# MOST FREQUENT ELEMENT

import java.util.**\***;

public class Source {

  public static int mostFrequentElement(int[] *arr*) {

*// Write code here*

    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>(); *// Creating a Hashmap and adding all array elements as keys*

    int maxfreq = 0;

    int result = 0;

    for (int i = 0; i < *arr*.length; i++) { *//Iterating through the array*

      int key = *arr*[i];

      if (map.containsKey(key)) { *//if the key being iterated over is present in hashmap*

        int freq = map.get(key);

        freq++; *//increment frequency*

        map.put(key, freq); *//store element and it's frequency in a key value pair*

      } else {

        map.put(key, 1); *//else element has only appeared once and frequency is only 1*

      }

    }

    for (Map.Entry<Integer, Integer> value : map.entrySet()) { *//iterating throught the hashmap using entryset*

      if (maxfreq < value.getValue()) { *//if maximum frequency is lesser than iterated element in the hashmap*

        result = value.getKey(); *//result is set to the most frequent element*

        maxfreq = value.getValue(); *//maximum frequency is updated to the current frequency of iterated element*

      }

    }

    return result;

  }

  public static void main(String[] *args*) {

    int n;

    Scanner sc = new Scanner(System.in);

    n = sc.nextInt();

    int arr[] = new int[n];

    for (int i = 0; i < n; i++) {

      arr[i] = sc.nextInt();

    }

    System.out.println(mostFrequentElement(arr));

  }

}

# Check Whether an Undirected Graph is a Tree or Not

import java.util.**\***;

public class Source {

    private int vertexCount;

    private static LinkedList<Integer> adj[];

    Source(int *vertexCount*) {

**this**.vertexCount = *vertexCount*;

**this**.adj = new LinkedList[*vertexCount*];

        for (int i = 0; i < *vertexCount*; ++i) {

            adj[i] = new LinkedList<Integer>();

        }

    }

    public void addEdge(int *v*, int *w*) {

        if (!isValidIndex(*v*) || !isValidIndex(*w*)) {

            return;

        }

        adj[*v*].add(*w*);

        adj[*w*].add(*v*);

    }

  private boolean isValidIndex(int *i*) {

*// Write code here*

    if ((*i* < vertexCount) && (*i* >= 0)) { *// If index is lesser than total number of vertexes*

      return true; *// and is non zero, return True.*

    } else {

      return false;

    }

  }

  private boolean isCyclic(int *v*, boolean *visited*[], int *parent*) {

*// Write code here*

*visited*[*v*] = true; *//Setting the vertex being checked as visited.*

    for (int i : adj[*v*]) { *// Iterating through adjacent vertices array*

      if (!*visited*[i]) { *// If vertex being iterated has not been visited,*

        if (isCyclic(i, *visited*, *v*)) { *//isCyclic is recursively called to check for cycles from i.*

          return true; *//then graph is cyclic.*

        }

      } else if (i != *parent*) { *//also if i is equal to the parent, it means i is visiting it,*

        return true; *//then also graph is cyclic.*

      }

    }

    return false;

  }

  public boolean isTree() {

*// Write Code here*

    boolean visited[] = new boolean[vertexCount]; *// Making a boolean array for each vertex.*

    if (isCyclic(0, visited, -1)) {

      return false; *// If graph is not connected, it can not be cycle, hence not a tree.*

    }

    for (int i = 0; i < visited.length; i++) { *// Checking if all vertices are visited*

      if (!visited[i]) {

        return false; *// if a vertice is unvisited, graph is not a tree.*

      }

    }

    return true; *//No cycles and graph is cyclic.*

  }

  public static void main(String *args*[]) {

    Scanner sc = new Scanner(System.in);

*// Get the number of nodes from the input.*

    int noOfNodes = sc.nextInt();

*// Get the number of edges from the input.*

    int noOfEdges = sc.nextInt();

    Source graph = new Source(noOfNodes);

*// Adding edges to the graph*

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

      graph.addEdge(sc.nextInt(), sc.nextInt());

    }

    if (graph.isTree()) {

      System.out.println("Yes"); *//changed to match output*

    } else {

      System.out.println("No"); *//changed to match output*

    }

  }

}

# Find All Symmetric Pairs in an Array

import java.util.**\***;

public class Source {

    public static void symmetricPair(int[][] *arr*) {

*// Write code here*

    HashMap<Integer, Integer> map = new HashMap<Integer, Integer>(); *//Creating a hashmap*

    for (int i = 0;i < *arr*.length; i++) { *//Iterating throught the array*

      int first = *arr*[i][0]; *//Storing iterated pair first element in variable*

      int second = *arr*[i][1]; *//Storing iterated pair second element in variable*

      if (map.get(second) != null && map.get(second) == first) { *//if the second element of pair in map is equal to first element being iterated,*

        System.out.print(second+ " " +first+"\n"); *//print the symmetric pair.*

      } else {

        map.put(first, second); *//Putting the first and second elements into the hashmaps*

      }

    }

  }

  public static void main(String *arg*[]) {

    Scanner sc = new Scanner(System.in);

    int row = sc.nextInt();

    int arr[][] = new int[row][2];

    for (int i = 0; i < row; i++) {

      for (int j = 0; j < 2; j++) {

        arr[i][j] = sc.nextInt();

      }

    }

    symmetricPair(arr);

  }

}

# Find All Common Element in All Rows of Matrix

import java.util.**\***;

public class Source {

  public static void printElementInAllRows(int *mat*[][]) {

*// Write code here*

    Map<Integer, Integer> map = new HashMap<>(); *//Creating a hashmap.*

    for (int i = 0; i < *mat*[0].length; i++) { *//Iterating through first row.*

      map.put(*mat*[0][i], 1); *//Storing first row elements as key and value as it's count.*

    }

    for (int i = 1; i < *mat*.length; i++) { *//Loop to traverse rows.*

      for (int j = 0; j < *mat*[0].length; j++) { *//Loop to traverse columns.*

        if (map.get(*mat*[i][j]) != null && map.get(*mat*[i][j]) == i) { *//If element is not in map is in the iterated row,*

          map.put(*mat*[i][j], i + 1); *//Increment count of that element by 1.*

        }

      }

    }

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

    for (Map.Entry<Integer, Integer> entry : map.entrySet()) {

      if (entry.getValue() == *mat*.length) {

        result.add(entry.getKey());

      }

    }

    Collections.sort(result);

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

      System.out.print(result.get(i) + " ");

    }

  }

  public static void main(String[] *args*) {

    Scanner sc = new Scanner(System.in);

    int row = sc.nextInt();

    int col = sc.nextInt();

    int matrix[][] = new int[row][col];

    for (int i = 0; i < row; i++) {

      for (int j = 0; j < col; j++) {

        matrix[i][j] = sc.nextInt();

      }

    }

    printElementInAllRows(matrix);

  }

}

# Find Itinerary in Order

import java.util.**\***;

public class Source {

  public static void findItinerary(Map<String, String> *tickets*) {

*// Write code here*

    Map<String, String> reverse = new HashMap<String, String>(); *//Creating another hashmap.*

    for (Map.Entry<String, String> entry : *tickets*.entrySet()) { *//Iterating through tickets hashmap,*

      reverse.put(entry.getValue(), entry.getKey()); *//and storing reverse order by putting value as key and key as value.*

    }

    String origin = ""; *//Creating string to store origin.*

    for (Map.Entry<String, String> entry : *tickets*.entrySet()) { *//Iterating through tickets hashmap again.*

      if (!reverse.containsKey(entry.getKey())) { *//If origin is not any other destination,*

        origin = entry.getKey(); *//we set origin as the starting point.*

        break;

      }

    }

    if (origin == null) { *//If there is no origin,*

      System.out.println("Invalid Input"); *//list is cyclic and origin can not be found.*

      return;

    }

    String destination = *tickets*.get(origin); *//Get destination for origin and store it in variable.*

    while (destination != null) { *//While loop to loop until there are no more destinations.*

      System.out.println(origin + "->" + destination); *//Printing this line of the Itinerary*

      origin = destination; *//Setting origin to destination, to find next ticket.*

      destination = *tickets*.get(destination); *//Updating destination from original tickets Hashmap.*

    }

  }

  public static void main(String[] *args*) {

    Map<String, String> tickets = new HashMap<String, String>();

    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt();

    for (int i = 0; i < n; i++) {

      tickets.put(sc.next(), sc.next());

    }

    findItinerary(tickets);

  }

}