//Tabulation

```
int[][] dp[n+1][m+1] = {0};
    for(int i:0→n+1)  dp[i][0] = i
    for(int j:0→m+1)  dp[0][j] = j

for(int i=1; i≤N; i++){
    for(int j=1; j≤M; j++){
        if(s1[i-1]==s2[j-1])
            dp(i,j) = dp(i-1,j-1)
        else{
            ins = dp[i,j-1]
            rep = dp[i-1,j-1]
            del = dp[i-1,j]

            dp[i,j] = min(ins, rep, del)
        }
    }
}
```

(*)

K = 13

21	29	29	29
21	5	5	5
13	21	21	21
6	8	21	20

(0,0) (N-1, M-1)

src → dest

find unique paths from src to dest where XOR = K

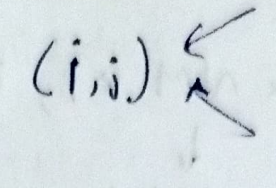
(i,j) → (i+1, j)
→ (i, j+1)

```
f(i, j, mat, xor, n, m, k){
    if(i>=n || j>=m) ret 0;
    if(i==n-1 || j==m-1) ret (xor^mat(i,i)==k);
    x = f(i+1,j, mat, xor^mat(i,i), n, m, k);
    y = f(i, i+1, mat, xor^mat(i,i), n, m, k);
    ret x+y;
}
```

// memoization.

```
f(i,j, mat, xor, n, m, k, int[][][] dp){
    if(i>=n || j>=m) ret 0;
    if(i==n-1 && j==m-1) ret (xor^mat[i,j]==k);
    if(dp[i][j][xor] != -1) ret dp[i][j][xor];
    x = f(i+1, j, mat, xor^mat(i,j), n,m,k, dp);
    y = f(i, j+1, mat, xor^mat(i,j), n,m,k, dp);
    ret dp[i][j][xor] = (x+y);
}
```

// Tabulation.




```
int dp[N+1][M+1][120] = {0}
    dp[1][1][mat[0][0]] = {1}
for(int i=1; i≤N; i++){

  for(int j=1; j≤M; j++){

    for(int curr=0; curr≤80; curr++){

      x = dp[i-1][j][curr ^ mat[i-1][j-1]]

      y = dp[i][j-1][curr ^ mat[i-1][j-1]]

      dp[i+j, curr] = x+y

    }
  }
}
```

mat

10	1	0	1	1	5
1	1	1	1	2	6
1	1	1	5	3	1
0	2	3	4	3	2
0	1	1	0	1	0
1	1	1	0	0	0

K = 20 (students)

1 (instructor)

Matrix is the capacities of NXM labs. There are K students and 1 instructor. We have to place them such that distance between farthest student and instructor should be minimum.

1. $((NM)^2 \times NM \times 1, 1)$

to generate sub matrices — to check sum capability

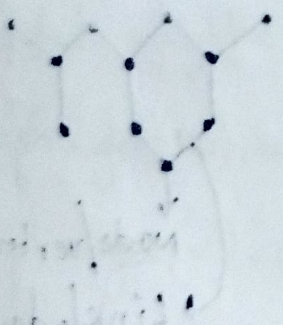
2. BS + Dp

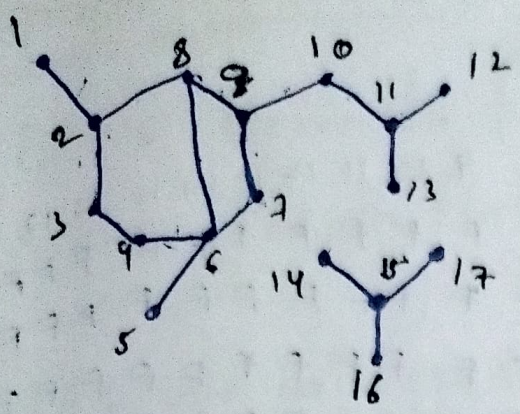
$N \times M + \log_2(\text{high} - \text{low} + 1) \times N \times M, N \times M$

($N \times M$: sum computation in dp table; $\log_2(\text{high} - \text{low} + 1)$: BS on answer; $N \times M$: validity of answer; $N \times M$: space)

```
low = 0
high = max(N,M)
while(low <= high)
{
    mid = (l+h)/2
    if(valid(mat, n, m, mid, dp)){
        ans = mid;
        h = mid-1;
    } else { l = mid+1; }
}
```

```
valid (mat, n, m, mid, dp){
    for(int i=0; i<n ; i++){
        for(int j=0; j<m; j++){
            x = compute (i,j, mid, dp);
            if (x >= k+1)
                ret True;
    } }
    ret false;
}
```

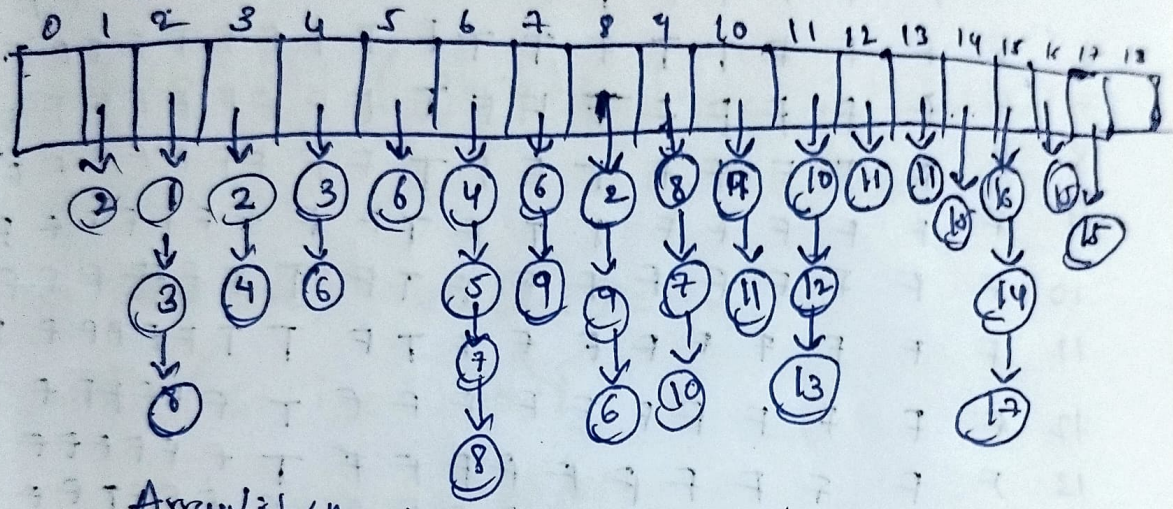




Check if there exists a path from src to dest.

src	dest
2	4
13	16

1. Adjacency List



ArrayList<ArrayList<Integer>> graph ;

Undirected weighted

graph.get(u).add((v,w))

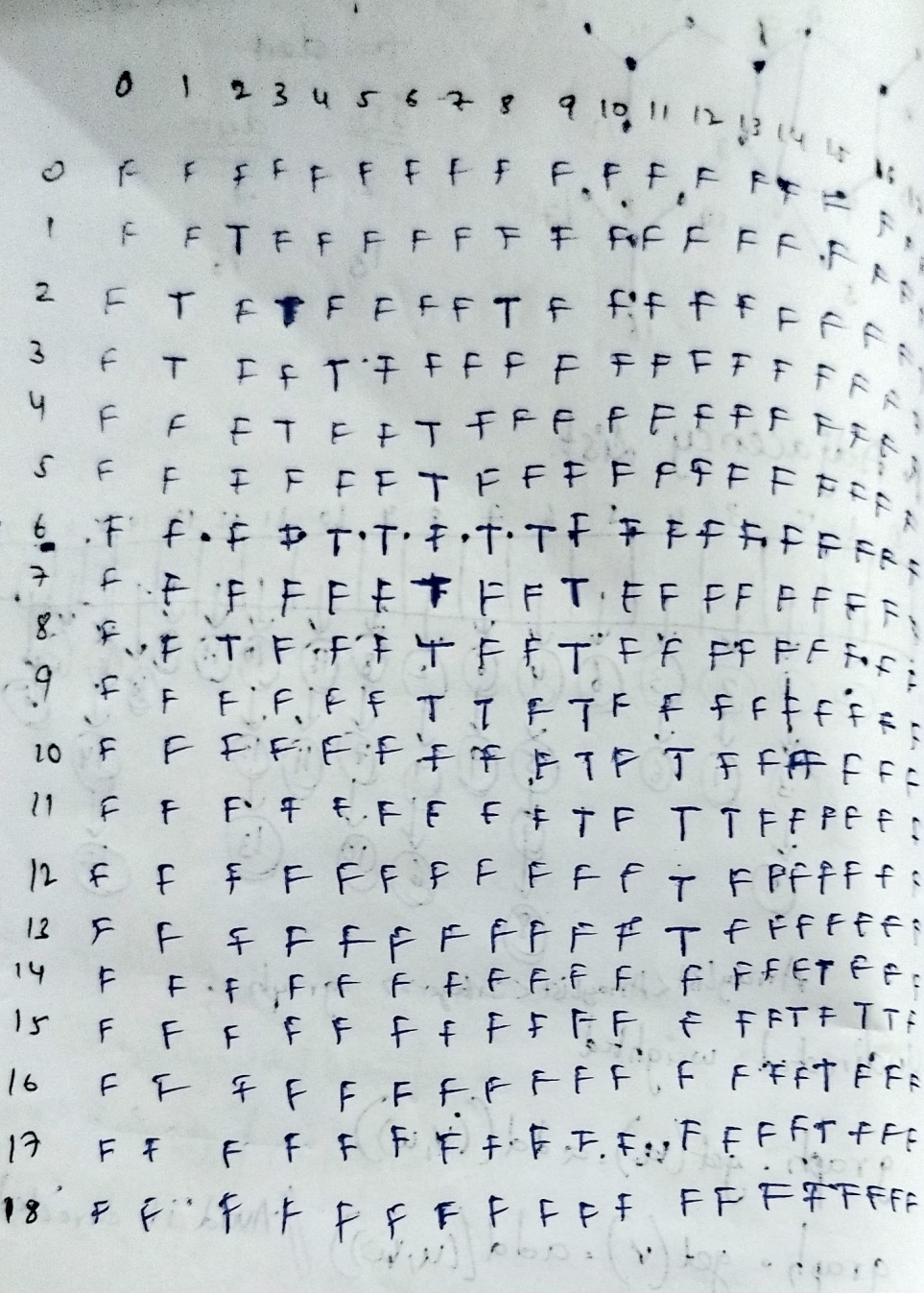
graph.get(v).add((u,w)) // Avoid if directed.

Undirected unweighted

graph.get(u).add(v)

graph.get(v).add(u) // Avoid if directed

Adjacency matrix



ist
- Space efficient (Adv)
- Checking an edge existence is an expensive operation (Dis Adv)

trix
- Checking an edge is O(1) (Adv)
- Space is too much ((N+1) x (N×1)) (Dis Adv)

Graph Traversal

BFS :- Breadth first search

DFS - Depth first search

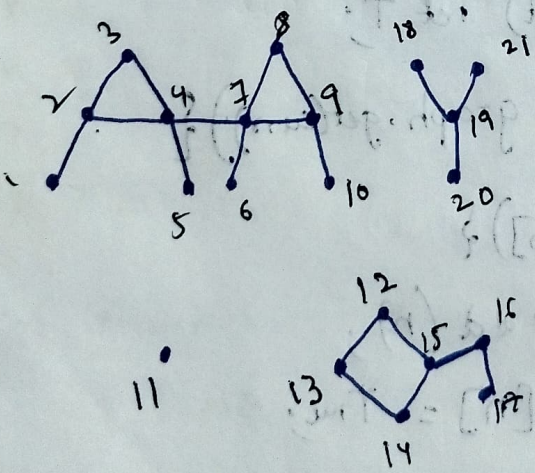
BFS : We will go to all neighbours of a node. (We can use a queue for it)

```
boolean bfs (graph, int src, int dest){
    queue<int> q;                    // boolean vis[]
    q.add(src); vis[src] = True;
    while (q.size()>0){
        int curr = q.poll();
        if(curr == dest) ret T;
        for (int n: graph.get(curr)) {
            if(!vis[n]){
                q.add(n);
                vis[n] = True;
            }
        }
    }
    return false;
}
```

DFS: for a node we will explore all possible nodes of each neighbour

```
boolean dfs(graph, int src, int dest, boolean[] vis){
if(src==dest)ret T;
vis[src]=T;
for(int ele : graph[src]){
  if(!vis[ele] && dfs(graph, ele, dest, vis))
    ret T;
}
ret f;
}
```

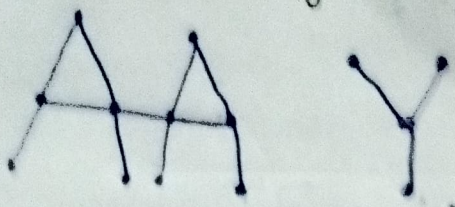
(**) Given a graph, find # connected components.



Undirected
Unweighted
Simple
Weakly connected

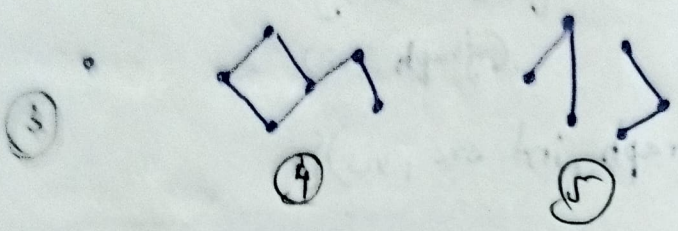
```
boolean vis [N+1] = {F}
int cnt = 0;
N
∀ if(! vis[i]) [cnt++ bfs or dfs (graph, i, vis)]
i=1

void bfs( graph, int src, vis){
    vis[src] = T;
    queue> q;
    q.add(src);
    while (q.size()>0){
        int curr = q.poll();
        for(int n : graph.get(curr)){
            if(!vis[n]) {
                vis[n] = T;
                q.add(n)
            }
    } }
}
void dfs (graph, int src, vis){
    vis[src] = T;
    for(int n : graph [src]){
        if (! vis[n])
            dfs (graph, n, vis);
} }
```

Check which graph is also a Tree?



N → No. of nodes

M → No. of edges



Properties of a graph which is Tree

1. Acyclic ($M == N-1$)
2. Connected components (cc) $= 1$

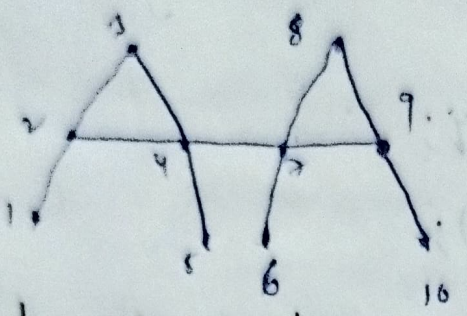
if $M == N - cc$ (it's a tree)

Find the len of shortest path from src to dest.

```
void short(graph, src, dist[])
{
    queue<int> q;
    q.add(src);
    dist[src] = 0;
    while (q.size()>0){
        int curr = q.poll();
        for(int n: graph.get(curr)){
            if(dist[n] == -1){
                q.add(n);
                dist[n] = dist[curr]+1;
            }
        }
    }
}
```

1. Path in a graph
2. No. of connected components
3. Check forest
4. Shortest path [Unweighted]
5. 1D 2D 3D DP SI primary.

10/4/25



Longest path length?

Solutions

1) N^2, N

for every node find distance to all other nodes.

2) N+N, N

From any random node, if we do bfs the farthest node is a src or dest node. From src or dest node we will find longest path.

MAIN

```
int src = bfs(graph, n, dis, true);
int any = bfs(graph, n, dis, false);
```

MAIN

```
int src = bfs(graph, n, 1, true);
int ans = bfs(graph, n, src, false);

int bfs(... graph, int n, int src, bool flag){
    int dist[n+1] = {-1};
    dist[src] = 0;
    int curr = 0;
    Queue<int> q;
    q.add(src);
    while(q.size() > 0){
        curr = q.poll();
        for(int n: graph.get(curr)){
            if(dist[n] == -1){
                q.add(n);
                dist[n] = dist[curr] + 1;
            }
        }
    }
    if(flag) return curr;
    return dist[curr];
```

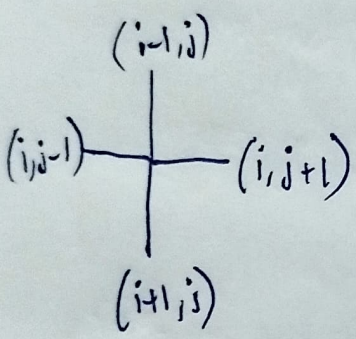
1	0	0	1	1	
1	1	1	1	1	
0	1	0	0	0	
0	0	1	1	0	
1	1	0	0	1	

0 → water

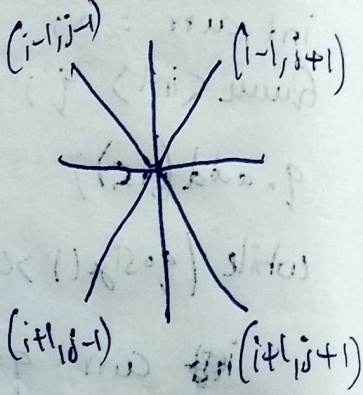
1 → land

find no of # islands

N-4 connectivity



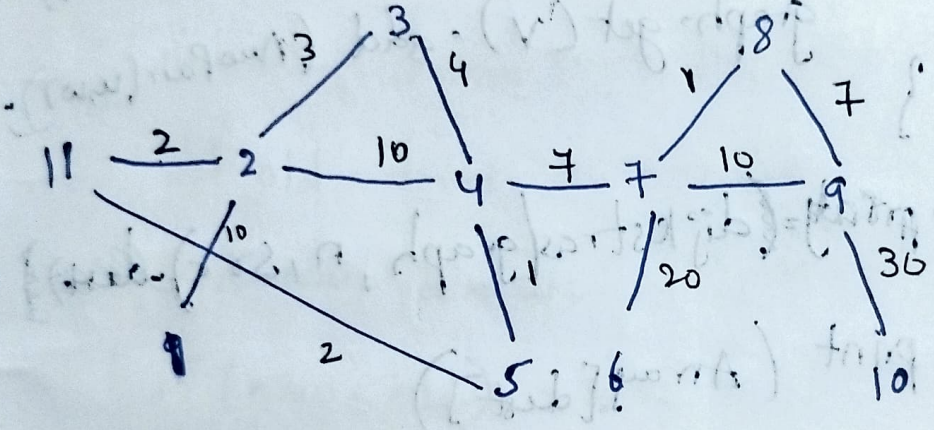
N-8 connectivity




```
void dfs(mat, i, j, n, m) {
if(i<0||i>=n||j<0||j>=m||mat(i,j)==0)ret;
    mat(i,j) = 0;
    dfs(mat, i+1, j, n, m);
    dfs(mat, i, j-1, n, m);
    dfs(mat, i-1, j, n, m);
    dfs(mat, i, j+1, n, m);
}
```

```
void dfs (mat, i, j, n, m) {
 int dx = {1, -1, 0, 0} -1, 1, 1, -1, }
 int dy = { 0, 0, 1, -1 } 1, 1, -1, -1 }
 if (i<0||i>n||j>m||j<0||mat(i,j)==0)
   mat(i,j) = 0;
 for (int k=0; k<4 or 8; k++)
     dfs (mat, i+dx[k], j+dy[k], n, m);
}
```

(*)



Find the min weighted path from 2 to 9. (Dijkstras)

1. Greedy BFS.
2. Source to every other node. (min weighted path)

Read N, M;

Arraylist<Arraylist<Pair>> graph = new Arraylist<>();

N
∀ graph.add(new Arraylist<>());
i=0

loop M {

Read u, v, w;

graph.get(u).add(new Pair(v, w));

graph.get(v).add(new Pair(u, w));

}

List<> array = dijkstras(graph, n, src);

print(Array[dist])

List<Integer> dijkstras(graph, int n, int src){

PriorityQueue<Pair> pq = new PQ<>((p1, p2)

→ p1.first - p2.first)

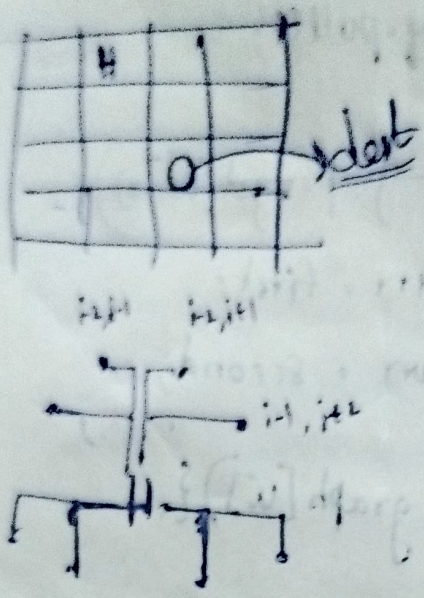
Arraylist<Integer> dist = new List<>(n+1);

dist.fill(I.MAX), dist[src] = 0;

pq.add(new pair(0, src));

```
while (pq.size()>0){

    Pair curr = pq.poll();

    for(Pair e : graph.get(curr).
    int d = curr.first;
    int u = curr.second;

    for (Pair n : graph[u]){

        int v = n.first;
        int w = n.second;
        int old = dist[v];
        int new= d+w;
        if (new<old)
        {
            pq.add((new,v));
            dist[v] = new;
        }
    }
}
return dist;
}
```

Knight on a chess Board



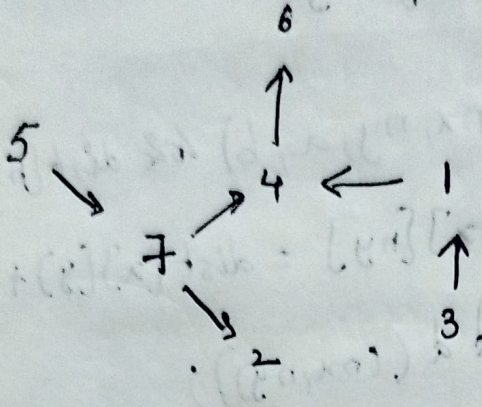

```
int solve (int a, int b, int c, int d, int e, int f) {
    int dx = {
    int dy = {
    int[][] dist = new [a+1][b+1]
        dist = {-1};
    dist[c][d] = 0;
    q.add ((c,d));
    while (q.size() > 0) {
        Pair curr = q.poll;
        int x = curr.first;
        int y = curr.second;
```

24/04/25

Topological Sort

Kahn's Algorithm

O(n log n)



Paths to solve the tasks

5 7 2 3 1 4 6

3 1 5 7 2 4 6

↳ (lexicographically shortest path)

We can directly cmplt a task if it is independent. If it dependent, first we hv to solve ~~dep~~ those ~~tasks~~ to cmplt the dependent task

(eg: 4 can be solved only if 1,7 are solved)

So the path we choose shld be like take the smallest number from possible moves we have.

3 is the least independent task.

Then possible tasks to be solved are 1, 5, so we chose 1.

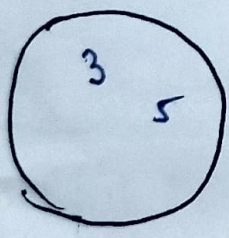
Then only possible task to be cmplted is 5, then 7.

After 7 we hv 4, 2 as possible tasks so we chose 2 then 4 and 6.

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

(indeg arr)

min heap



add elements with indeg 0

getMin → 3 (neigh : 1)

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

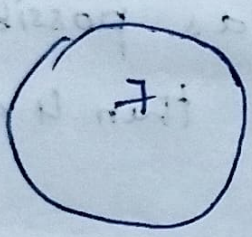
getMin → 1

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

None of the ~~nodes~~ nodes indeg is 0

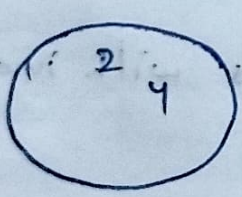
getMin → 5

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



getMin → 7

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



getMin → 2

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

getMin → 4

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

getMin → 6.

```
main(){
  int ind[N+1] = {0};

  loop(M){
      Read u, v;
      graph.get(u).add(v);
      ind[v]++;
  }
  Arraylist<Integer> ans = kahns(graph, N, ind)
}

Arraylist<Integ> kahns(graph, N, ind){

  PriorityQueue<Integ> pq
          = new PriorityQueue<>((p1, p2) ->
Arraylist<Integer> res = new Arraylist<>();   (p1 - p2));
  for(int i: ind){
    if(i==0) pq.add(i);
  }
while(pq.size()>0){
  int x = pq.getMin();
  res.add(x);
  ind[x] = -1;
  for (int i: graph.get(x)) {
      ind[i]--;
      if (ind[i]==0) pq.add(i);
  }
}
return res;
}
```

NlogN

```
for(int k=0; k<8; k++)
    int nx = x + dx[k]
    int ny = y + dy[k]
    if(valid(nx, ny, a, b) && dist[nx][ny]==-1)
        dist[nx][ny] = dist[x][y] + 1;
        q.add((nx,ny));
}}
ret dist[e][f];
}
```

Lab Agenda

1. Longest path in a graph
2. No. of Islands
3. Shortest path (weighted)
4. Knight in a chessboard [unweighted]
5. " [weighted].

03/4/2025

- N Floors
  K eggs
  There exists a floor f till where the eggs doesn't break
  (if f=4 from 0 to 4 egg doesn't break)
  Find min egg drops required to find f.
  (DP + BS)

GRAPHS

1. Undirected / Directed
2. Unweighted / Weighted
3. Simple / complex
4. Cyclic / Acyclic
5. Strongly / weakly
   ↓
   From any node we can reach all other nodes