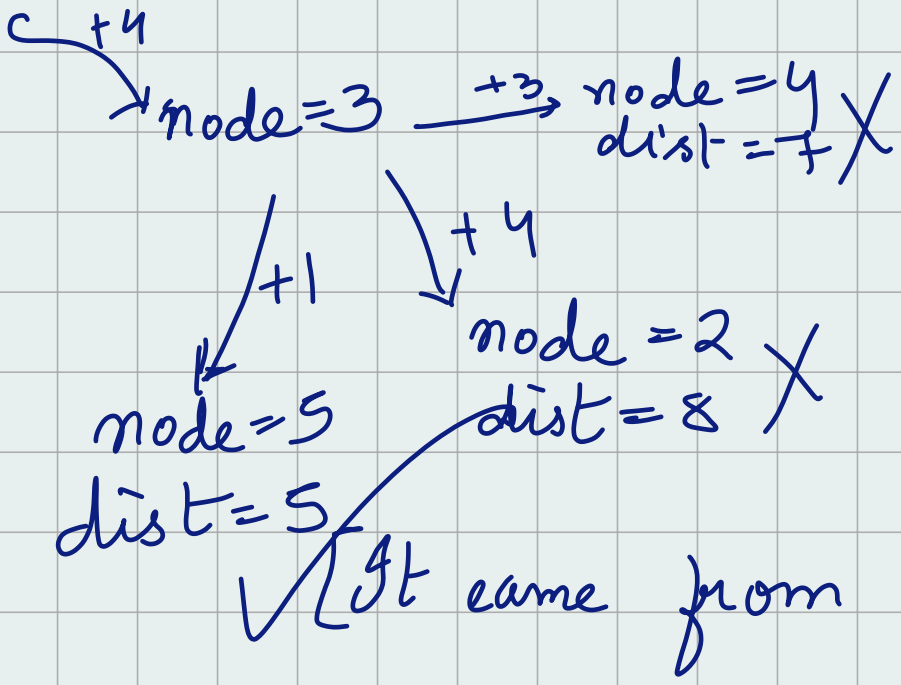
PQ

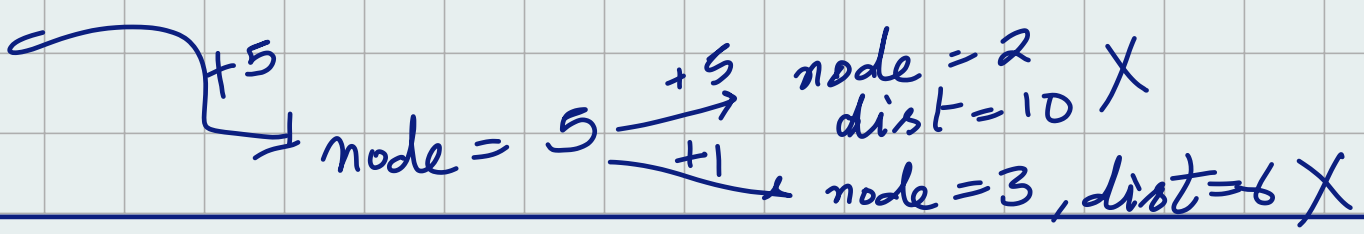
{5, 5}
{3, 5}
{4, 3}
{1, 4}
{2, 2}
{0, 1}

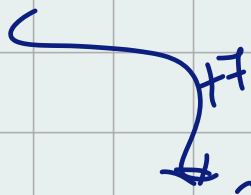
PQ {dist, node}


+1
node = 4
+1
node 1
dist = 2
+3
node = 3
dist = 4
{ It arrived from 4 so go to parent array and update it }


dist = 2
node = 2
+2
node = 1
dist = 4
X
+5
+4
node = 5
dist = 7
node = 3
dist = 6
X

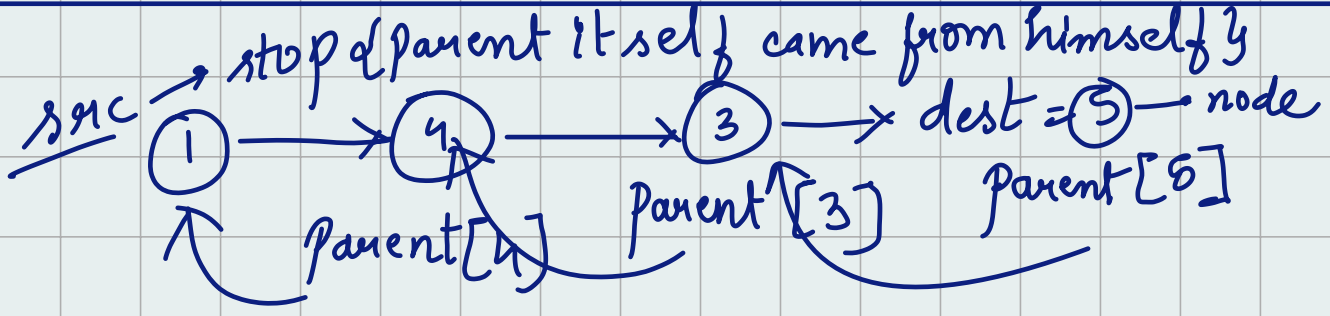

+4
node = 3
+3
node = 4
dist = 7
X
+1
+4
node = 5
dist = 5
node = 2
dist = 8
X
[It came from 3 to 5]


+5
node = 5
+5
node = 2
dist = 10
X
+1
node = 3, dist = 6
X



node = 5

It will yeild me nothing as we saw {5.5} above.



node = ~~5~~ ~~3~~ ~~4~~ 1 — break

arr → [ ] 5 3 4

```
while (parent[node] != node){
    // store the node
    arr.add(node)
    node = parent[node]
}
arr.add(src); // add it manually
collection.reverse(arr);
```

{1 → 3 → 4 → 5}

* Code

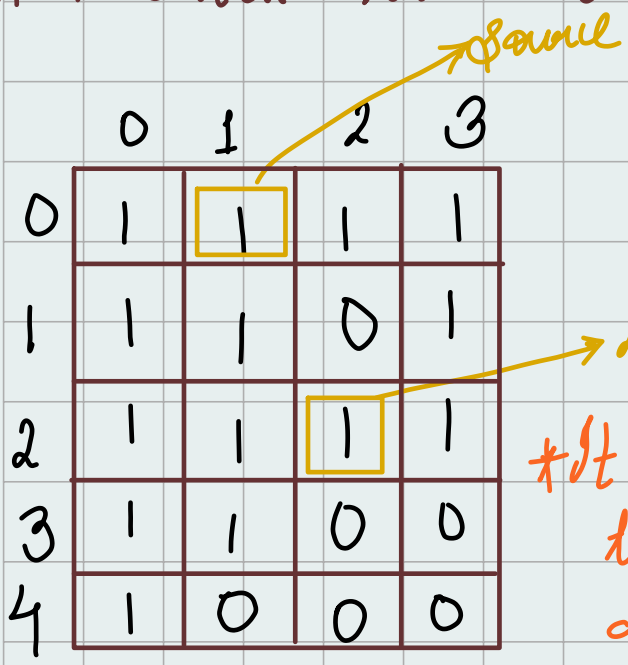
```
public class PrintPath{

    public static List<Integer> shortestPath(
            int n, int m, int[][] edges){

        ArrayList<ArrayList<Pair>> adj =
                new ArrayList<>();

        for(int i=0, i<=n; i++){
            adj.add(new Arraylist<>());
        }
        for(int i=0; i<m; i++){
            adj.get(edges[i][0]).add(
                new Pair(edges[i][1], edges[i][2]));

            adj.get(edges[i][1]).add(new Pair(
                edges[i][0], edges[i][2]));
        }

        // create a priority Queue for storing the
        nodes along with distances in the form of
        a pair {dist, node}
```

```
PriorityQueue<pair> pq = new
    PriorityQueue<>((x,y) -> x.first -
        y.first);

int[]dist = new int[n+1] // from 1 to N
int[]parent = new int[n+1] // from 1 to N.

Arrays.fill(dist, Integer.MAX_VALUE);

for(int i=0; i<=n ; i++){
    parent[i] = i;
}

dist[1] = 0;

pq.add(new Pair(0, 1));
    // distance 0, node 1

while(!pq.isEmpty()){

    pair it = pq.peek();

    int dis = it.distance;

    int node = it.node;
    pq.poll();
```

```
for (Pair iter : adj.get(node)){
    int adjNode = iter.node;
    int edgeWeight = iter.distance;
    if (dis + edgeWeight < dist[
                                adjNode]){
        dist[adjNode] = dis + edgeWeight;
        pq.add(new Pair(dist[adjNode]
                                adjNode));
    }
}
}
List<Integer> path = new ArrayList<>();
if (dist[n] == Integer.MAX_VALUE){
    path.add(-1);
    return path;
}
```

```
// backtracking the path // O(N) + Elog V
int node = n;
while (parent[node] != node){
    path.add(node);
    node = parent[node];
}
path.add(1);
Collections.reverse(path);
return path;
}
}
```

* Shortest Distance in a Binary maze



Dijkstra's Algorithm

*It can only travel in 4 directions

As the dist^n is in the increasing fashion order simple Queue is efficient. Cause the log(N) is not added in the T.C. now

dist[ ][ ]

(3, (1,3))
(2, (1,0))
~~(2, (2,1))~~
~~(2, (0,3))~~
~~(1, (0,0))~~
~~(1, (1,1))~~
~~(1, (0,2))~~
~~{ 0 {0,1} }~~ only

~~Priority~~ Queue

{ dist, { row, col } }

+0

(0,1) +1→ (0,2) dist=1 ✗ cannot be considered

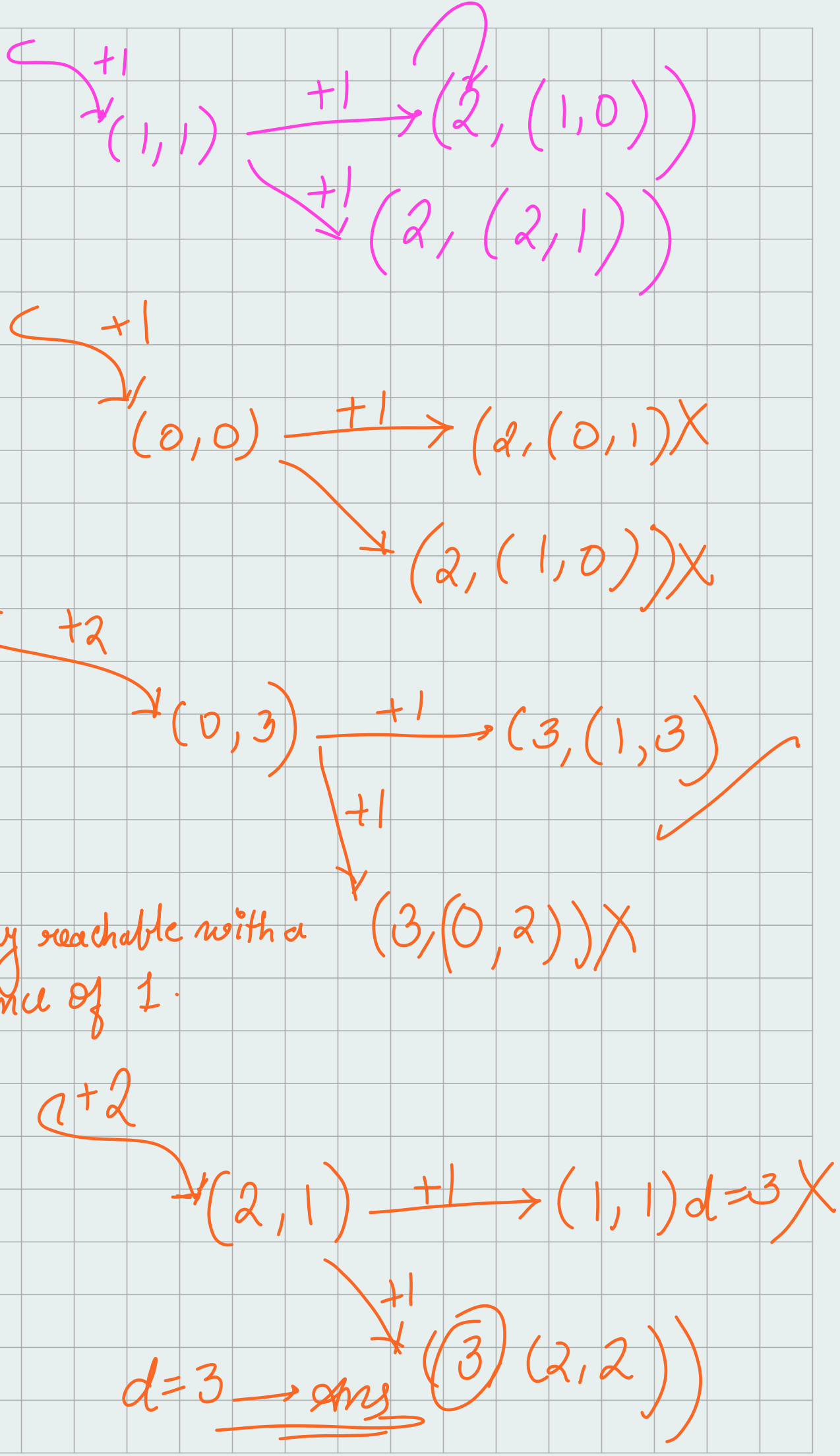
(0,1) +1↓ (1,1) dist=1

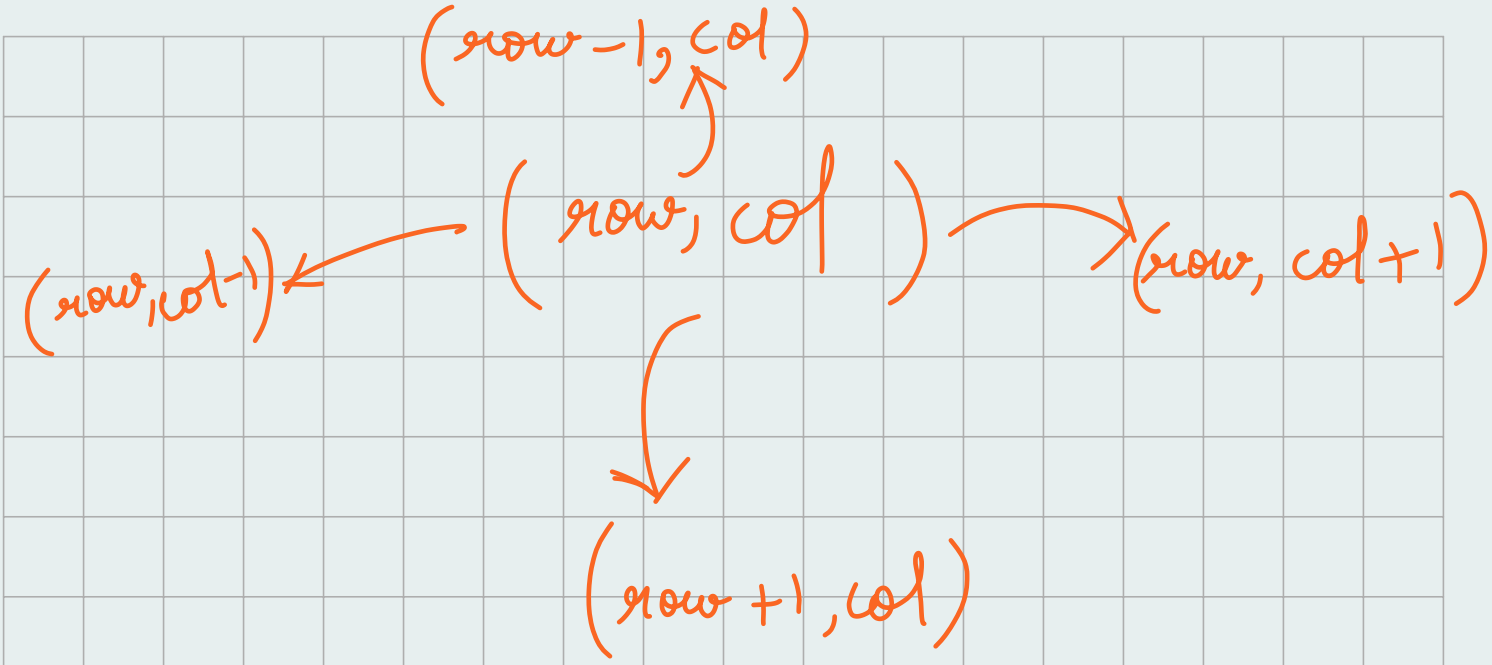
(0,1) +1↙ (0,0) dist=1

1

(0,2) +1→ (0,3) dist=2

(0,2) +1↓ (1,2) dist=2


+1
(1,1)
+1
(2,(1,0))
+1
(2,(2,1))
+1
(0,0)
+1
(2,(0,1))X
(2,(1,0))X
+2
(0,3)
+1
(3,(1,3)
+1
(3,(0,2))X
Already reachable with a distance of 1.
+2
(2,1)
+1
(1,1) d=3 X
+1
(3) (2,2))
d=3 → ans



del row [ ] = { -1, 0, 1, 0 }

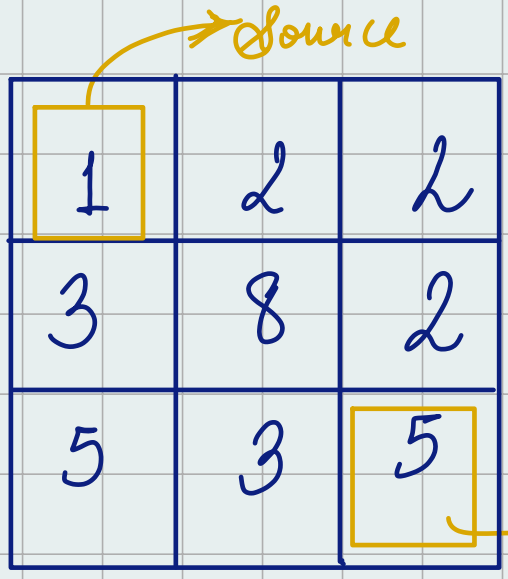
del col [ ] = { 0, +1, 0, -1 }

index ⟶ 0 1 2 3

Code :-

```
public class shortestDistance {

    int shortestPath(int[]
```

* Path with minimum effort :-



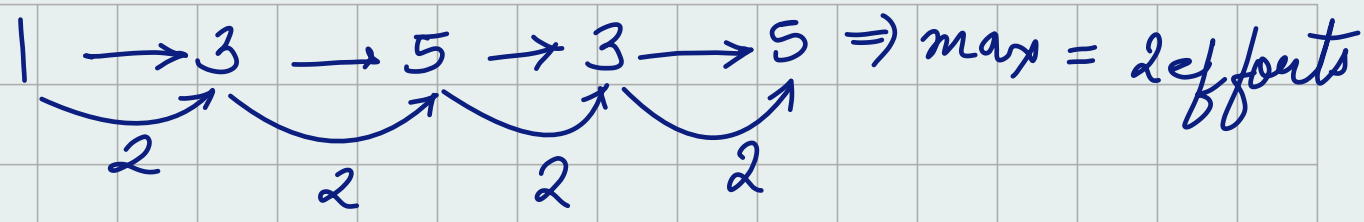
* Max of all difference in the path is the effort

$1 \to 2 \to 2 \to 2 \to 5 \Rightarrow (max) = 3$ effort

(absolute Difference) 1 0 0 3

$1 \to 3 \to 8 \to 2 \to 5 \Rightarrow max = 6$ effort

2 5 6 3



$\Rightarrow$ max = 2 efforts

now, min (3, 6, 2) = 2 Ans

Source

1	2	2
3	8	2
5	3	5

Destination

dist[ ][ ]

	0	1	2
0	0	∞ 1	∞ 1
1	∞ 2	5 ∞ 6	∞ 1
2	∞ 2	∞ 2	2 ∞ 3

effort → { Max of all difference in the path }

(2,(2,2))
(2(2,1))
(2(2,0))
(5(1,1))
(3,(2,2))
(1,(1,2))
(6,(1,1))
(1,(0,2))
(2(1,0))
(1,(0,1))
(0,(0,0))

{diff, row, col}

Priority Queue (min heap)

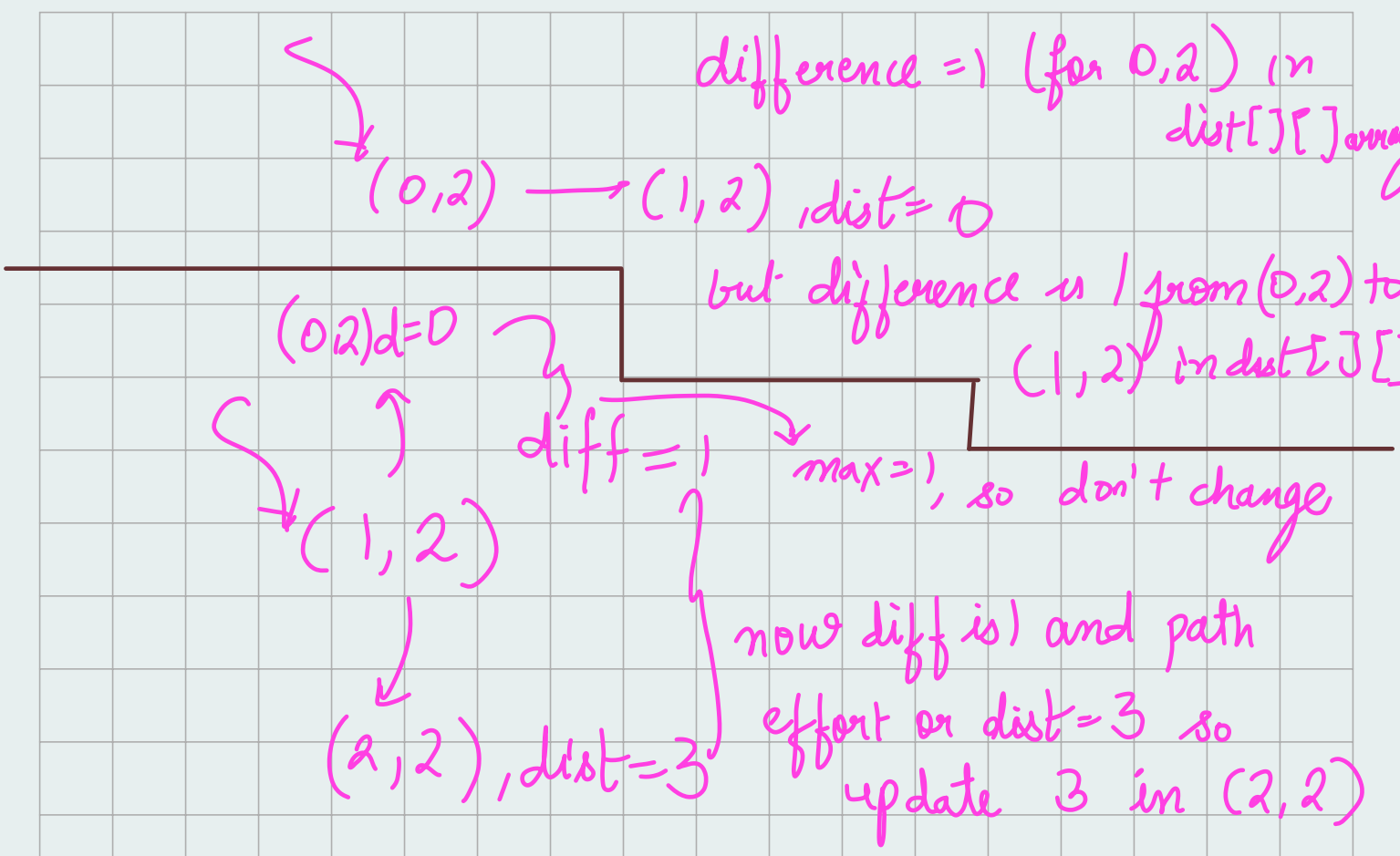
+0 → (0,0) difference = 1

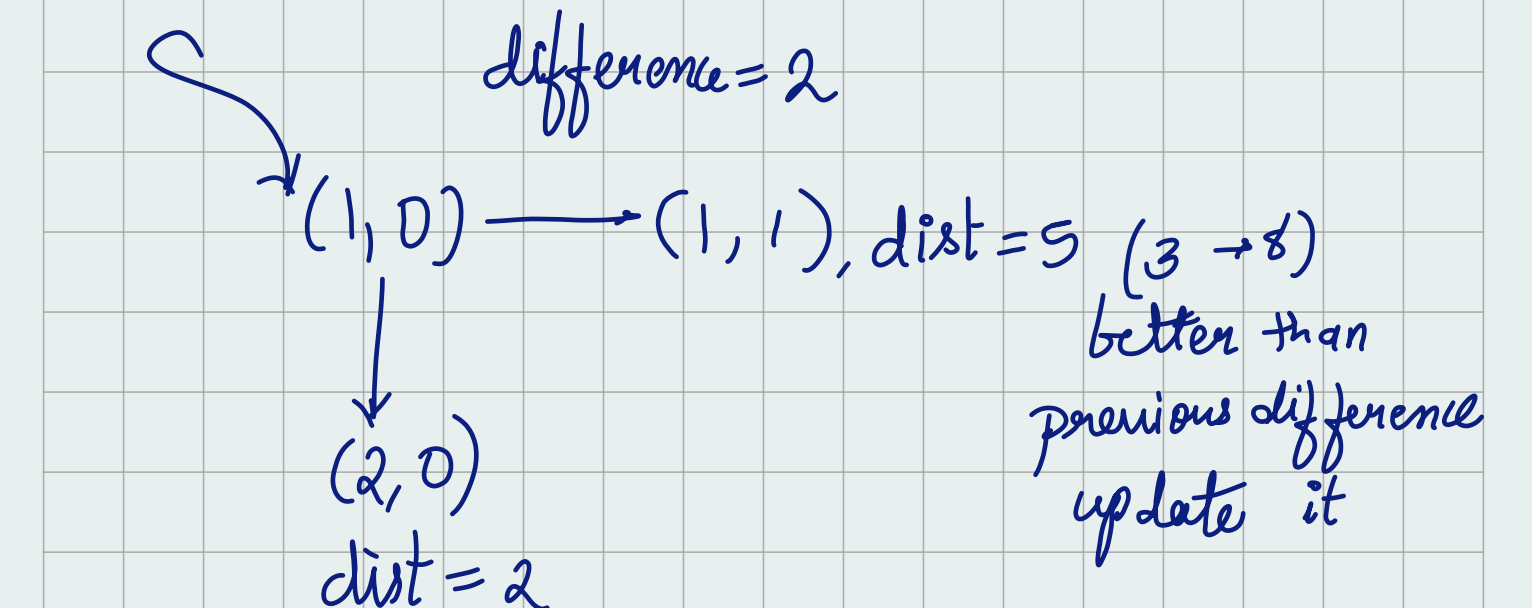
(0,0) +1 → (0, 1), dist=1

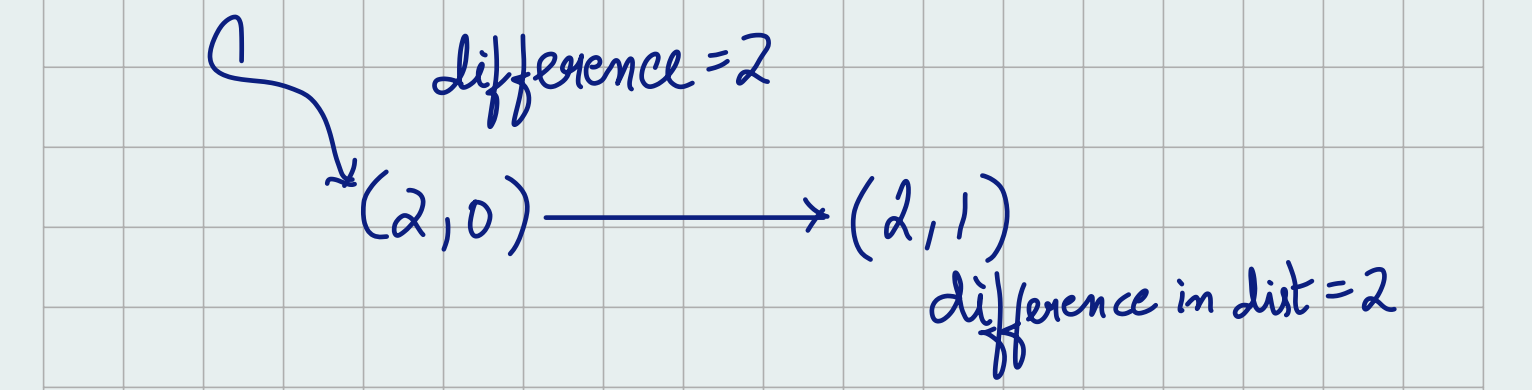
(0,0) +2 → (1, 0), dist = 2

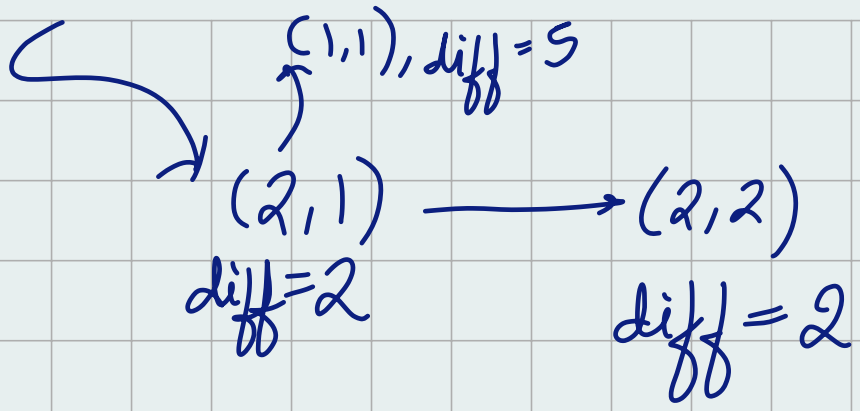
+1 (0,1) → (0,2), dist = 0 difference = 1

(0,1) → (1,1) dist=6


difference = 1 (for 0,2) in dist[][] array
(0,2) → (1,2), dist = 0
but difference is 1 from (0,2) to (1,2) in dist[][]
(0,2) d=0
diff = 1
max = 1, so don't change
(1,2)
now diff is 1 and path effort or dist = 3 so update 3 in (2,2)
(2,2), dist = 3


difference = 2
(1,0) → (1,1), dist = 5
(3 → 8)
better than previous difference update it
(2,0)
dist = 2


difference = 2
(2,0) → (2,1)
difference in dist = 2



---

d = 2

(2,2) → destination

---

* Check for destination while taking out of pq and not while inserting.

effort → 2

Code :-

```
public class minimumEffort {

    public int minimumEffortPath ( int rows,
        int columns, int [][] heights ) {

        PriorityQueue <Tuple> pq = new
```

```
PriorityQueue<>((a, b) -> a.distance -
  b.distance);

int n = heights.length;
int m = heights[0].length;

int[][] distance = new int[n][m];

for(int i=0; i<n; i++){

    for(int j=0; i<m; j++){

        distance[i][j] = Integer.
                         MAX_VALUE;

    }
}
distance[0][0] = 0;

pq.add(new Tuple(0, 0, 0));

int[] delRow = {-1, 0, 1, 0};

int[] delCol = {0, 1, 0, -1};

// n*m
```

```
while (!pq.isEmpty()){
    Tuple itr = pq.poll();
    int diff = itr.distance;
    int row = itr.row;
    int col = itr.col;
    // 4 directions
    for (int i=0; i<4; i++){
        int newRow = row + delRow[i];
        int newCol = col + delCol[i];
        if (row == n-1 && col ==
            m-1){
            return diff;
        }
        if (newRow >=0 && newCol>=0
            && newRow<n && newCol<m)
        {
            int newEffort = Math.max
```

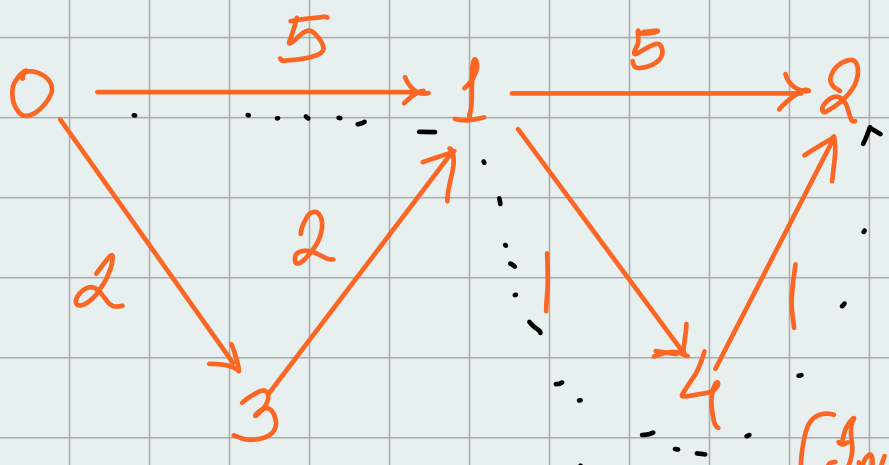
```
(diff), Math.abs(heights[row]
[col] - heights[newRow]
[newCol]));

if(newEffort < distance[newRow]
        [newCol]) {

    distance[newRow]
    [newCol] = newEffort;

    pq.add(new Tuple(
    newEffort, newRow, newCol));
}
}
}
}
}

return 0;
}
}

class Tuple {
```

```
int distance;
int row,
int col ;

public Tuple( int distance, int row, int col){

    this distance = distance;
    this.row = row;
    this.col = col;

}

}
```

Q) Cheapest flights within K stops.

$src = 0$
$dest = 2$
$k = 2$

Here storing shortest distance will not work cause we have to complete maximum 2 stops in between, & taking the last destination. In Priority Queue the dist array will stop the algorithm and return the inbetween ans even if the passenger not completely went to dest. Queue will work perfectly



src = 0
dest = 2
k = 2

(Including desⁿ)
stops <= 3

dest

src 4 1

dist[ ] =

0	~~5~~ ~~6~~	~~∞~~ ~~∞~~	~~∞~~ 2	~~∞~~ ~~6~~	5
0	1	2	3	4	

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

{stops, node, dist}

0
node = 0 —+5→ node = 1
dist = 5
stop = 1

+2

node = 3
dist = 2
stop = 1

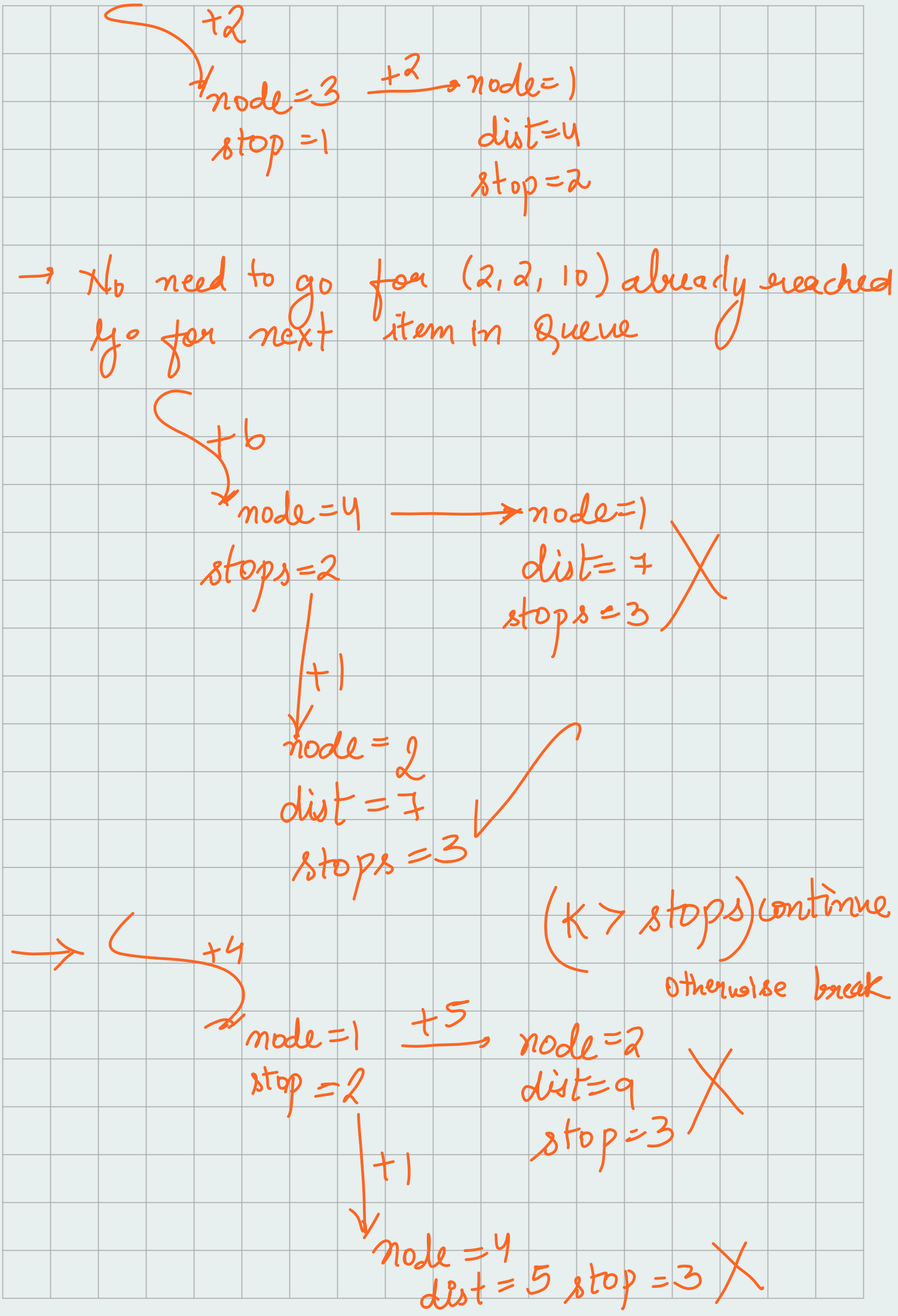
+5

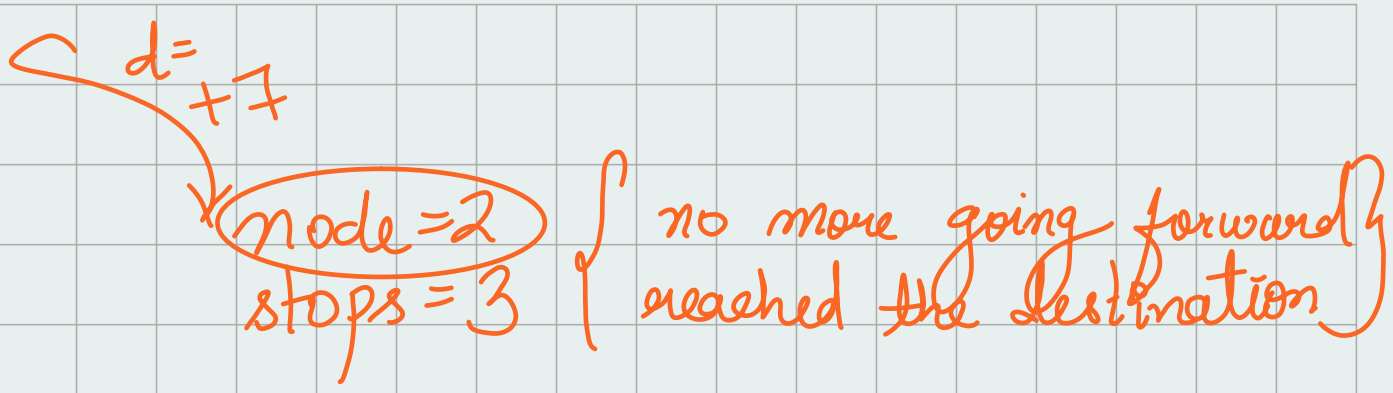
stop = 1 {previously}

node = 1 —+5→ node = 2
dist = 10
stop = 2

+1

node = 4
dist = 6
stop = 2


+2
node=3
stop=1
+2
node=1
dist=4
stop=2
→ No need to go for (2, 2, 10) already reached
go for next item in Queue
+6
node=4
stops=2
node=1
dist=7
stops=3
+1
node=2
dist=7
stops=3
(K > stops) continue
otherwise break
+4
node=1
stop=2
+5
node=2
dist=9
stop=3
+1
node=4
dist=5 stop=3



Code :-

```
public class cheapestFlight{

    public int CheapestFlight(int n, int[][] flights,
      int src, int dst, int k){

        Arrayhist< ArrayList<Pair>> adj = new
                                ArrayList<>();

        for( int i=0;  i<n ; i++){
              adj.add(new Arrayhist<>());

        }

        //Fill adjacency list with flight data.

        for( int[] flight : flights ){
              adj.get( flight[0]).add(
                  newPair( flight[1], flight[2])),
```

```
Queue<Tuple> q = new LinkedList<>();
q.add(new Tuple(0, src, 0));
int[] distance = new int[n];
Arrays.fill(distance, Integer.MAX_VALUE);
distance[src] = 0;
while (!q.isEmpty()){
    Tuple itr = q.poll();
    int stops = itr.stops;
    int node = itr.node;
    int cost = itr.cost;
    // continue if the number of steps
    is allowed within the range
    if (stops <= K){
        for(Pair iter : adj.get[node]){
            iter adjNode = iter.node;
```

```
                int edW = iter.distance;

                if(cost+edW < distance[
                                adjNode ]){

                    distance[adjNode] =
                        cost + edW;

                    q.add(new Tuple(
                    stops+1, adjNode, cost+edW));
                }
            }
        }
    }
    if(distance[dst] == Integer.MAX_VALUE){
        return -1;
    }

    return distance[dst];
}
}
```

```
class Pair {

    int node;
    int distance;

    public Pair(int node, int distance) {
        this.node = node;
        this.distance = distance;
    }
}

class Tuple {

    int stops;
    int node;
    int cost;

    public Tuple(int stops, int node,
                 int cost) {
        this.stops = stops;
        this.node = node;
        this.cost = cost;
    }
}
```

* Minimum multiplications to reach end.

mod = [100000]

start = 3 end = 30

arr[] = {2, 5, 7}

start = 3 $\xrightarrow{x2}$ 6 $\xrightarrow{x5}$ 30

start = 7 end = 66175

arr[] = {3, 4, 65}

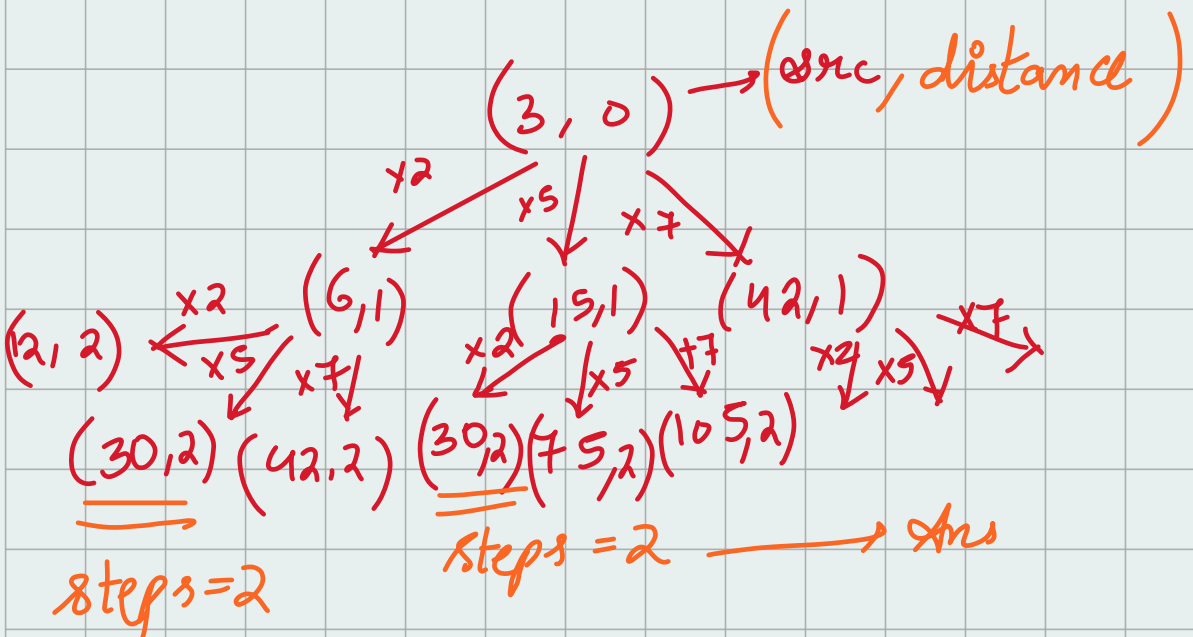
start = 7 $\xrightarrow{x3}$ 21 $\xrightarrow{x3}$ 63 $\xrightarrow{x65}$ 4095 $\xrightarrow{x65}$ 266175

→ now mod it number is too big

→ 266175 % 100000

ans 66175

start = 3 arr[] = {2, 5, 7} end → 30

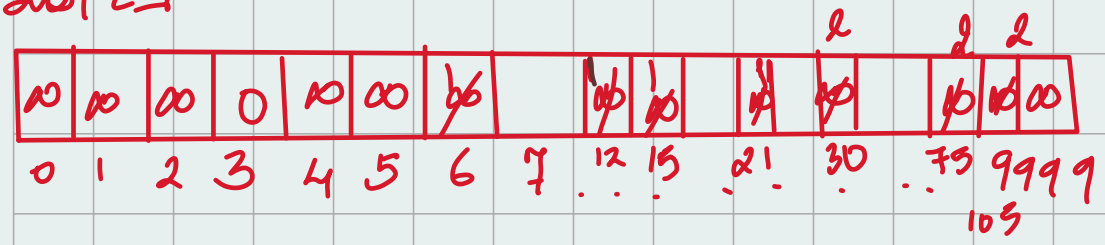


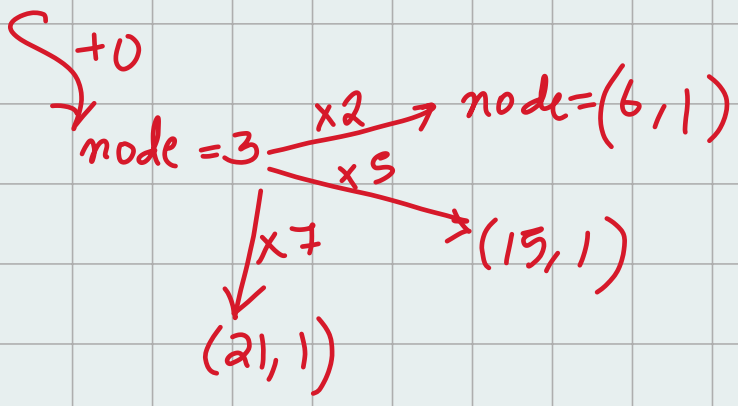
0, 1, 2, 3, 4, ... 9998, 9999

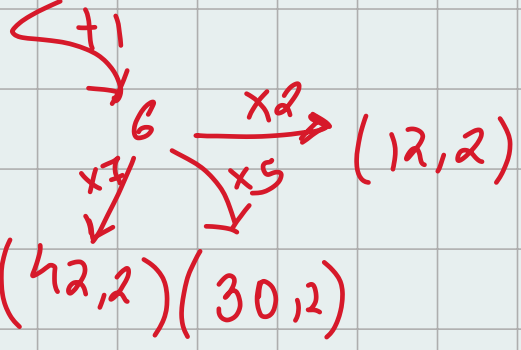
start = 3, end = 75, mod = 100000

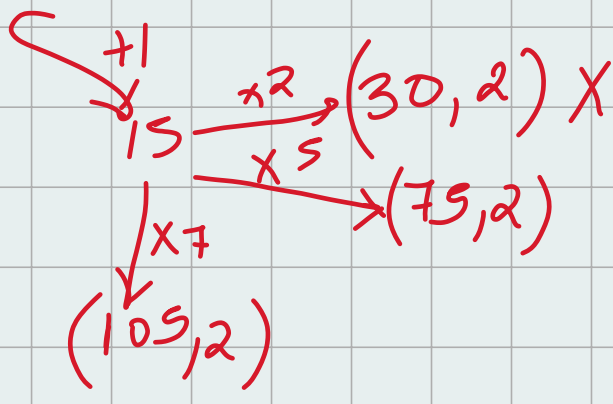
arr = {2, 5, 7}

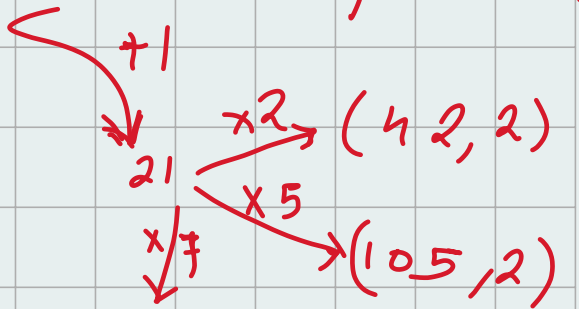
dist[]











(2,147)
(2,105)
(2,42)
(2,105)
(2,75)
(2,30)
(2,42)
(2,30)
(2,12)
~~(1,21)~~
~~(1,15)~~
~~(1,6)~~
~~(0,3)~~

Queue
(steps, num)

Similarly → for 2 steps at 75 (147, 2)

will return with 2 steps'