**Q1. Path in Directed Graph**

Given an directed graph having **A** nodes labelled from **1** to **A** containing **M** edges given by matrix **B** of size M x 2such that there is a edge directed from node

**B[i][0]** to node **B[i][1]**.

Find whether a path exists from node **1** to node **A**.

Return **1** if path exists else return **0**.

**Logic -**

1. Either Breadth First Search (BFS) or Depth First Search (DFS) can be used to find path between two vertices.
2. Take the first vertex as source in BFS (or DFS), follow the standard BFS (or DFS). If the second vertex is found in our traversal, then return 1 else return 0.

**Trade-offs between BFS and DFS:** Breadth-First search can be useful to find the shortest path between nodes, and depth-first search may traverse one adjacent node very deeply before ever going into immediate neighbours.

**Code -**

from collections import deque

class Solution:

# @param A : integer

# @param B : list of list of integers

# @return an integer

def solve(self, A, B):

graph = [[] for i in range(A+1)]

# Create Graph

for i in range(len(B)):

u = B[i][0]

v = B[i][1]

graph[u].append(v)

# graph[v].append(u)

visited = [0 for i in range(A+1)]

q = deque()

q.append(1)

while(len(q) != 0):

x = q.popleft()

visited[x] = 1

for j in range(len(graph[x])):

if(visited[graph[x][j]] == 0): q.append(graph[x][j])

return visited[A]

**Q2. First Depth First Search**  
You are given N towns (1 to N). All towns are connected via unique directed path as mentioned in the input.

Given 2 towns find whether you can reach the first town from the second without repeating any edge.

**B C** : query to find whether B is reachable from C.

Input contains an integer array **A** of size N and 2 integers B and C ( 1 <= B, C <= N ).

There exist a directed edge from A[i] to i+1 for every 1 <= i < N. Also, it's guaranteed that A[i] <= i for every 1 <= i < N.

**NOTE:** Array A is 0-indexed. A[0] = 1 which you can ignore as it doesn't represent any edge.

**Logic – Both methods will work (BFS and DFS)**

**Code -**

from collections import deque

class Solution:

def dfs(self, graph, visited, C):

visited[C] = 1

for i in range(len(graph[C])):

temp = graph[C][i]

if (visited[temp] == 0): self.dfs(graph, visited, temp)

def bfs(self, graph, visited, C):

q = deque()

q.append(C)

while(len(q) != 0):

x = q.popleft()

visited[x] = 1

for j in range(len(graph[x])):

if(visited[graph[x][j]] == 0): q.append(graph[x][j])

def solve(self, A, B, C):

graph = [[] for i in range(len(A)+1)]

visited = [0 for i in range(len(A)+1)]

for i in range(1, len(A)):

graph[A[i]].append(i+1)

self.bfs(graph, visited, C)

return visited[B]

**Q3. Number of islands**

Given a matrix of integers **A** of size **N x M** consisting of **0** and **1**. A group of connected **1's** forms an island. From a cell **(i, j)** such that **A[i][j] = 1** you can visit any cell that shares a corner with (i, j) and value in that cell is 1.

More formally, from any cell (i, j) if A[i][j] = 1 you can visit:

* **(i-1, j)** if (i-1, j) is inside the matrix and A[i-1][j] = 1.
* **(i, j-1)** if (i, j-1) is inside the matrix and A[i][j-1] = 1.
* **(i+1, j)** if (i+1, j) is inside the matrix and A[i+1][j] = 1.
* **(i, j+1)** if (i, j+1) is inside the matrix and A[i][j+1] = 1.
* **(i-1, j-1)** if (i-1, j-1) is inside the matrix and A[i-1][j-1] = 1.
* **(i+1, j+1)** if (i+1, j+1) is inside the matrix and A[i+1][j+1] = 1.
* **(i-1, j+1)** if (i-1, j+1) is inside the matrix and A[i-1][j+1] = 1.
* **(i+1, j-1)** if (i+1, j-1) is inside the matrix and A[i+1][j-1] = 1.

Return the number of islands.

**Logic – DFS is used here, and number of traversals gives the number of islands**

1. Whenever a cell with unvisited value ‘1’ is encountered we explore all the nodes that are reachable from it and continue exploring until no more nodes are left to explore.
2. While exploring we mark them visited (A[i][j] = 0) so that no nodes can be explored twice.
3. After completion of traversal increament the count of islands.
4. Find for the 1 which is not visited yet.

**Code -**

import sys

sys.setrecursionlimit(10000)

class Solution:

# @param A : list of list of integers

# @return an integer

dx = [-1, -1, -1, 0, 0, 1, 1, 1]

dy = [-1, 0, 1, -1, 1, -1, 0, 1]

def dfs(self, A, i, j):

if(i==-1 or j==-1): return

if(i==len(A) or j==len(A[0])): return

if(A[i][j] == 0): return

A[i][j] = 0

for x in range(8):

self.dfs(A, i+self.dx[x], j+self.dy[x])

def solve(self, A):

cnt = 0

for i in range(len(A)):

for j in range(len(A[0])):

if(A[i][j] == 0): continue

self.dfs(A, i, j)

cnt += 1

return cnt

**Q4. Rotten Oranges**

Given a matrix of integers **A** of size N x M consisting of 0, 1 or 2.

Each cell can have three values:

The value 0 representing an empty cell.

The value 1 representing a fresh orange.

The value 2 representing a rotten orange.

Every minute, any fresh orange that is adjacent (Left, Right, Top, or Bottom) to a rotten orange becomes rotten. Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1 instead.

**Logic – Use BFS, and mark the visited cells as 2**

1. Every turn, the rotting spreads from each rotting orange to other adjacent oranges.
2. Initially, the rotten oranges have ‘depth’ 0, and every time they rot a neighbor,  
   the neighbors have 1 more depth. We want to know the largest possible depth.
3. Use multi-source BFS to achieve this with all cells containing rotten oranges as starting nodes.
4. At the end check if there are fresh oranges left or not.

**Code -**

public class Solution {

public class Node {

int i;

int j;

Node(int x, int y) {

i = x;

j = y;

}

}

public int solve(ArrayList<ArrayList<Integer>> A) {

int n = A.size();

int m = A.get(0).size();

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

for(int i=0;i<A.size();i++) {

for(int j=0;j<A.get(i).size();j++) {

if(A.get(i).get(j) == 2) {

q.add(new Node(i, j));

}

}

}

q.add(null);

int cnt = 0;

while(q.size()>1){

Node x = q.remove();

if(x==null) {

cnt+=1;

q.add(null);

continue;

}

int i = x.i;

int j = x.j;

if(i<n-1 && A.get(i+1).get(j) == 1) {

q.add(new Node(i+1, j));

A.get(i+1).set(j, 2);

}

if(i>0 && A.get(i-1).get(j) == 1) {

q.add(new Node(i-1, j));

A.get(i-1).set(j, 2);

}

if(j<m-1 && A.get(i).get(j+1) == 1) {

q.add(new Node(i, j+1));

A.get(i).set(j+1, 2);

}

if(j>0 && A.get(i).get(j-1) == 1) {

q.add(new Node(i, j-1));

A.get(i).set(j-1, 2);

}

}

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

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

if(A.get(i).get(j) == 1) {

return -1;

}

}

}

return cnt;

}

}

**Q5. Coloring a Cycle Graph**

Given the number of vertices **A** in a Cyclic Graph.

Your task is to determine the **minimum number of colors** required to color the graph so that **no two Adjacent vertices** have the same color.

**Logic -**

1. If the no. of vertices is Even then it is Even Cycle and to color such graph we require 2 colors.
2. If the no. of vertices is Odd then it is Odd Cycle and to color such graph we require 3 colors.

**Code -**

public class Solution {

public int solve(int A) {

return A%2==0 ? 2 : 3;

}

}

**Q6 Check Bipartite Graph**

**Logic -** You can use and approach either BFS or DFS to check whether the graph can be colored using two colors or not.

1. Start from any node that hase not been colored yet:  
   a. Assign color1 to this nodes  
   b. check its adjacent nodes  
   a. if this is colored in color1 then the graph can’t be bipartite ,return 0.  
   b. else if this is colored in color1 do nothing.  
   c. else color it with color 2 and push it into queue.
2. Repet step1 until no nodes is left for coloring.

**Code -**

public class Solution {

public int solve(int A, ArrayList<ArrayList<Integer>> B) {

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

int [] visited = new int[A];

Arrays.fill(visited, -1);

// Create graph

for(int i=0;i<A;i++) graph.add(new ArrayList<Integer>());

for(int i=0;i<B.size();i++) {

int u = B.get(i).get(0);

int v = B.get(i).get(1);

graph.get(u).add(v);

graph.get(v).add(u);

}

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

// Applying BFS on all non visited nodes

for(int x=0;x<A;x++){

if(visited[x] != -1) continue;

q.add(x);

visited[x] = 0;

while(q.size()>0) {

int u = q.remove();

for(int i=0;i<graph.get(u).size();i++) {

int temp = graph.get(u).get(i);

if(visited[temp] == -1) {

visited[temp] = 1-visited[u];

q.add(temp);

}

// If parent color == child color return 0

if(visited[temp] == visited[u]) return 0;

}

}

}

return 1;

}

}

**Q7. Construct Roads**

A country consist of **N** cities connected by N - 1 roads. King of that country want to construct **maximum** number of roads such that the new country formed remains **bipartite country**.

**Bipartite country** is a country, whose cities can be partitioned into 2 sets in such a way, that for each road (u, v) that belongs to the country, u and v belong to different sets. Also, there should be no multiple roads between two cities and no self loops.

Return the maximum number of roads king can construct. Since the answer could be large return **answer % 109 + 7**.

**NOTE:** All cities can be visited from any city.

**Logic –** Count nodes in alternate levels (ans = x1\*y1 - (n-1))

**Code -**

public class Solution {

// Count nodes in alternate levels

public int solve(int A, ArrayList<ArrayList<Integer>> B) {

int[] visited = new int[A+1];

Arrays.fill(visited, -1);

int mod = (1000\*1000\*1000+7);

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

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

graph.add(new ArrayList<Integer>());

}

for(int i=0;i<B.size();i++) {

int u = B.get(i).get(0);

int v = B.get(i).get(1);

graph.get(u).add(v);

graph.get(v).add(u);

}

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

q.add(1);

q.add(-1);

visited[1] = 1;

long x1 = 0;

long y1 = 0;

boolean flag = true;

while(q.size()>1) {

int x = q.remove();

if(x==-1) {

q.add(-1);

flag = !flag;

continue;

}

visited[x] = 1;

if(flag) x1+=1;

else y1+=1;

for(int i=0;i<graph.get(x).size();i++) {

int temp = graph.get(x).get(i);

if(visited[temp] == -1) {

q.add(temp);

}

}

}

return (int)(x1%mod \* y1%mod - A+1) % mod;

}

}

**Q8. Cycle in Undirected Graph**

Given an undirected graph having **A** nodes labelled from **1** to **A** with **M** edges given in a form of matrix **B** of size M x 2 where (B[i][0], B[i][1]) represents two nodes B[i][0] and B[i][1] connected by an edge.

Find whether the graph contains a cycle or not, return **1** if cycle is present else return **0**.

**Logic –**

1. Like directed graphs, we can use DFS to detect cycle in an undirected graph in O(A+M) time.
2. We do a DFS traversal of the given graph. For every visited vertex ‘v’, if there is an adjacent ‘u’ such that u is already visited and u is not parent of v, then there is a cycle in graph.
3. If we don’t find such an adjacent for any vertex, we say that there is no cycle.
4. The assumption of this approach is that there are no parallel edges between any two vertices

**Code –**

public class Solution {

public boolean dfs(ArrayList<ArrayList<Integer>> graph, boolean[] visited, int parent, int v) {

visited[v] = true;

for(int i=0;i<graph.get(v).size();i++) {

int temp = graph.get(v).get(i);

if(!visited[temp]) {

if (dfs(graph, visited, v, temp)) return true;

} else {

if(temp != parent) return true;

}

}

return false;

}

public int solve(int A, ArrayList<ArrayList<Integer>> B) {

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

boolean[] visited = new boolean[A+1];

for(int i=0;i<A+1;i++) {

graph.add(new ArrayList<Integer>());

}

for(int i=0;i<B.size();i++) {

int u = B.get(i).get(0);

int v = B.get(i).get(1);

graph.get(u).add(v);

graph.get(v).add(u);

}

for(int i=1;i<=A;i++) {

if(!visited[i]) {

if(dfs(graph, visited, -1, i)) return 1;

}

}

return 0;

}

}

**Q9. Valid Path**

There is a rectangle with left bottom as (0, 0) and right up as (x, y).

There are **N** circles such that their centers are inside the rectangle.

Radius of each circle is **R**. Now we need to find out if it is possible that we can move from (0, 0) to (x, y) without touching any circle.

**Note** : We can move from any cell to any of its 8 adjecent neighbours and we cannot move outside the boundary of the rectangle at any point of time.

**Logic –**

1. Create a matrix of B+1 rows and A+1 columns indicating all the points of the graph.
2. Check for every point if it lies in any of the circle. If yes then mark it as zero else mark it as 1.
3. Apply DFS/BFS on the adjancency matrix and check if the (A, B) point is visited.

**Code –**

public class Solution {

// Apply DFS on the Adjancency matrix

public boolean dfs(int[][] graph, int i, int j) {

int n = graph.length;

int m = graph[0].length;

if(i<=-1 || j<=-1 || i>=n || j>=m) return false;

if(graph[i][j] == 0) return false;

if(i==n-1 && j==m-1) return true;

graph[i][j] = 0;

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

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

for(int p=0;p<8;p++) {

if (dfs(graph, i+tempx[p], j+tempy[p])) return true;

}

return false;

}

public String solve(int A, int B, int C, int D, ArrayList<Integer> E, ArrayList<Integer> F) {

int[][] graph = new int[B+1][A+1];

for(int i=0;i<=B;i++) Arrays.fill(graph[i], 1);

// Create graph

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

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

for(int k=0;k<C;k++) {

int x = E.get(k);

int y = F.get(k);

int ans = 0;

ans += ((x-j)\*(x-j));

ans += ((y-i)\*(y-i));

if(ans <= (D\*D)) {

graph[i][j] = 0;

break;

}

}

}

}

return dfs(graph, 0, 0) ? "YES" : "NO";

}

}

**Q10. Distance of nearest cell**

Given a matrix of integers **A** of size **N x M** consisting of **0** or **1**.

For each cell of the matrix find the distance of nearest 1 in the matrix.

Distance between two cells **(x1, y1)** and **(x2, y2)** is defined as **|x1 - x2| + |y1 - y2|**.

Find and return a matrix **B** of size **N x M** which defines for each cell in A distance of nearest **1** in the matrix A.

**NOTE:** There is atleast one 1 is present in the matrix.

**Logic –**

1. The idea is to use multi-source BFS. At the begining insert all the nodes having value 1 in the queue.
2. We first explore immediate adjacent of all 1’s, then adjacent of adjacent, and so on.
3. Only if the distance at the cell of matrix is greater than the current distance, then only we update the distance of the cell.
4. Therefore we find minimum distance.
5. Time Complexity: O( N x M)

**Code –**

public class Solution {

public class Node {

int i;

int j;

public Node(int x, int y) {

i = x;

j = y;

}

}

public ArrayList<ArrayList<Integer>> solve(ArrayList<ArrayList<Integer>> A) {

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

Queue<Node> qu = new LinkedList<Node>();

for(int i=0;i<A.size();i++) {

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

for(int j=0;j<A.get(0).size();j++) {

temp.add(0);

}

ans.add(temp);

}

for(int i=0;i<A.size();i++) {

for(int j=0;j<A.get(0).size();j++) {

if(A.get(i).get(j) == 1) qu.add(new Node(i, j));

}

}

qu.add(null);

int cnt = 1;

while(qu.size() > 1) {

Node x = qu.remove();

if(x == null) {

cnt += 1;

qu.add(null);

continue;

}

int p = x.i;

int q = x.j;

if(p>0 && A.get(p-1).get(q) != 1) {

qu.add(new Node(p-1, q));

ans.get(p-1).set(q, cnt);

A.get(p-1).set(q, 1);

}

if(q>0 && A.get(p).get(q-1) != 1) {

qu.add(new Node(p, q-1));

ans.get(p).set(q-1, cnt);

A.get(p).set(q-1, 1);

}

if(p<A.size()-1 && A.get(p+1).get(q) != 1) {

qu.add(new Node(p+1, q));

ans.get(p+1).set(q, cnt);

A.get(p+1).set(q, 1);

}

if(q<A.get(0).size()-1 && A.get(p).get(q+1) != 1) {

qu.add(new Node(p, q+1));

ans.get(p).set(q+1, cnt);

A.get(p).set(q+1, 1);

}

}

return ans;

}

}

**Q 11 Possibility of Finishing**

There are a total of **A** courses you have to take, labeled from **1** to **A**.

Some courses may have prerequisites, for example to take course **2** you have to first take course **1**, which is expressed as a pair: **[1,2]**.

So you are given two integer array **B** and **C** of same size where for each i **(B[i], C[i])** denotes a pair.

Given the total number of courses and a list of prerequisite pairs, is it possible for you to finish all courses?

Return **1** if it is **possible** to finish all the courses, or **0** if it is **not possible** to finish all the courses.

**Logic -** The problem reduces down to finding a directed cycle in the whole graph. If any such cycle is present, it is not possible to finish all the courses.

1. Method 1 is to find a cycle is using DFS, and comparing child nodes with parent nodes.
2. Method 2 is use topological sort and check if all nodes are visited on not by using a variable cnt. Steps for topological sort
   1. Create an indegree array, in which number of incoming nodes for every node will be present.
   2. Add the nodes with 0 incoming nodes in a queue.
   3. Apply BFS and reduce the incoming nodes as each course is completed and also keep adding the nodes with 0 indegree in the queue

**Code -**

public class Solution {

public int solve(int A, int[] B, int[] C) {

int[] indeg = new int[A+1];

Arrays.fill(indeg, 0);

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

for(int i=0;i<=A;i++) graph.add(new ArrayList<Integer>());

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

int u = B[i];

int v = C[i];

graph.get(u).add(v);

indeg[v] += 1;

}

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

int cnt = 0;

for(int i=1;i<=A;i++) {

if(indeg[i] == 0) q.add(i);

}

while(q.size() > 0) {

int curr = q.remove();

cnt += 1;

for(int i=0;i<graph.get(curr).size();i++) {

int child = graph.get(curr).get(i);

indeg[child] -= 1;

if(indeg[child] == 0) q.add(child);

}

}

// System.out.print(cnt);

return cnt==A ? 1 : 0;

}

}

**Q12. Dijsktra**

Given a weighted undirected graph having A nodes and M weighted edges, and a source node C.

You have to find an integer array D of size A such that:

=> D[i] : Shortest distance form the C node to node i.

=> If node i is not reachable from C then -1.

Note:

There are no self-loops in the graph.

No multiple edges between two pair of vertices.

The graph may or may not be connected.

Nodes are numbered from 0 to A-1.

Your solution will run on multiple testcases. If you are using global variables make sure to clear them.

**Logic -**

1. Initialize a distance array of integers(denoting distance of source to node i) with infinity.
2. Initialize d[source]=0 (distance from source to source is 0).  
   Insert {d[source],source} into a min heap based on distance.
3. extract first element from the heap:
   1. if this element is mark visited then again start extract top element from heap else mark this as vis and expore adjacent nodes of the top node of the heap.
   2. if distance[adjacentnode]>distance[curr]+weight of the edge between curr and adjacent node update distacne[adjacentnode] = distance[curr]+weight of the edge between curr and adjacent node
   3. insert this node alongwith weight into minheap.

**Code -**

public class Solution {

public class Node {

int value;

int weight;

Node(int val, int wt) {

value = val;

weight = wt;

}

}

public class myCMP implements Comparator <Node>{

public int compare(Node a, Node b) {

return a.weight - b.weight;

}

}

public ArrayList<Integer> solve(int A, ArrayList<ArrayList<Integer>> B, int C) {

int[] dist = new int[A];

Arrays.fill(dist, Integer.MAX\_VALUE);

dist[C] = 0;

ArrayList<ArrayList<Node>> graph = new ArrayList<ArrayList<Node>>();

for(int i=0;i<=A;i++) graph.add(new ArrayList<Node>());

for(int i=0;i<B.size();i++) {

int u = B.get(i).get(0);

int v = B.get(i).get(1);

int wt = B.get(i).get(2);

graph.get(u).add(new Node(v, wt));

graph.get(v).add(new Node(u, wt));

}

PriorityQueue<Node> pq = new PriorityQueue<Node>(new myCMP());

pq.add(new Node(C, 0));

while(pq.size() > 0) {

Node curr = pq.poll();

if(curr.weight != dist[curr.value]) continue;

for(int i=0;i<graph.get(curr.value).size();i++) {

Node child = graph.get(curr.value).get(i);

if(dist[child.value] > dist[curr.value] + child.weight) {

dist[child.value] = dist[curr.value] + child.weight;

pq.add(new Node(child.value, dist[child.value]));

}

}

}

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

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

if(dist[i] == Integer.MAX\_VALUE) ans.add(-1);

else ans.add(dist[i]);

}

return ans;

}

}

**Q13. Clone Graph**

Clone an undirected graph. Each node in the graph contains a label and a list of its neighbors.

**Note:** The test cases are generated in the following format (use the following format to use **See Expected Output** option):  
First integer N is the number of nodes.  
Then, N intgers follow denoting the label (or hash) of the N nodes.  
Then, N2 integers following denoting the adjacency matrix of a graph, where **Adj[i][j] = 1** denotes presence of an undirected edge between the ith and jth node, **O** otherwise.

**Logic –** Use DFS/BFS and keep a Hashmap to check if a node is already created or not.

1. If Node is null return null
2. Check in the hashmap if already clone is created or not. If yes return that clone node.
3. If not created, create a new node and add it to the hashmap. Later traverse the neighbours of the node and recursively call the getClone function to get the clone nodes of the neighbours.

**Code -**

/\*\*

\* Definition for undirected graph.

\* class UndirectedGraphNode {

\* int label;

\* List<UndirectedGraphNode> neighbors;

\* UndirectedGraphNode(int x) { label = x; neighbors = new ArrayList<UndirectedGraphNode>(); }

\* };

\*/

public class Solution {

public UndirectedGraphNode getClone(UndirectedGraphNode node, HashMap<UndirectedGraphNode, UndirectedGraphNode> map) {

if(node == null) return null;

if(map.containsKey(node)) return map.get(node);

UndirectedGraphNode clone = new UndirectedGraphNode(node.label);

map.put(node, clone);

for(int i=0;i<node.neighbors.size();i++) {

UndirectedGraphNode child = node.neighbors.get(i);

clone.neighbors.add(getClone(child, map));

}

return clone;

}

public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {

HashMap<UndirectedGraphNode, UndirectedGraphNode> map = new HashMap<UndirectedGraphNode, UndirectedGraphNode>();

return getClone(node, map);

}

}

**Q14. Batches**

**A** students applied for admission in IB Academy. An array of integers **B** is given representing the strengths of **A** people i.e. **B[i]** represents the strength of **ith** student.

Among the **A** students some of them knew each other. A matrix **C** of size **M x 2** is given which represents relations where ith relations depicts that C[i][0] and C[i][1] knew each other.

All students who know each other are placed in one batch.

Strength of a batch is equal to sum of the strength of all the students in it.

Now the number of batches are formed are very much, it is impossible for IB to handle them. So IB set criteria for selection: All those batches having strength **at least D** are selected.

Find the number of batches selected.

**Logic -**

Modify the above problem in the form of an undirected weighted graph.  
Consider students as nodes and relations as edges between them.  
All connected components come under one batch.  
Strength of a batch is the sum of the weight of nodes of connected components of the graph(batch).

After Modifying the problem statement to graph perspective, It is easy to see find the solution.

Initiaize ans = 0

Pick any unvisited node and find the sum of all the weights of nodes which are reachable from this node and mark all such nodes as visited. if this sum is greater than equal to D then increment ans.

If N is the number of students and M is the number of relations then

**Code -**

public class Solution {

public int dfs(ArrayList<ArrayList<Integer>> graph, ArrayList<Integer> B, int[] visited, int v) {

visited[v] = 1;

int s = B.get(v-1);

for(int i=0;i<graph.get(v).size();i++) {

int child = graph.get(v).get(i);

if(visited[child] == 0) {

s += dfs(graph, B, visited, child);

}

}

return s;

}

public int solve(int A, ArrayList<Integer> B, ArrayList<ArrayList<Integer>> C, int D) {

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

for(int i=0;i<=A;i++) graph.add(new ArrayList<Integer>());

for(int i=0;i<C.size();i++) {

int u = C.get(i).get(0);

int v = C.get(i).get(1);

graph.get(u).add(v);

graph.get(v).add(u);

}

int[] visited = new int[A+1];

Arrays.fill(visited, 0);

int ans = 0;

for(int i=1;i<=A;i++) {

if(visited[i] == 0) if(dfs(graph, B, visited, i) >= D) ans+=1;

}

return ans;

}

}

**Q15. Construction Cost**

Given a graph with **A** nodes and **C** weighted edges. Cost of constructing the graph is the sum of weights of all the edges in the graph.

Find the **minimum cost** of constructing the graph by selecting some given edges such that we can reach every other node from the **1st** node.

**Logic -**

As it can be easily be seen that the graph will not have any cyles and every other node should be rechable from the 1st.

The resulting graph is connected and without cycles. So, it will be a tree.

To minimize the cost, we can find minimum spanning tree using Kruskal or Prim algorithms.

**Code -**

public class Solution {

public class Node {

int a;

int b;

int weight;

public Node(int i, int j, int wt) {

a = i;

b = j;

weight = wt;

}

}

public int findRoot(int i, int[]parent) {

if(i==parent[i]) return i;

return parent[i] = findRoot(parent[i], parent);

}

public boolean union(int A, int B, int[]parent) {

int rootA = findRoot(A, parent);

int rootB = findRoot(B, parent);

if(rootA != rootB) {

parent[rootA] = rootB;

return true;

}

return false;

}

public class mycmp implements Comparator<Node> {

public int compare(Node a, Node b) {

return a.weight - b.weight;

}

}

public int solve(int A, ArrayList<ArrayList<Integer>> B) {

int[] parent = new int[A+1];

for(int i=0;i<=A;i++) parent[i] = i;

ArrayList<Node> graph = new ArrayList<Node>();

for(int i=0;i<B.size();i++) {

graph.add(new Node(B.get(i).get(0), B.get(i).get(1), B.get(i).get(2)));

}

Collections.sort(graph, new mycmp());

long cost = 0;

int mod = (1000\*1000\*1000+7);

for(int i=0;i<graph.size();i++) {

if(union(graph.get(i).a, graph.get(i).b, parent)) {

cost += graph.get(i).weight;

cost %= mod;

}

}

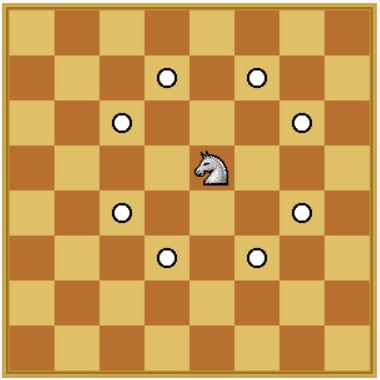
return (int)cost;

}

}

**Q16. Knight On Chess Board**

Given any source point, **(C, D)** and destination point, **(E, F)** on a chess board of size **A x B**, we need to find whether Knight can move to the destination or not.



The above figure details the movements for a knight ( 8 possibilities ).

If yes, then what would be the **minimum** number of steps for the knight to move to the said point. If knight can not move from the source point to the destination point, then return **-1**.

**NOTE:** A knight cannot go out of the board.

**Logic -**

A knight can move to 8 positions from (x,y).   
  
(x, y) ->   
 (x + 2, y + 1)   
 (x + 2, y - 1)  
 (x - 2, y + 1)  
 (x - 2, y - 1)  
 (x + 1, y + 2)  
 (x + 1, y - 2)  
 (x - 1, y + 2)  
 (x - 1, y - 2)  
  
Corresponding to the knight's move, we can define edges.   
(x,y) will have an edge to the 8 neighbors defined above.   
  
To find the shortest path, we just run a plain BFS.

**Code -**

public class Solution {

public class Node {

int x;

int y;

public Node(int a, int b) {

x = a;

y = b;

}

}

public int knight(int A, int B, int C, int D, int E, int F) {

C-=1;

D-=1;

E-=1;

F-=1;

if(E>A || F>B) return -1;

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

q.add(new Node(C, D));

q.add(null);

int[][] graph = new int[A][B];

for(int i=0;i<A;i++) Arrays.fill(graph[i], Integer.MAX\_VALUE);

int cnt = 0;

while(q.size() > 1) {

Node curr = q.remove();

if (curr == null) {

cnt+=1;

q.add(null);

continue;

}

if(graph[curr.x][curr.y] != Integer.MAX\_VALUE) continue;

graph[curr.x][curr.y] = Math.min(graph[curr.x][curr.y], cnt);

int x = curr.x;

int y = curr.y;

if(x-2 >= 0 && y-1 >= 0) q.add(new Node(x-2, y-1));

if(x-2 >= 0 && y+1 < B) q.add(new Node(x-2, y+1));

if(x-1 >= 0 && y-2 >= 0) q.add(new Node(x-1, y-2));

if(x-1 >= 0 && y+2 < B) q.add(new Node(x-1, y+2));

if(x+1 < A && y-2 >= 0) q.add(new Node(x+1, y-2));

if(x+2 < A && y-1 >= 0) q.add(new Node(x+2, y-1));

if(x+2 < A && y+1 < B) q.add(new Node(x+2, y+1));

if(x+1 < A && y+2 < B) q.add(new Node(x+1, y+2));

}

if(graph[E][F] == Integer.MAX\_VALUE) return -1;

return graph[E][F];

}

}

**Q17. Prims Algorithm**

**Logic -**

1. Create a min heap which will give Node with smallest weight first.
2. Create a visited array.
3. Follow below steps till min heap is not empty.
   1. Remove the Node with smallest weight.
   2. If that node is already visited continue to the next node.
   3. Else mark the node as visited and add the weight to the cost.
   4. Traverse the children nodes of the curr Node and add it to the min heap, if it is not visited.

**Code -**

// Prims algorithm

public class Solution {

public class Node {

int val;

int wt;

public Node(int value, int weight) {

val = value;

wt = weight;

}

}

public class myCMP implements Comparator<Node> {

public int compare(Node a, Node b) {

return a.wt-b.wt;

}

}

public int solve(int A, int[][] B) {

ArrayList<ArrayList<Node>> graph = new ArrayList<ArrayList<Node>>();

for(int i=0;i<=A;i++) graph.add(new ArrayList<Node>());

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

int u = B[i][0];

int v = B[i][1];

int wt = B[i][2];

graph.get(u).add(new Node(v, wt));

graph.get(v).add(new Node(u, wt));

}

boolean[] visited = new boolean[A+1];

PriorityQueue<Node> pq = new PriorityQueue<Node>(new myCMP());

pq.add(new Node(1, 0));

long cost = 0;

int mod = (1000\*1000\*1000+7);

while(pq.size() > 0) {

Node curr = pq.poll();

if(visited[curr.val]) continue;

cost += curr.wt;

cost %= mod;

visited[curr.val] = true;

for(int i=0;i<graph.get(curr.val).size();i++) {

Node child = graph.get(curr.val).get(i);

if(!visited[child.val]) {

pq.add(new Node(child.val, child.wt));

}

}

}

return (int)cost;

}

}

**Q18. Floyd Warshall Algorithm**

Given a matrix of integers **A** of size **N x N**, where **A[i][j]** represents the weight of directed edge from i to j (i ---> j).

If **i == j, A[i][j] = 0**, and if there is no directed edge from vertex i to vertex j, **A[i][j] = -1**.

Return a matrix **B** of size **N x N** where **B[i][j]** = shortest path from vertex i to vertex j.

If there is no possible path from vertex i to vertex j , **B[i][j] = -1**

**Note:** Rows are numbered from top to bottom and columns are numbered from left to right.

**Logic –** Give each node a chance to come in between every 2 nodes, and calculate the minimum distance.

**Code -**

public class Solution {

// Check if a vertex can come between any 2 vertices

public int[][] solve(int[][] A) {

int n = A.length;

long [][] dist = new long[n][n];

// Creating dist matrix

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

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

if(A[i][j] == -1) dist[i][j] = Integer.MAX\_VALUE;

else dist[i][j] = A[i][j];

}

}

// Main algo

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

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

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

dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);

}

}

}

// Convert int to long

int[][] ans = new int[n][n];

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

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

if(dist[i][j] == Integer.MAX\_VALUE) ans[i][j] = -1;

else ans[i][j] = (int) dist[i][j];

}

}

return ans;

}

}

**Q19. Bellman Ford Algorithm**

**Logic –**

1. It first calculates the shortest distances which have at most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on.
2. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated.
3. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times.
4. The idea is, assuming that there is no negative weight cycle if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give the shortest path with at-most (i+1) edges

**Code -**

// Main Algorithm

int[] dist = new int[];

Array.fill(dist, Integer.MAX\_VALUE);

dist[source] = 0;

for(int i=1;i<n;i++) {

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

int u = B[j][0];

int v = B[j][1];

int w = B[j][2];

dist[v] = min(dist[v], dist[u] + w);

}

}

***In the nth iteration if the dist[u] further decreases then there is a negative cycle in the graph.***

**5 graph problems remaining**