**C3 - DSA Assignment**

### **Most Frequent Element**

**Description**

Given an array of integers, find the most frequent element in the array. If multiple elements appear a maximum number of times, print any one of them. Return -1 in case there are no elements.

**Input Format:**

The first line contains the size of the array, **N**.

The second line contains the elements of the array**.**

**Output Format:**

The output contains the most frequent element in the array.

**Sample Test Cases:**

**Input:**

8

1 3 2 5 2 1 9 2

**Output:**

2

**Input:**

8

10 15 12 11 13 12 11 11

**Output:**

11

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

public static int mostFrequentElement(int[] arr) {

if (arr.length == 0) {

return -1; // Return -1 if the array is empty

}

// Create a HashMap to store the frequency of each element

Map<Integer, Integer> frequencyMap = new HashMap<>();

// Iterate through the array and count the frequency of each element

for (int num : arr) {

frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);

}

// Find the most frequent element

int mostFrequent = arr[0];

int maxFrequency = frequencyMap.get(arr[0]);

for (int num : arr) {

if (frequencyMap.get(num) > maxFrequency) {

mostFrequent = num;

maxFrequency = frequencyMap.get(num);

}

}

return mostFrequent;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int 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 if an Undirected Graph is a Tree or Not**

**Description**

Given a graph with N vertices. Check whether an undirected graph is a tree or not.

An undirected graph is tree if it has following properties.

1. There is no cycle.
2. The graph is connected.

Another way to understand this is to say that a tree is an undirected graph in which any two vertices are connected by exactly one path.

**Input Format**:

The first line contains an integer representing a number **N**as the number of vertices of the graph.

The second contains an integer **n** representing the number of edges in the graph.

Each next **n** line will have two space-separated integers, representing two vertices of each edge.

For example:

1 2 represents an edge from node 1 to node 2.

**Output Format:**

Print ‘Yes’ (without quotes and with capital letter ‘Y’) if the graph is a tree, otherwise ‘No’ (without quotes and with capital letter ‘N’).

**Sample Test Cases:**

**Input:**

15

4

11 10

10 12

10 13

13 14

**Output:**

No

**Input:**

6

5

1 0

0 2

0 3

3 4

4 5

**Output:**

Yes

**Execution time limit**

5 seconds

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) {

return i >= 0 && i < vertexCount;

}

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

visited[v] = true;

// Recur for all vertices adjacent to this vertex

Iterator<Integer> iterator = adj[v].listIterator();

while (iterator.hasNext()) {

int adjacent = iterator.next();

// If an adjacent vertex is not visited yet, then recur for it

if (!visited[adjacent]) {

if (isCyclic(adjacent, visited, v)) {

return true;

}

}

// If an adjacent vertex is visited and not the parent of the current vertex,

// then there is a cycle in the graph

else if (adjacent != parent) {

return true;

}

}

return false;

}

public boolean isTree() {

// Mark all the vertices as not visited and not part of the recursion stack

boolean visited[] = new boolean[vertexCount];

Arrays.fill(visited, false);

// Call the recursive helper function to detect cycle in different DFS trees

if (isCyclic(0, visited, -1)) {

return false;

}

// Check if all vertices are visited after DFS traversal. If not, graph is disconnected

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

if (!visited[i]) {

return false;

}

}

return true;

}

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");

} else {

System.out.println("No");

}

}

}

### **Find kth Largest Element in a Stream**

**Description**

Given an infinite stream of integers, find the kth largest element at any point in time.

**Input Format:**

The first line contains an integer ‘**N**’ as the size of the stream.

The second line contains an integer ‘**K’** representing the position of the largest number.

The third line contains elements of the stream of size ‘**N**’.

**Output Format:**

The output contains the kth largest number at any point in time. If the kth largest number does not exist, then print ‘None’ (without quotes and with capital letter ‘N’).

**Sample Test Cases:**

**Input:**

8

3

20 30 21 80 60 50 110 15

**Output:**

None

None

3 largest number is 20

3 largest number is 21

3 largest number is 30

3 largest number is 50

3 largest number is 60

3 largest number is 60

**Input:**

5

2

10 40 11 20 60

**Output:**

None

2 largest number is 10

2 largest number is 11

2 largest number is 20

2 largest number is 40

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int k = sc.nextInt();

int stream[] = new int[n];

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

stream[i] = sc.nextInt();

int kthLargest = findKthLargest(stream, i + 1, k);

if (kthLargest == Integer.MIN\_VALUE) {

System.out.println("None");

} else {

System.out.println(k + " largest number is " + kthLargest);

}

}

}

public static int findKthLargest(int[] stream, int size, int k) {

if (k > size) {

return Integer.MIN\_VALUE;

}

PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);

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

if (minHeap.size() < k) {

minHeap.offer(stream[i]);

} else if (stream[i] > minHeap.peek()) {

minHeap.poll();

minHeap.offer(stream[i]);

}

}

return minHeap.size() == k ? minHeap.peek() : Integer.MIN\_VALUE;

}

}

### **Sort Nearly Sorted Array**

**Description**

Given a k-sorted array of n elements, where each element is at most k steps away from its target position as it would have been in an array that is sorted in ascending order. Write a program to sort the array in O(n log k) time.

For example, an element at index i in an array that was sorted in ascending order can be found at indexes i - 3, i - 2, i - 1, i, i + 1, i + 2 and i + 3 in the given k-sorted array.

**Input Format**:

The first line contains an integer ‘**N**’ as the size of the array.

The second line contains an integer ‘**K’** representing the maximum number of steps that each element can deviate from its target position as it would have been in an array that is sorted in ascending order.

The third line contains the elements of the k-sorted array.

**Output Format:**

The output contains the elements of the array that is sorted in ascending order.

**Sample Test Cases:**

**Input:**

7

3

7 6 4 3 9 11 10

**Output:**

3 4 6 7 9 10 11

**Input:**

7

3

6 5 3 2 8 10 9

**Output:**

2 3 5 6 8 9 10

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

private static void sortArray(int[] arr, int k) {

PriorityQueue<Integer> minHeap = new PriorityQueue<>(k + 1);

// Insert the first k + 1 elements into the min-heap

for (int i = 0; i <= k && i < arr.length; i++) {

minHeap.offer(arr[i]);

}

int index = 0;

// Process the remaining elements of the array

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

arr[index++] = minHeap.poll(); // Remove the minimum element and assign it to the sorted array

minHeap.offer(arr[i]); // Add the next element from the array to the min-heap

}

// Extract remaining elements from the min-heap and assign them to the sorted array

while (!minHeap.isEmpty()) {

arr[index++] = minHeap.poll();

}

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int k = sc.nextInt();

int arr[] = new int[n];

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

arr[i] = sc.nextInt();

}

sortArray(arr, k);

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

System.out.print(arr[i] + " ");

}

}

}

### **Find Sum Between pth and qth Smallest Element**

**Description**

Given an array and two numbers p and q, find the sum of all the elements of the array such that the range of the elements is between the pth and qth smallest elements of the array.

It may be assumed that (1 <= p < q <= n) and all elements of the array are distinct.

**Input Format**:

The first line contains an integer ‘**N**’ as the size of the array.

The second line contains the elements of an array of size ‘**N**’.

The third line contains an integer ‘**p’**.

The fourth line contains an integer ‘**q’**.

**Output Format:**

The output contains the sum of all the elements of the array such that the range of the elements is between the **p**thand **qth**smallest elements of the array.

**Sample Test Cases:**

**Input:**

7

30 18 32 14 22 20 24

3

6

**Output:**

46

**Input:**

8

12 20 10 30 15 11 16 20

2

7

**Output:**

63

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

public static int sumBetweenPthToQthSmallestElement(int[] arr, int p, int q) {

// Sort the array

Arrays.sort(arr);

// Find the pth and qth smallest elements

int pthSmallest = arr[p - 1];

int qthSmallest = arr[q - 1];

// Initialize sum

int sum = 0;

// Iterate through the array and sum up elements between pth and qth smallest

for (int num : arr) {

if (num > pthSmallest && num < qthSmallest) {

sum += num;

}

}

return sum;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int arr[] = new int[n];

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

arr[i] = sc.nextInt();

}

int p = sc.nextInt();

int q = sc.nextInt();

System.out.println(sumBetweenPthToQthSmallestElement(arr, p, q));

}

}

### **Find All Symmetric Pairs in an Array**

**Description**

Given an array of pairs of integers, find all the symmetric pairs in it. Two pairs (a, b) and (c, d) are said to be symmetric if b is equivalent to c and a is equivalent to d.

For example, (10, 20) and (20, 10) are symmetric. It may be assumed that the first element in each pair is distinct.

Note that in the output, you need to print only the first occurring pair out of the two symmetric pairs in the given array. For example, if (a, b) and (c, d) are symmetric, you will only print the one that occurs first in the given array.

**Input Format**:

The first line contains an integer representing the number of pairs.

The second line contains the elements of each pair. Each pair has two integers’ so, this line contains (number of pairs \* 2) elements.

**Output Format:**

The output contains the first occurring pair in each group of symmetric pairs in each line.

**Sample Test Cases:**

**Input:**

5

31 30 40 50 15 20 50 40 20 15

**Output:**

40 50

15 20

**Input:**

4

20 40 50 11 10 50 40 20

**Output:**

20 40

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

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

// Create a hash map to store pairs

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

// Traverse through the array of pairs

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

int first = arr[i][0];

int second = arr[i][1];

// Check if the second element of the current pair is present in the map

if (map.containsKey(second) && map.get(second) == first) {

// If found, print the symmetric pair

System.out.println(second + " " + first);

} else {

// Otherwise, add the current pair to the map

map.put(first, second);

}

}

}

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 Each Row of Matrix**

**Description**

Given an M x N matrix of integers, find all the common elements that are present in all rows of the given matrix in O(M \* N) time.

For example, take a look at the matrix given below.

11 12 11 14 18

13 17 18 15 11

18 17 17 13 11

18 11 12 17 19

In this matrix, 11 and 18 appear in each row. Hence, 11 and 18 will be the common elements that are present in each row.

**Note that if multiple common elements are present in each row, you need to print them in ascending order.**

**Input Format:**

The first line contains two integers **M**as the number of rows and **N** as the number of columns.

From the next line, there are **N**elements in a line up to **M** lines, representing the elements of the matrix.

**Output Format:**

The output contains all the common elements that are present in each row of the given matrix, **printed in ascending order.**

**Sample Test Cases:**

**Input:**

4 5

11 12 11 14 18

13 17 18 15 11

18 17 17 13 11

18 11 12 17 19

**Output:**

11 18

**Input:**

4 4

17 12 11 10

13 17 10 15

10 17 17 13

12 11 10 17

**Output:**

10 17

**Execution time limit**

5 seconds

import java.util.HashMap;

import java.util.HashSet;

import java.util.Map;

import java.util.Scanner;

import java.util.Set;

import java.util.TreeSet;

public class Source {

private static void getCommonElementsInAllRows(int[][] inputMatrix) {

int rows = inputMatrix.length;

int columns = inputMatrix[0].length;

Map<Integer, Integer> elementCountMap = new HashMap<>();

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

elementCountMap.put(inputMatrix[0][j], 1);

}

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

Set<Integer> currentRowElements = new HashSet<>();

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

currentRowElements.add(inputMatrix[i][j]);

}

for (int element : currentRowElements) {

if (elementCountMap.containsKey(element)) {

elementCountMap.put(element, elementCountMap.get(element) + 1);

}

}

}

TreeSet<Integer> commonElements = new TreeSet<>();

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

if (entry.getValue() == rows) {

commonElements.add(entry.getKey());

}

}

for (int element : commonElements) {

System.out.print(element + " ");

}

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int rows = scanner.nextInt();

int columns = scanner.nextInt();

int[][] matrix = new int[rows][columns];

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

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

matrix[i][j] = scanner.nextInt();

}

}

getCommonElementsInAllRows(matrix);

scanner.close();

}

}

### **Find Itinerary in Order**

**Description**

Given a list of tickets in pairs as source city and destination city, find the itinerary in order using the given list.

Note that the given list of tickets is non-cyclic and there is one ticket from each city except the destination city.

**Input Format:**

The first line contains an integer **N**representing the number of tickets.

For the next **N**lines**,** each line contains a ticket representing the source city and the destination city.

**Output Format:**

Print ‘Invalid Input’ if the start city is **null,** else print city in an itinerary in order, a pair in each line as shown below:

Bangalore->Ranchi

Ranchi->Jamshedpur

**Sample Test Cases:**

**Input:**

4

Bangalore Ranchi

Jamshedpur Kolkata

Ranchi Jamshedpur

Kolkata Varanasi

**Output:**

Bangalore->Ranchi

Ranchi->Jamshedpur

Jamshedpur->Kolkata

Kolkata->Varanasi

**Input:**

4

Chennai Bangalore

Bombay Delhi

Goa Chennai

Delhi Goa

**Output:**

Bombay->Delhi

Delhi->Goa

Goa->Chennai

Chennai->Bangalore

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

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

// Write code here

String startCity=null;

for (String city : tickets.keySet()) {

if (!tickets.containsValue(city)) {

startCity = city;

break;

}

}

if (startCity == null) {

System.out.println("Invalid Input");

return;

}

// Print the itinerary in order

while (startCity != null ) {

String destination = tickets.get(startCity);

if(destination!=null){

System.out.println(startCity + "->" + destination);

}

startCity = destination;

}

}

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);

}

}

### **Search Element in a Rotated Array**

**Description**

Suppose you rotate an array of integers that is sorted in increasing order, at an unknown pivot. Check whether the given key is present in the array or not. Construct a way to find an element in the given array in O(log n) time, where n is the number of elements in the array.

For example, 14 15 16 17 18 19 11 12 13 is an array that is sorted in increasing order and rotated at 11.

**Input Format:**

The first line contains an integer **N**as the size of the array.

The second line contains elements of the array.

The third line contains the key to be searched in the given array.

**Output Format:**

Print the index of an array if the key is found in the array, else print **-1.**

**Sample Test Cases:**

**Input:**

9

14 15 16 17 18 19 11 12 13

6

**Output:**

-1

**Input:**

9

14 15 16 17 18 19 11 12 13

11

**Output:**

6

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

public static int search(int arr[], int left, int right, int key) {

// Write code here

if (left > right) {

return -1;

}

int mid = (left + right) / 2;

if (arr[mid] == key) {

return mid;

}

if (arr[left] <= arr[mid]) {

if (key >= arr[left] && key <= arr[mid]) {

return search(arr, left, mid - 1, key);

}

return search(arr, mid + 1, right, key);

}

if (key >= arr[mid] && key <= arr[right]) {

return search(arr, mid + 1, right, key);

}

return search(arr, left, mid - 1, key);

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int arr[] = new int[n];

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

arr[i] = sc.nextInt();

}

int key = sc.nextInt();

int i = search(arr, 0, n - 1, key);

if (i != -1) {

System.out.println(i);

} else {

System.out.println("-1");

}

}

}

### **Find Median After Merging Two Arrays**

**Description**

Write a program to find the median of the array obtained after merging the two given **sorted** integer arrays of the same length. **Note that both the arrays are sorted.**

**Input Format:**

The first line contains an integer **N**as the size of both the arrays.

The second line contains elements of the first array of size **N**.

The next line contains elements of the second array of size **N.**

**Output Format:**

Print the median of the array obtained after merging the two given integer arrays.

**Sample Test Cases:**

**Input:**

5

1 12 15 26 38

2 13 17 30 45

**Output:**

16

**Execution time limit**

5 seconds

import java.util.\*;

public class Source {

public static int median(int[] arr1, int[] arr2 , int n){

// Write code here

int i = 0, j = 0;

int m1 = -1, m2 = -1;

int count;

for (count = 0; count <= n; count++) {

if (i == n) {

m1 = m2;

m2 = arr2[0];

break;

}

else if (j == n) {

m1 = m2;

m2 = arr1[0];

break;

}

if (arr1[i] < arr2[j]) {

m1 = m2;

m2 = arr1[i];

i++;

}

else{

m1 = m2;

m2 = arr2[j];

j++;

}

}

return (m1 + m2) / 2;

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

int arr1[] = new int[n];

int arr2[] = new int[n];

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

arr1[i] = sc.nextInt();

}

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

arr2[i] = sc.nextInt();

}

System.out.println(median(arr1, arr2, n));

}

}