**Q1. Black Shapes**

**Problem Description**

Given character matrix **A** of dimensions **N×M** consisting of O's and X's, where O = white, X = black.

Return the number of black shapes. A black shape consists of one or more adjacent X's (diagonals not included)

**Problem Constraints**

1 <= **N**, **M** <= 1000

A[i][j] = 'X' or 'O'

**Input Format**

The First and only argument is character matrix **A**.

**Output Format**

Return a single integer denoting number of black shapes.

**Example Input**

Input 1:

A = [ [X, X, X], [X, X, X], [X, X, X] ]

Input 2:

A = [ [X, O], [O, X] ]

**Example Output**

Output 1:

1

Output 2:

2

**Example Explanation**

Explanation 1:

All X's belong to single shapes

Explanation 2:

Both X's belong to different shapes

public class Solution {

public int black(String[] A) {

int row = A.length;

int col = A[0].length();

int count = 0;

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

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

if (A[i].charAt(j) == 'X') {

count++;

dfs(A, i, j);

}

}

}

return count;

}

public void dfs(String[] A, int i, int j) {

if (i < 0 || i > A.length - 1 || j < 0 || j > A[0].length() - 1 || A[i].charAt(j) == 'O'

|| A[i].charAt(j) == 'Z') {

return;

}

String str = A[i];

StringBuilder sb = new StringBuilder(str);

sb.setCharAt(j, 'Z');

A[i] = sb.toString();

dfs(A, i + 1, j);

dfs(A, i - 1, j);

dfs(A, i, j + 1);

dfs(A, i, j - 1);

}

}

**Q2. Capture Regions on Board**

**Problem Description**

Given a 2-D board **A** of size **N x M** containing **'X' and 'O'**, capture all regions surrounded by **'X'**.

A region is captured by flipping all **'O's** into **'X's** in that surrounded region.

**Problem Constraints**

1 <= N, M <= 1000

**Input Format**

First and only argument is a N x M character matrix A.

**Output Format**

Return nothing. Make changes to the the input only as matrix is passed by reference.

**Example Input**

Input 1:

A = [

[X, X, X, X],

[X, O, O, X],

[X, X, O, X],

[X, O, X, X]

]

Input 2:

A = [

[X, O, O],

[X, O, X],

[O, O, O]

]

**Example Output**

Output 1:

After running your function, the board should be:

A = [

[X, X, X, X],

[X, X, X, X],

[X, X, X, X],

[X, O, X, X]

]

Output 2:

After running your function, the board should be:

A = [

[X, O, O],

[X, O, X],

[O, O, O]

]

**Example Explanation**

Explanation 1:

O in (4,2) is not surrounded by X from below.

Explanation 2:

No O's are surrounded.

public class Solution {

public void solve(ArrayList<ArrayList<Character>> a) {

int r = a.size();

int c = a.get(0).size();

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

if (a.get(i).get(0) == 'O') {

dfs(a, i, 0, r, c);

}

if (a.get(i).get(c - 1) == 'O') {

dfs(a, i, c - 1, r, c);

}

}

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

if (a.get(0).get(j) == 'O') {

dfs(a, 0, j, r, c);

}

if (a.get(r - 1).get(j) == 'O') {

dfs(a, r - 1, j, r, c);

}

}

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

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

if (a.get(i).get(j) == 'O') {

a.get(i).set(j, 'X');

}

if (a.get(i).get(j) == 'S') {

a.get(i).set(j, 'O');

}

}

}

}

public void dfs(ArrayList<ArrayList<Character>> a, int i, int j, int row, int col) {

if (i < 0 || i > row - 1 || j < 0 || j > col - 1 || a.get(i).get(j) != 'O') {

return;

}

a.get(i).set(j, 'S');

dfs(a, i + 1, j, row, col);

dfs(a, i - 1, j, row, col);

dfs(a, i, j + 1, row, col);

dfs(a, i, j - 1, row, col);

}

}

**3. Shortest Distance in a Maze**

**Problem Description**

Given a matrix of integers A of size **N** x **M** describing a maze. The maze consists of empty locations and walls.

1 represents a wall in a matrix and 0 represents an empty location in a wall.

There is a ball trapped in a maze. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall (maze boundary is also considered as a wall). When the ball stops, it could choose the next direction.

Given two array of integers of size B and C of size 2 denoting the starting and destination position of the ball.

Find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the starting position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

**Problem Constraints**

2 <= **N**, **M** <= 100

0 <= A[i] <= 1

0 <= B[i][0], C[i][0] < N

0 <= B[i][1], C[i][1] < M

**Input Format**

The first argument given is the integer matrix **A**.

The second argument given is an array of integer **B**.

The third argument if an array of integer **C**.

**Output Format**

Return a single integer, the minimum distance required to reach destination

**Example Input**

Input 1:

A = [ [0, 0],   
 [0, 0] ]

B = [0, 0]

C = [0, 1]

Input 2:

A = [ [0, 1],   
 [1, 0] ]

B = [0, 0]

C = [1, 1]

**Example Output**

Output 1:

1

Output 2:

-1

public class ShortestDistanceInAMaze {

class Point {

int x;

int y;

int steps;

public Point(int r, int c, int step) {

x = r;

y = c;

steps = step;

}

}

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

int n = A.length;

int m = A[0].length;

boolean visited[][] = new boolean[n][m];

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

int dir[] = new int[] { 0, 1, 0, -1, 0 };

q.add(new Point(B[0], B[1], 0));

while (!q.isEmpty()) {

Point currP = q.poll();

int currR = currP.x;

int currC = currP.y;

if (currR == C[0] && currC == C[1]) {

return currP.steps;

}

for (int i = 0; i < dir.length - 1; i++) {

int nRow = currR;

int nCol = currC;

int step = currP.steps;

while (nRow >= 0 && nRow < n && nCol >= 0 && nCol < m && A[nRow][nCol] == 0) {

nRow += dir[i];

nCol += dir[i + 1];

step++;

}

nRow -= dir[i];

nCol -= dir[i + 1];

step--;

if (!visited[nRow][nCol]) {

q.add(new Point(nRow, nCol, step));

visited[nRow][nCol] = true;

}

}

}

return -1;

}

4. **Possibility of Finishing**

**Problem Description**

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.

**Problem Constraints**

1 <= A <= 6\*104

1 <= length(B) = length(C) <= 105

1 <= B[i], C[i] <= A

**Input Format**

The first argument of input contains an integer A, representing the number of courses.

The second argument of input contains an integer array, B.

The third argument of input contains an integer array, C.

**Output Format**

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

**Example Input**

Input 1:

A = 3

B = [1, 2]

C = [2, 3]

Input 2:

A = 2

B = [1, 2]

C = [2, 1]

**Example Output**

Output 1:

1

Output 2:

0

public class Solution {

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

ArrayList<Integer>[] g = createAdjList(A, B, C);

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

Arrays.fill(v, false);

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

Arrays.fill(st, 0);

return detectCycle(g, v, st) ? 0 : 1;

}

public boolean detectCycle(ArrayList<Integer>[] g, boolean[] v, int[] st) {

for (int i = 1; i < g.length; i++) {

if (!v[i] && dfs(i, g, v, st)) {

return true;

}

}

return false;

}

public boolean dfs(int s, ArrayList<Integer>[] g, boolean[] v, int[] st) {

v[s] = true;

st[s] = 1;

for (int i = 0; i < g[s].size(); i++) {

int neighbor = g[s].get(i);

if (!v[neighbor] && dfs(neighbor, g, v, st)) {

return true;

}

if (st[neighbor] == 1) {

return true;

}

}

st[s] = 0;

return false;

}

public ArrayList<Integer>[] createAdjList(int A, int[] B, int[] C) {

ArrayList<Integer>[] g = new ArrayList[A + 1];

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

g[i] = new ArrayList<Integer>();

}

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

int from = B[i];

int to = C[i];

g[from].add(to);

}

return g;

}

}

5. **Maximum Depth**

**Problem Description**

Given a Tree of **A** nodes having A-1 edges. Each node is numbered from **1** to **A** where **1** is the root of the tree.

You are given **Q** queries. In each query, you will be given two integers **L** and **X**. Find the value of such node which lies at level **L mod (MaxDepth + 1)** and has value greater than or equal to **X**.

Answer to the query is the **smallest** possible value or **-1**, if all the values at the required level are smaller than **X**.

**NOTE:**

 Level and Depth of the root is considered as 0.

 It is guaranteed that each edge will be connecting exactly two different nodes of the tree.

 Please read the input format for more clarification.

**Problem Constraints**

2 <= A, Q(size of array E and F) <= 105

1 <= B[i], C[i] <= A

1 <= D[i], E[i], F[i] <= 106

**Input Format**

The first argument is an integer A denoting the number of nodes.

The second and third arguments are the integer arrays B and C where for each i (0 <= i < A-1), B[i] and C[i] are the nodes connected by an edge.

The fourth argument is an integer array D, where D[i] denotes the value of the (i+1)th node

The fifth and sixth arguments are the integer arrays E and F where for each i (0 <= i < Q), E[i] denotes L and F[i] denotes X for ith query.

**Output Format**

Return an array of integers where the ith element denotes the answer to ith query.

**Example Input**

Input 1:

A = 5

B = [1, 4, 3, 1]

C = [5, 2, 4, 4]

D = [7, 38, 27, 37, 1]

E = [1, 1, 2]

F = [32, 18, 26]

Input 2:

A = 3

B = [1, 2]

C = [3, 1]

D = [7, 15, 27]

E = [1, 10, 1]

F = [29, 6, 26]

**Example Output**

Output 1:

[37, 37, 27]

Output 2:

[-1, 7, 27]

public class Solution {

static int maxn = 100009;

static int n, q;

static int mx = 0;

static int[] val = new int[maxn];

static ArrayList < ArrayList < Integer > > adj;

static ArrayList < ArrayList < Integer > > lvl;

public static void graph() {

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

lvl = new ArrayList < ArrayList < Integer > > (maxn);

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

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

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

}

mx = 0;

}

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

graph();

n = A;

q = F.length;

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

val[i + 1] = D[i];

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

adj.get(B[i]).add(C[i]);

adj.get(C[i]).add(B[i]);

}

mx = 0;

dfs(1, 1, 0);

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

Collections.sort(lvl.get(i));

}

int[] res = new int[q];

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

int l = E[i];

int x = F[i];

l %= (mx + 1);

int it = lowerBound(lvl.get(l), 0, lvl.get(l).size(), x);

if (it == lvl.get(l).size())

res[i] = -1;

else res[i] = lvl.get(l).get(it);

}

return res;

}

public static void dfs(int u, int v, int d) {

mx = Math.max(mx, d);

lvl.get(d).add(val[u]);

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

if (i == v) continue;

dfs(i, u, d + 1);

}

}

static int lowerBound(ArrayList < Integer > a, int low, int high, int element) {

while (low < high) {

int middle = low + (high - low) / 2;

if (element > a.get(middle)) {

low = middle + 1;

} else {

high = middle;

}

}

return low;

}

}

6. **First Depth First Search**

**Problem Description**

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.

**Problem Constraints**

1 <= **N** <= 100000

**Input Format**

First argument is vector **A**

Second argument is integer **B**

Third argument is integer **C**

**Output Format**

Return 1 if reachable, 0 otherwise.

**Example Input**

Input 1:

A = [1, 1, 2]

B = 1

C = 2

Input 2:

A = [1, 1, 2]

B = 2

C = 1

**Example Output**

Output 1:

0

Output 2:

1

public class Solution {

// DO NOT MODIFY THE ARGUMENTS WITH "final" PREFIX. IT IS READ ONLY

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

ArrayList<Integer>[] graph = createGraph(A);

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

Arrays.fill(visited, false);

boolean ans = checkPath(B, C, visited, graph);

return ans ? 1 : 0;

}

public boolean checkPath(int b, int c, boolean[] visited, ArrayList<Integer>[] graph) {

visited[c] = true;

if (b == c) {

return true;

}

if (!graph[c].isEmpty()) {

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

int newTown = graph[c].get(i);

if (!visited[newTown]) {

if (checkPath(b, newTown, visited, graph)) {

return true;

}

}

if (newTown == b) {

return true;

}

}

}

return false;

}

public ArrayList<Integer>[] createGraph(int[] A) {

int N = A.length;

ArrayList<Integer>[] g = new ArrayList[N + 1];

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

g[i] = new ArrayList<>();

}

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

int from = A[i];

int to = i + 1;

g[from].add(to);

}

return g;

}

}

7. **Path in Directed Graph**

**Problem Description**

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**.

**NOTE:**

* There are no self-loops in the graph.
* There are no multiple edges between two nodes.
* The graph may or may not be connected.
* Nodes are numbered from 1 to A.
* Your solution will run on multiple test cases. If you are using global variables make sure to clear them.

**Problem Constraints**

2 <= A <= 105

1 <= M <= min(200000,A\*(A-1))

1 <= B[i][0], B[i][1] <= A

**Input Format**

The first argument given is an integer **A** representing the number of nodes in the graph.

The second argument given a matrix **B** of size M x 2 which represents the **M** edges such that there is a edge directed from node **B[i][0]** to node **B[i][1]**.

**Output Format**

Return **1** if path exists between node **1** to node **A** else return **0**.

**Example Input**

Input 1:

A = 5

B = [ [1, 2]

[4, 1]

[2, 4]

[3, 4]

[5, 2]

[1, 3] ]

Input 2:

A = 5

B = [ [1, 2]

[2, 3]

[3, 4]

[4, 5] ]

**Example Output**

Output 1:

0

Output 2:

1

public class Solution {

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

ArrayList<Integer>[] g = createAdjList(A, B);

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

Arrays.fill(v, false);

return dfs(1, A, g, v);

}

public int dfs(int s, int A, ArrayList<Integer>[] g, boolean[] v) {

v[s] = true;

if (s == A) {

return 1;

}

for (int i = 0; i < g[s].size(); i++) {

int neighbor = g[s].get(i);

if (!v[neighbor]) {

if (dfs(neighbor, A, g, v) == 1) {

return 1;

}

}

}

return 0;

}

public ArrayList<Integer>[] createAdjList(int A, int[][] B) {

ArrayList<Integer>[] g = new ArrayList[A + 1];

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

g[i] = new ArrayList<>();

}

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

int from = B[i][0];

int to = B[i][1];

g[from].add(to);

}

return g;

}

}

8. **Cycle in Directed Graph**

**Problem Description**

Given an directed graph having **A** nodes. A matrix **B** of size M x 2 is given which represents the **M** edges such that there is a edge directed from node **B[i][0]** to node **B[i][1]**.

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

**NOTE:**

* The cycle must contain atleast two nodes.
* There are no self-loops in the graph.
* There are no multiple edges between two nodes.
* The graph may or may not be connected.
* Nodes are numbered from 1 to A.
* Your solution will run on multiple test cases. If you are using global variables make sure to clear them.

**Problem Constraints**

2 <= A <= 105

1 <= M <= min(200000,A\*(A-1))

1 <= B[i][0], B[i][1] <= A

**Input Format**

The first argument given is an integer **A** representing the number of nodes in the graph.

The second argument given a matrix **B** of size M x 2 which represents the **M** edges such that there is a edge directed from node **B[i][0]** to node **B[i][1]**.

**Output Format**

Return **1** if cycle is present else return **0**.

**Example Input**

Input 1:

A = 5

B = [ [1, 2]

[4, 1]

[2, 4]

[3, 4]

[5, 2]

[1, 3] ]

Input 2:

A = 5

B = [ [1, 2]

[2, 3]

[3, 4]

[4, 5] ]

**Example Output**

Output 1:

1

Output 2:

0

public class Solution {

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

ArrayList<Integer>[] g = createAdjList(A, B);

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

Arrays.fill(v, false);

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

Arrays.fill(st, 0);

boolean ans = detectCycle(g, A, v, st);

return ans ? 1 : 0;

}

public boolean detectCycle(ArrayList<Integer>[] g,int A, boolean[] v, int[] st) {

for (int i = 1; i < g.length; i++) {

if (!v[i] && dfs(i, A, v, g, st)) {

return true;

}

}

return false;

}

public boolean dfs(int s, int A, boolean[] v, ArrayList<Integer>[] g, int[] st) {

v[s] = true;

st[s] = 1;

for (int i = 0; i < g[s].size(); i++) {

int neighbor = g[s].get(i);

if (!v[neighbor] && dfs(neighbor, A, v, g, st)) {

return true;

}

if (st[neighbor] == 1) {

return true;

}

}

st[s] = 0;

return false;

}

public ArrayList<Integer>[] createAdjList(int A, int[][] B) {

ArrayList<Integer>[] g = new ArrayList[A + 1];

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

g[i] = new ArrayList<>();

}

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

int from = B[i][0];

int to = B[i][1];

g[from].add(to);

}

return g;

}

}

9. **Number of islands**

**Problem Description**

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.

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

**Problem Constraints**

1 <= N, M <= 100

0 <= A[i] <= 1

**Input Format**

The only argument given is the integer matrix A.

**Output Format**

Return the number of islands.

**Example Input**

Input 1:

A = [

[0, 1, 0]

[0, 0, 1]

[1, 0, 0]

]

Input 2:

A = [

[1, 1, 0, 0, 0]

[0, 1, 0, 0, 0]

[1, 0, 0, 1, 1]

[0, 0, 0, 0, 0]

[1, 0, 1, 0, 1]

]

**Example Output**

Output 1:

2

Output 2:

5

public class Solution {

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

int row = A.length;

int col = A[0].length;

int count = 0;

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

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

if (A[i][j] == 1) {

count++;

dfs(A, i, j);

}

}

}

return count;

}

public void dfs(int[][] A, int i, int j) {

if (i < 0 || i > A.length - 1 || j < 0 || j > A[0].length - 1 || A[i][j] == 2 || A[i][j] == 0) {

return;

}

A[i][j] = 2;

dfs(A, i + 1, j);

dfs(A, i - 1, j);

dfs(A, i, j + 1);

dfs(A, i, j - 1);

dfs(A, i + 1, j + 1);

dfs(A, i + 1, j - 1);

dfs(A, i - 1, j + 1);

dfs(A, i - 1, j - 1);

}

}

10. **Rotten Oranges**

**Problem Description**

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.

**Note**: Your solution will run on multiple test cases. If you are using global variables, make sure to clear them.

**Problem Constraints**

1 <= **N**, **M** <= 1000

0 <= **A[i][j]** <= 2

**Input Format**

The first argument given is the integer matrix **A**.

**Output Format**

Return the minimum number of minutes that must elapse until no cell has a fresh orange.

If this is impossible, return -1 instead.

**Example Input**

Input 1:

A = [ [2, 1, 1]

[1, 1, 0]

[0, 1, 1] ]

Input 2:

A = [ [2, 1, 1]

[0, 1, 1]

[1, 0, 1] ]

**Example Output**

Output 1:

4

Output 2:

-1

public class Solution {

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

int n = A.length;

int m = A[0].length;

Queue<int[]> q = new LinkedList<>();

int dir[] = new int[] { 0, 1, 0, -1, 0 };

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

for (int j = 0; j < A[0].length; j++) {

if (A[i][j] == 2) {

q.add(new int[] { i, j, 0 });

}

}

}

int time = 0;

while (!q.isEmpty()) {

int[] cell = q.poll();

int r = cell[0];

int c = cell[1];

time = Math.max(time, cell[2]);

for (int i = 0; i < dir.length - 1; i++) {

int nR = r + dir[i];

int nC = c + dir[i + 1];

if (nR >= 0 && nR < n && nC >= 0 && nC < m && A[nR][nC] == 1) {

A[nR][nC] = 2;

q.add(new int[] { nR, nC, time + 1 });

}

}

}

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

for (int j = 0; j < A[0].length; j++) {

if (A[i][j] == 1) {

return -1;

}

}

}

return time;

}

}

11. **Another BFS (Trick involved)**

**Problem Description**

Given a weighted undirected graph having A nodes, a source node C and destination node D.

Find the shortest distance from C to D and if it is impossible to reach node D from C then return -1.

You are expected to do it in Time Complexity of O(A + M).

**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.

**Problem Constraints**

1 <= **A** <= 105

0 <= **B**[i][0], **B**[i][1] < **A**

1 <= **B**[i][2] <= 2

0 <= **C** < **A**

0 <= **D** < **A**

**Input Format**

The first argument given is an integer **A**, representing the number of nodes.

The second argument given is the matrix **B**, where B[i][0] and B[i][1] are connected through an edge of weight B[i][2].

The third argument given is an integer **C**, representing the source node.

The fourth argument is an integer **D**, representing the destination node.

Note: B[i][2] will be either 1 or 2.

**Output Format**

Return the shortest distance from C to D. If it is impossible to reach node D from C then return -1.

**Example Input**

Input 1:

A = 6

B = [ [2, 5, 1]

[1, 3, 1]

[0, 5, 2]

[0, 2, 2]

[1, 4, 1]

[0, 1, 1] ]

C = 3

D = 2

Input 2:

A = 2

B = [ [0, 1, 1]

]

C = 0

D = 1

**Example Output**

Output 1:

4

Output 2:

1

public class Solution {

class Pair {

int node;

int dist;

public Pair(int x, int y) {

node = x;

dist = y;

}

}

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

if(C==D) {

return 0;

}

ArrayList<ArrayList<Integer>> g = createAdjList(A, B);

boolean v[] = new boolean[g.size()];

Arrays.fill(v, false);

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

q.add(new Pair(C, 0));

v[C] = true;

while (!q.isEmpty()) {

Pair p = q.poll();

int cNode = p.node;

int d = p.dist;

for (int i = 0; i < g.get(cNode).size(); i++) {

int neighbor = g.get(cNode).get(i);

if (!v[neighbor]) {

if (neighbor == D) {

return d + 1;

}

q.add(new Pair(neighbor, d + 1));

v[neighbor] = true;

}

}

}

return -1;

}

public ArrayList<ArrayList<Integer>> createAdjList(int A, int[][] B) {

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

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

g.add(i, new ArrayList<Integer>());

}

int size = A;

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

int from = B[i][0];

int to = B[i][1];

int wt = B[i][2];

if (wt == 2) {

int dummy = size;

g.add(dummy, new ArrayList<Integer>());

g.get(from).add(dummy);

g.get(dummy).add(from);

g.get(dummy).add(to);

g.get(to).add(dummy);

size++;

} else {

g.get(from).add(to);

g.get(to).add(from);

}

}

return g;

}

}

12. **Commutable Islands**

**Problem Description**

There are **A** islands and there are **M** bridges connecting them. Each bridge has some **cost** attached to it.

We need to find bridges with **minimal cost** such that all islands are connected.

It is guaranteed that input data will contain **at least one** possible scenario in which all islands are connected with each other.

**Problem Constraints**

1 <= A, M <= 6\*104

1 <= B[i][0], B[i][1] <= A

1 <= B[i][2] <= 103

**Input Format**

The first argument contains an integer, **A**, representing the number of islands.

The second argument contains an 2-d integer matrix, **B**, of size M x 3 where Island B[i][0] and B[i][1] are connected using a bridge of cost B[i][2].

**Output Format**

Return an integer representing the minimal cost required.

**Example Input**

Input 1:

A = 4

B = [ [1, 2, 1]

[2, 3, 4]

[1, 4, 3]

[4, 3, 2]

[1, 3, 10] ]

Input 2:

A = 4

B = [ [1, 2, 1]

[2, 3, 2]

[3, 4, 4]

[1, 4, 3] ]

**Example Output**

Output 1:

6

Output 2:

6

public class Solution {

class Pair {

int dist;

int next;

public Pair(int x, int y) {

dist = x;

next = y;

}

}

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

int ans = 0;

ArrayList<Pair>[] g = createAdjList(A, B);

PriorityQueue<Pair> pq = new PriorityQueue<>((a, b) -> a.dist - b.dist);

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

Arrays.fill(v, false);

int source = B[0][0];

Pair p = new Pair(B[0][2], B[0][1]);

pq.add(p);

v[source] = true;

for (int i = 0; i < g[source].size(); i++) {

pq.add(new Pair(g[source].get(i).dist, g[source].get(i).next));

}

while (!pq.isEmpty()) {

Pair p1 = pq.poll();

int neighbor = p1.next;

if (v[neighbor]) {

continue;

}

v[neighbor] = true;

ans += p1.dist;

for (int i = 0; i < g[neighbor].size(); i++) {

int nextNode = g[neighbor].get(i).next;

if (!v[nextNode]) {

pq.add(g[neighbor].get(i));

}

}

}

return ans;

}

public ArrayList<Pair>[] createAdjList(int A, int[][] B) {

ArrayList<Pair>[] g = new ArrayList[A + 1];

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

g[i] = new ArrayList<>();

}

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

g[B[i][0]].add(new Pair(B[i][2], B[i][1]));

g[B[i][1]].add(new Pair(B[i][2], B[i][0]));

}

return g;

}

}

13. **Dijkstra**

**Problem Description**

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 from 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.
* There are no multiple edges between two pairs of vertices.
* The graph may or may not be connected.
* Nodes are numbered from 0 to A-1.
* Your solution will run on multiple test cases. If you are using global variables, make sure to clear them.

**Problem Constraints**

1 <= A <= 1e5

0 <= B[i][0],B[i][1] < A

0 <= B[i][2] <= 1e3

0 <= C < A

**Input Format**

* The first argument is an integer A, representing the number of nodes in the graph.
* The second argument is a matrix B of size M x 3, where each row represents an edge in the graph. The three columns of each row denote the **source node B[i][0]**, **the destination node B[i][1]**, and the **weight of the edge B[i][2]**.
* The third argument is an integer **C**, representing the **source node** for which the shortest distance to all other nodes needs to be found.

**Output Format**

Return the integer array D.

**Example Input**

Input 1:

A = 6

B = [ [0, 4, 9]

[3, 4, 6]

[1, 2, 1]

[2, 5, 1]

[2, 4, 5]

[0, 3, 7]

[0, 1, 1]

[4, 5, 7]

[0, 5, 1] ]

C = 4

Input 2:

A = 5

B = [ [0, 3, 4]

[2, 3, 3]

[0, 1, 9]

[3, 4, 10]

[1, 3, 8] ]

C = 4

**Example Output**

Output 1:

D = [7, 6, 5, 6, 0, 6]

Output 2:

D = [14, 18, 13, 10, 0]

public class Solution {

class Pair {

int wt;

int next;

public Pair(int x, int y) {

wt = x;

next = y;

}

}

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

ArrayList<Pair>[] g = createAdjList(A, B);

int ans[] = new int[A];

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

ans[C] = 0;

Queue<Pair> pq = new PriorityQueue<>((a, b) -> a.wt - b.wt);

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

pq.add(new Pair(g[C].get(i).wt, g[C].get(i).next));

}

while (!pq.isEmpty()) {

Pair pNode = pq.poll();

int wt = pNode.wt;

int next = pNode.next;

if (ans[next] > wt) {

ans[next] = wt;

} else {

continue;

}

for (int i = 0; i < g[next].size(); i++) {

pq.add(new Pair(g[next].get(i).wt + ans[next], g[next].get(i).next));

}

}

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

if(ans[i]==Integer.MAX\_VALUE) {

ans[i] = -1;

}

}

return ans;

}

public ArrayList<Pair>[] createAdjList(int A, int[][] B) {

ArrayList<Pair>[] g = new ArrayList[A];

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

g[i] = new ArrayList<Pair>();

}

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

g[B[i][0]].add(new Pair(B[i][2], B[i][1]));

g[B[i][1]].add(new Pair(B[i][2], B[i][0]));

}

return g;

}

}

14. **Construction Cost**

**Problem Description**

Flipkart has ‘**A**’ local distribution centers located across a large metropolitan city. Each distribution center needs to be interconnected through roads to facilitate efficient movement of goods. The cost of constructing a road between any two distribution centers is represented by the weight of the edge connecting them.  
  
Given a graph with ‘**A**’ nodes representing the distribution centers and **C** weighted edges representing the possible roads between them, your task is to find the minimum total cost of constructing roads such that every distribution center can be reached from the first distribution center.  
  
Cost Calculation:  
The cost of constructing the roads is the sum of the weights of the edges selected for the construction.  
  
**NOTE**: Return the answer modulo **10^9+7** as the answer can be large.

**Problem Constraints**

1 <= A <= 100000  
0 <= C <= 100000  
1 <= B[i][0], B[i][1] <= N  
1 <= B[i][2] <= 109

**Input Format**

First argument is an integer A.  
Second argument is a 2-D integer array B of size C×3 denoting edges. B[i][0]and B[i][1]are the distribution centers connected by the ith edge with construction cost B[i][2].

**Output Format**

Return an integer denoting the minimum construction cost.

**Example Input**

Input 1:

A = 3

B = [ [1, 2, 14]

[2, 3, 7]

[3, 1, 2] ]

Input 2:

A = 3

B = [ [1, 2, 20]

[2, 3, 17] ]

**Example Output**

Output 1:

9

Output 2:

37

public class Solution {

class Pair {

int dist;

int next;

public Pair(int x, int y) {

dist = x;

next = y;

}

}

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

if (B.length < 1) {

return 0;

}

ArrayList<Pair>[] g = createAdjList(A, B);

Queue<Pair> pq = new PriorityQueue<>((a, b) -> a.dist - b.dist);

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

Arrays.fill(v, false);

int startingNode = B[0][0];

v[startingNode] = true;

for (int i = 0; i < g[startingNode].size(); i++) {

pq.add(new Pair(g[startingNode].get(i).dist, g[startingNode].get(i).next));

}

long total = 0;

while (!pq.isEmpty()) {

Pair p1 = pq.poll();

int neighbor = p1.next;

if (!v[neighbor]) {

total += p1.dist;

v[neighbor] = true;

for (int i = 0; i < g[neighbor].size(); i++) {

pq.add(new Pair(g[neighbor].get(i).dist, g[neighbor].get(i).next));

}

}else {

continue;

}

}

total = total%1000000007;

return (int)total;

}

public ArrayList<Pair>[] createAdjList(int A, int[][] B) {

ArrayList<Pair>[] g = new ArrayList[A + 1];

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

g[i] = new ArrayList<>();

}

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

g[B[i][0]].add(new Pair(B[i][2], B[i][1]));

g[B[i][1]].add(new Pair(B[i][2], B[i][0]));

}

return g;

}

}

15. **Largest Distance between nodes of a Tree**

**Problem Description**

Find largest distance Given an arbitrary unweighted rooted tree which consists of **N** (2 <= N <= 40000) nodes.

The goal of the problem is to find largest distance between two nodes in a tree. Distance between two nodes is a number of edges on a path between the nodes (there will be a unique path between any pair of nodes since it is a tree).

The nodes will be numbered 0 through N - 1.

The tree is given as an array **A**, there is an edge between nodes A[i] and i (0 <= i < N). Exactly one of the i's will have A[i] equal to -1, it will be root node.

**Problem Constraints**

2 <= **|A|** <= 40000

**Input Format**

First and only argument is vector **A**

**Output Format**

Return the length of the longest path

**Example Input**

Input 1:

A = [-1, 0]

Input 2:

A = [-1, 0, 0]

**Example Output**

Output 1:

1

Output 2:

2

public class LargestDistanceBetweenNodesOfATree {

class Pair {

int dist;

int node;

public Pair(int x, int y) {

dist = x;

node = y;

}

}

int root = 0;

Pair ans;

public int solve(int[] A) {

ArrayList<Integer>[] g = createAdjList(A);

boolean v[] = new boolean[A.length];

ans = new Pair(-1, -1);

dfs(root, 0, g, v);

Arrays.fill(v, false);

dfs(ans.node, 0, g, v);

return ans.dist;

}

public void dfs(int src, int dist, ArrayList<Integer>[] g, boolean v[]) {

v[src] = true;

if (dist > ans.dist) {

ans.dist = dist;

ans.node = src;

}

for (int i : g[src]) {

if (!v[i]) {

dfs(i, dist + 1, g, v);

}

}

}

public ArrayList<Integer>[] createAdjList(int[] A) {

ArrayList<Integer>[] g = new ArrayList[A.length];

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

g[i] = new ArrayList<>();

}

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

if (A[i] != -1) {

g[i].add(A[i]);

g[A[i]].add(i);

} else {

root = i;

}

}

return g;

}