CodeQuotient

Department of Computer Science & Engineering

LAB MANUAL

22CS025 - Algorithm Design & Implementation-5Sem-2023

Student Name	**PRANSHU SHARMA**
Roll Number	
Course Name	**22CS025 - Algorithm Design & Implementation-5Sem-2023**

Faculty Incharge

Table of Contents

39	Print binary tree with level order traversal	Incorrect Answer
40	Print nodes at odd levels of the binary tree	Not Attempted
41	Write iterative version of inorder traversal	Not Attempted
42	Complete the inorder(), preorder() and postorder() functions for traversal with recursion	45 - 46
43	Construct tree from given inorder and post order traversal	Not Attempted
44	Count the number of leaf and non-leaf nodes in a binary tree	Not Attempted
45	Print all paths to leaves and their details of a binary tree	Not Attempted
46	Find the right node of a given node	47 - 48
47	Convert a binary tree into its mirror tree	Not Attempted
48	Print cousins of a given node in Binary Tree	49 - 50
49	Print nodes in a top view of Binary Tree	Not Attempted
50	Find out if the tree can be folded or not	51 - 52
51	Given two trees are identical or not	53 - 54
52	Find maximum depth or height of a binary tree	55 - 56
53	Evaluation of expression tree	Not Attempted
54	Given binary tree is binary search tree or not	57 - 58
55	Find the kth smallest element in the binary search tree	Incorrect Answer
56	Convert Level Order Traversal to BST	Incorrect Answer

Aim: Factorial using recursion

Write a recursive function **factorial** that accepts an integer *n* as a parameter and returns the factorial of *n*, or *n!*.
A factorial of an integer is defined as the product of all integers from 1 through that integer inclusive. For example, the call of **factorial(4)** should return 1 * 2 * 3 * 4, or 24. The factorial of 0 and 1 are defined to be 1.
You may assume that the value passed is non-negative and that its factorial can fit in the range of type int.
Input Format:
The first line of input contains number of testcases , T.
Then T lines follow, which contains an integer,n.
Output Format:
For each testcase print the factorial in new line
Sample Input

```
2
4
3
```

Sample Output

```
24
6
```

Solution:

```
class Result{
  /*
   * Complete the function 'factorial' given below
   * @params
   *  n -> an integer whose factorial is to be calculated
   * @return
   *  The factorial of integer n
   */
  static int factorial(int n) {
    // Write your code here
       if (n <= 1) return 1;
    return n * factorial(n - 1);
  }
}
```

Aim: Prime factors using recursion

Write a program that accepts an integer *n* (where n >=2) and print all the prime factors of n using recursion.

Sample Input

24

Sample Output

```
2
2
2
3
```

Constraints : Do not declare any global variables. Do not use any loops; you must use recursion. You can declare as many primitive variables like ints as you like. You are allowed to define other "helper" functions if you like; they are subject to these same constraints.

Solution:

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        printPrimeFactors(n, 2);   // Start checking from the smallest prime
    }
    // Recursively prints prime factors of n
    public static void printPrimeFactors(int n, int i) {
        if (n == 1) return;  // Base case: fully factorized
        if (n % i == 0) {    // i is a factor
            System.out.println(i);
            printPrimeFactors(n / i, i);   // Keep dividing by same factor
        } else {
            printPrimeFactors(n, i + 1);   // Try next number
        }
    }
}
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: power(base, exp)

Write a recursive function **power** that accepts two integers representing a *base* and an *exponent*, and returns the base raised to that exponent. For example, the call to power(3, 4) should return 3^4 i.e. 81. If the exponent passed is negative, then return -1.
Do not use loops or auxiliary data structures; Solve the problem recursively. Also do not use the provided library pow function in your solution.
Expected Time Complexity: O(log(n)); *here n denotes the exponent*
Input Format:
The first line of input conatins an integer T, denoting the number of test cases.
The second line of input contains 2 integers base and exponent seperated by space.
Output Format:
Print the answer when base is raised to the exponent.
Constraints:
-10 <= base <= 10
þÿ-15 <= exponent <= 15
Sample Input
2 // Test Cases
2 3
5 2
Sample Output
8
25

Solution:

```
class Result {
    static long power(int base, int exp) {
        if (exp < 0) return -1;     // negative exponent case
        if (exp == 0) return 1;     // base case
        long half = power(base, exp / 2);
        if (exp % 2 == 0)
            return half * half;
        else
            return base * half * half;
    }
}
```

Aim: Binary equivalent using recursion

Write a recursive function **decimalToBinary** that accepts an integer as a parameter and returns an integer whose digits look like that number's representation in binary (base-2). For example, the call of decimalToBinary(43) should return 101011.
Constraints: Do not use a string in your solution. Also do not use any built-in base conversion functions from the system libraries. solve the problem recursively.
Sample Input :
1 // no. of testcases
43
Sample Output :
101011

Solution:

```
class Result {
    static int decimalToBinary(int n) {
        // Base case
        if (n == 0) {
            return 0;
        }
        // Recursive call
        return decimalToBinary(n / 2) * 10 + (n % 2);
    }
}
```

Aim: Greatest common divisor using recursion

Write a recursive function **gcd** that accepts two positive non-zero integer parameters *i* and *j* and returns the greatest common divisor of these numbers.

Sample Input

```
2 // Test Cases
30 18  // i j (testcase 1)
11 17  // i j (testcase 2)þÿ
```

Sample Output

```
6
1
```

Constraints:. Solve the problem recursively.

Solution:

```
class Result
{
  static int gcd(int i, int j)
  {
        if(j==0)
            return i;
      return gcd(j,i%j);
  }
}
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: Find all pairs with sum K

Given a sorted list of **N** integers, find all distinct pairs of integers in the list with sum equal to a given number **K**, with O(n log n) or O(n) time complexity.
Complete the function below which takes the array and K as parameters and return the number of pairs if any and 0 otherwise.
Input Format:
First line of input will contain a positive integer T = number of test cases. Each test case will contain 2 lines.
First line of each test case contains two integers - N and K.
Next line will contain N numbers separated by space in non-decreasing order.
Constraints:
1 "d T "d 10
1 "d N "d 10^5
-(10^9) "d arr[i], K "d 10^9
Output Format:
For each test case, print number of distinct pairs whose sum will be equal to k. A pair must have two numbers at different indices.
Two pairs are different if at least one of the indices in them is uncommon. Indices - (2,3) and (3,2) are not distinct, but (2,3) and (2,4) are.
Sample Input

```
3  // Test Cases
10 11  // N K (testcase 1)
1 2 3 4 5 6 7 8 9 10
5 10   // N K (testcase 2)
2 4 6 8 10
6 27   // N K (testcase 3)
12 15 20 22 34 36
```

Sample Output

```
5
2
1
```

Solution:

```
import java.util.HashMap;
class Result {
    static int getPairsCount(int arr[], int n, int k) {
        HashMap<Integer, Integer> map = new HashMap<>();
        int count = 0;
        for (int i = 0; i < n; i++) {
            int x = arr[i];
            count += map.getOrDefault(k - x, 0);
            map.put(x, map.getOrDefault(x, 0) + 1);
        }
        return count;
    }
}
```

Aim: Find first occurrence of an integer in a sorted list with duplicates

Given a sorted list of integers, find the position of first occurrence of a given number **K** in the list in O(log n) time.

Input Format:

First line of input will contain a positive integer T = number of test cases.
Each test case will contain the following two lines:
  First line will contain two positive integer N = number of elements in list and K.
  Second line will contain N space separated integers in increasing order.

Constraints:

  1 <= N <= 10^5
 -(10^9) <= arr[i], K <= (10^9)

Output Format:

For each test case, print on a single line the index of first occurrence of K in the list on 0-based index. Print -1 if you cannot find K in the list.

Sample Input

```
3  // Test Cases
10 4  // N K (testcase 1)
1 2 4 4 4 4 5 8 9 10
15 7  // N K (testcase 2)
1 2 3 3 5 6 7 7 7 7 7 8 8 8 8
9 1   // N K (testcase 3)
-5 -4 -3 -2 -1 0 0 0 1
```

Sample Output

```
2
6
8
```

Solution:

```
import java.util.Scanner;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt(); // number of test cases
        while (T-- > 0) {
            int N = sc.nextInt();
            int K = sc.nextInt();
            int[] arr = new int[N];
            for (int i = 0; i < N; i++) {
                arr[i] = sc.nextInt();
            }
            int left = 0, right = N - 1;
            int answer = -1;
            while (left <= right) {
                int mid = left + (right - left) / 2;
                if (arr[mid] == K) {
                    answer = mid;      // possible answer
                    right = mid - 1;   // search left part
                } else if (arr[mid] < K) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
            System.out.println(answer);
        }
        sc.close();
    }
}
```

Aim: Rotation count of a sorted Array

Given an array of integers, find the minimum number of rotations performed on some sorted array to create this given array.
Input
First line of input will contain a number T = number of test cases. Each test case will contain 2 lines. First line will contain a number N = number of elements in the array (1 <= N <= 50). Next line will contain N space separated integers.
Complete the function **int rotationCount(int array[], int size)** which will receive array and size of the array as input and return how many times the sorted array is rotated. Function should return -1 if the array is not rotated.
Output
Print one line of output for each test case with the minimum number of rotations for given array.
Sample Input:

```
2
6
15 18 3 4 6 12
6
1 2 3 4 5 6
```

Sample Output

```
2
-1
```

Solution:

```
import java.util.Scanner;
class Main {
    static int rotationCount(int[] arr, int n) {
        // If already sorted (not rotated)
        if (arr[0] <= arr[n - 1]) return -1;
        int low = 0, high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            // Check if mid is minimum
            if (mid > 0 && arr[mid] < arr[mid - 1])
                return mid;
            // Decide search space
            if (arr[mid] >= arr[0])
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int T = sc.nextInt();
        while (T-- > 0) {
            int N = sc.nextInt();
            int[] arr = new int[N];
            for (int i = 0; i < N; i++)
                arr[i] = sc.nextInt();
            System.out.println(rotationCount(arr, N));
        }
        sc.close();
    }
}
```

Aim: Search element in a rotated sorted array

Given an array of **n** integers which is sorted but rotated by some number of times after sorting, and a integer **k**. Search the element k in this sorted rotated array efficiently. For example, suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. (i.e., 0 1 2 4 5 6 7 might become 4 5 6 7 0 1 2).
Assume there are no duplicate elements in the array.
Expected Time Complexity: O(log(N))
Expected Space Complexity: O(1)
Input Format
First line of input will contain a number T = number of test cases. Each test case will contain 3 lines. The first line will contain an integer k to be searched. Second line will contain a number n = number of elements in the array. Next line will contain N space separated integers.
Complete the function int searchRotatedSortedArray() to search a value target in array a of size given, and if target found in the array return its index, otherwise return -1.
You may assume no duplicate exists in the array.
Output Format
Print the index of k in given array for each test case in new line if found and print -1 if k is not present in given array.
Constraints
 1 <= T <= 10
-1000 <= k <= 1000
 1 <= n <= 10^5
-1000 <= arr[i] <= 1000
Sample Input

```
2   // Test Cases
3            // k (testcase 1)
6            // n
15 18 2 3 6 12  // arr[]
7            // k (testcase 2)
7            // n
4 5 8 9 1 2 3   // arr[]
```

Sample Output

```
3
-1
```

Solution:

```
class Result {
    static int searchRotatedSortedArray(int arr[], int k) {
        int low = 0;
        int high = arr.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            // Element found
            if (arr[mid] == k) {
                return mid;
            }
            // Left half is sorted
            if (arr[low] <= arr[mid]) {
                if (k >= arr[low] && k < arr[mid]) {
                    high = mid - 1;
                } else {
                    low = mid + 1;
                }
            }
            // Right half is sorted
            else {
                if (k > arr[mid] && k <= arr[high]) {
                    low = mid + 1;
                } else {
                    high = mid - 1;
                }
            }
        }
        return -1; // Not found
    }
}
```

Aim: Reverse the order of words of a string

Given a string of words, reverse the order of words in the string individually, not the whole string.
Complete the function **revWordsOrder(str)** that accepts a string as parameter and reverses the order of words of string.
Input Format:
The first line of input contains an integer T denoting the no of test cases . Then T test cases follow. Each test case contains the string str.
Output Format:
For each test case, print the resultant string in new lines.
Sample Input

```
2
Code Quotient Loves Code
Hello Coders
```

Sample Output

```
Code Loves Quotient Code
Coders Hello
```

Solution:

```
class Result {
static String revWordsOrder(String str) {
    List<String> words = Arrays.asList(str.split(" "));
    Collections.reverse(words);
    return String.join(" ", words);
}
}
```

Aim: String is subsequence or not

Given two strings, find whether second string is subsequence of first string. A subsequence of a string is defined as a string that can be obtained by deleting any number of characters from the original string.
Complete the function **strSubsequence(str1, str2)** that accepts two strings as parameters and print **YES** if str2 is a subsequence of str1 and **NO** otherwise.
Input Format
The first line of input contains an integer T denoting the no of test cases . Then T test cases follow.
Each test case contains a single line of input which contains two strings str1, str2 seperated by a space.
Output Format
For each test case, print YES or NO in new lines.
Constraints
1 <= T <= 10
Given strings consist of uppercase and lowercase English letters.
Sample Input
```
3
CodeQuotient CQ
CodeQuotient QC
Hello Hi
```
Sample Output
```
YES
NO
NO
```

Solution:

```
class Result {
static boolean strSubsequence(String s1, String s2) {
   int j = 0;
   for (int i = 0; i < s1.length() && j < s2.length(); i++)
       if (s1.charAt(i) == s2.charAt(j)) j++;
   return j == s2.length();
}
}
```

Aim: Technical Issue With The Keyboard

Aman likes to play with strings, so his friend Riya decided to send him a mail, containing a sorted string with unique characters. But, due to some technical issue with her keyboard, whenever she presses a key, the corresponding character gets printed multiple times. Now Riya needs your help for removing all the unwanted characters from her string. For example, If **aabbbccdef** is a sorted string generated by Riya, then you should output the following string as answer **abcdef**.
Complete the function **getDesiredString(str)** that will take the string as parameter and modify it as specified.
Input Format:
The first line of input contains an integer T denoting the no of test cases . Then T test cases follow. Each test case contains a string str.
Output Format:
For each test case, print the new string in new lines.
Constraints:
1 <= T <= 10
1 <= length of string <= 10^5
Each string will contain only lower-case English letters.þÿ
Sample Input
2
aabbccdef
abddddddd
Sample Output
abcdef
abd

Solution:

```
class Result {
static String getDesiredString(String s) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < s.length(); i++)
        if (i == 0 || s.charAt(i) != s.charAt(i - 1))
            sb.append(s.charAt(i));
    return sb.toString();
}
}
```

Aim: Implement strcat function

Implement the strcat() function from string library as your own function. The function will take two strings as arguments and concatenate the second string at the end of first string.
Input Format:
The first line of input contains an integer T denoting the no of test cases .
Then T test cases follow. Each test case contains two strings str2 and str1.
Function strcatCode(str1,str2) will take two parameters and concatenate the str2 string at end of str1.
Output Format:
For each test case, print the concatenated string in new lines.
Sample Input

```
1
Code Quotient
```

Sample Output

```
CodeQuotient
```

Solution:

```
static String strcatCode(String a, String b) {
    return a + b;
}
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: Count words

Write a function **countWords()** to count the numbers of words in a string.
A word is defined as text separated by space(' ') or multiple spaces.
The function will receive a string as input and return the numbers of words in this string.
Input Format
A single line of input which consists of the string whose words is to be counted
Output Format
Print the count the numbers of words in a string
Sample Input
Codequotient get better at coding
Sample Output
5

Solution:

```
class Result {
    static int countWords(String str) {
        int count = 0;
        int n = str.length();
        for (int i = 0; i < n; i++) {
            // Start of a word
            if (str.charAt(i) != ' ' &&
               (i == 0 || str.charAt(i - 1) == ' ')) {
                count++;
            }
        }
        return count;
    }
}
```

Aim: Spell the number

Given a non-negative integer between 0 and 9,99,999 print the english phrase corresponding to it i.e. 999999 is “nine lakhs ninety nine thousand nine hundred ninety nine”.
þÿInput Format:
The first line of input contains an integer T denoting the no of test cases . Then T test cases follow. Each test case contains one integer number.
Function void intToWord(int num) will take the number as parameter and print the english phrase for it in small letters separated by space.
Output Format:
For each test case, print the english phrase in new lines.
Sample Input

```
2  // No. of test cases
31
12345
```

Sample Output

```
thirty one
twelve thousand three hundred forty five
```

Solution:

```
class Result {
    static String[] o = {
        "", "one", "two", "three", "four", "five", "six",
        "seven", "eight", "nine", "ten", "eleven", "twelve",
        "thirteen", "fourteen", "fifteen", "sixteen",
        "seventeen", "eighteen", "nineteen"
    };
    static String[] t = {
        "", "", "twenty", "thirty", "forty",
        "fifty", "sixty", "seventy", "eighty", "ninety"
    };
    static void intToWord(int n) {
        if (n == 0) {
            System.out.print("zero");
            return;
        }
        if (n >= 100000) {
            int lakh = n / 100000;
            intToWord(lakh);
            System.out.print(lakh > 1 ? " lakhs" : " lakh");
            n %= 100000;
            if (n > 0) System.out.print(" ");
        }
        if (n >= 1000) {
            intToWord(n / 1000);
            System.out.print(" thousand");
            n %= 1000;
            if (n > 0) System.out.print(" ");
        }
        if (n >= 100) {
            intToWord(n / 100);
```

```
                System.out.print(" hundred");
                n %= 100;
                if (n > 0) System.out.print(" ");
            }
            if (n >= 20) {
                System.out.print(t[n / 10]);
                n %= 10;
                if (n > 0) System.out.print(" ");
            }
            if (n > 0) {
                System.out.print(o[n]);
            }
        }
    }
```

Aim: Check if strings are rotations or not

Given two strings, find whether both are rotations of each other or not. For example,
Given str1 = abacd and str2 = acdab, we can get str1 by rotating str2 and,
Given str1 = coder and str2 = cored, we can not get str1 by rotating str2.
þÿInput Format
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow. Each test case contains two strings str1 and str2.
Output Format
For each test case, print YES or NO in new lines.
Sample Input

```
2  // Test Cases
abacd  // testcase 1
acdab
coder  // testcase 2
cored
```

Sample Output

```
YES
NO
```

Solution:

```
class Result {
    // return 1 for YES and 0 for NO.
    static int isRotation(String str1, String str2) {
        // Lengths must be equal
        if (str1.length() != str2.length()) {
            return 0;
        }
        String temp = str1 + str1;
        if (temp.contains(str2)) {
            return 1;
        }
        return 0;
    }
}
```

Aim: First Unique Character

Given a string that contains only lowercase English letters, find the first non-repeating character in it and return its index. If it doesn't exist, then return -1.
Input Format:
First line contains a string.
Output Format:
Print the index of the first non-repeating character in the given string. If it doesn’t exist, then print -1.
Constraints:
‘a’ <= str[i] <= ‘z’
1 <= length of str <= 100000
Sample Input 1
codequotientchamp
Sample Output 1
2
Explanation 1
‘d’ is the first non-repeating character in the given string, therefore the output is its index i.e. 2.
Sample Input 2
silentlisten
Sample Output 2
-1
Explanation 2
All the characters in the given string are repeating, therefore the output is -1.

Solution:

```
class Result {
  static int firstUniqueChar(String str) {
    int[] freq = new int[26];
    // Step 1: Count frequency
    for (int i = 0; i < str.length(); i++) {
      freq[str.charAt(i) - 'a']++;
    }
    // Step 2: Find first unique character
    for (int i = 0; i < str.length(); i++) {
      if (freq[str.charAt(i) - 'a'] == 1) {
        return i;
      }
    }
    return -1; // No unique character found
  }
}
```

Aim: Find out the winner

Given an array of strings, find out the string which occurs maximum number of times. If two strings occurs maximum times, return the alphabetically later string. For example,
if array is ["Amit", "Girdhar", "Amit", "Girdhar", "Girdhar", "Amit", "Amit"] then return "Amit", and
if array is ["Amit", "Girdhar", "Amit", "Girdhar", "Girdhar", "Amit"] then return "Girdhar" as both occurs 3 times but Girdhar comes after Amit in alphabetical order.
So, write a function which accepts a string array as input and return the output string.

Input Format

The first line contains an integer n, the number of names in string array.
Each the n subsequent lines contains a string describing array[i] where 0 "d i < n.

Output Format

Print the output string

Constraints

1 "d n "d 10^5
1 "d length of string "d 64
string will contain only lowercase english alphabets i.e. from 'a' to 'z'

Sample Input

```
6
Amit
Girdhar
Amit
Girdhar
Girdhar
Amit
```

Sample Output

```
Girdhar
```

Solution:

```
import java.util.HashMap;
class Result {
    static String inspectStrings(String[] words) {
        HashMap<String, Integer> map = new HashMap<>();
        String winner = "";
        int maxCount = 0;
        for (int i = 0; i < words.length; i++) {
            String word = words[i];
            int count = map.getOrDefault(word, 0) + 1;
            map.put(word, count);
            // Higher frequency OR same frequency but lexicographically later
            if (count > maxCount ||
               (count == maxCount && word.compareTo(winner) > 0)) {
                maxCount = count;
                winner = word;
            }
        }
        return winner;
    }
}
```

Aim: Unique characters or not

Given a string, you need to test the characters for their uniqueness. If all the characters occur at most 1 time in the string, then print “YES”, otherwise if some character occurs at least twice in the string print “NO”.

Input Format:

The first line of input contains an integer T denoting the no of test cases . Then T test cases follow. Each test case contains the string str.

Function void isUniqueChars(char *str) will take the string as parameter and print YES or NO according to the occurence of characters in it.

Output Format:

For each test case, print YES or NO in new lines.

Constraints:

1 <= T <= 10

Given string can contain any valid ASCII character.

Sample Input

```
2
CodeQuotient
Coding
```

Sample Output

```
NO
YES
```

Solution:

```
class Result {
    // Return true if string contains all unique characters
    static boolean isUniqueChars(String str) {
        boolean[] visited = new boolean[256]; // ASCII size
        for (int i = 0; i < str.length(); i++) {
            int ch = str.charAt(i); // ASCII value
            if (visited[ch]) {
                return false; // duplicate found
            }
            visited[ch] = true;
        }
        return true;
    }
}
```

Aim: Count frequency of each character

Given a string that contains only lowercase characters. Write a program to print all the distinct characters along with their frequency in the order of their occurrence. For example if the string is “helloworld”, then you should print **h1 e1 l3 o2 w1 r1 d1**
Complete the given function **countFrequency()** and print the frequency of each character as per the given statement.
Input Format
First line contains a string with lowercase characters.
Constraints
'a' <= str[i] <= 'z'
1 <= length of str <= 100000
Output Format
Print all the distinct characters along with their frequency followed by a space. And the characters must be printed in the order of their occurrence.
Sample Input
codequotient
Sample Output
c1 o2 d1 e2 q1 u1 t2 i1 n1

Solution:

```
class Result {
  static void countFrequency(String str) {
    int[] freq = new int[26];
    // Count frequency
    for (int i = 0; i < str.length(); i++) {
      freq[str.charAt(i) - 'a']++;
    }
    // Print in order of occurrence
    boolean[] printed = new boolean[26];
    for (int i = 0; i < str.length(); i++) {
      int idx = str.charAt(i) - 'a';
      if (!printed[idx]) {
        System.out.print(str.charAt(i) + "" + freq[idx] + " ");
        printed[idx] = true;
      }
    }
  }
}
```

Aim: Check if two arrays are equal or not

Given two array of same size, find out if given arrays are equal or not.
Note: Two arrays are said to be equal if both of them contain same set of elements, although arrangements (or permutation) of elements may be different.
Note : If there are repetitions, then counts of repeated elements must also be same for two array to be equal..
For example, if A = [11, 12, 13] and B = [12, 11, 13] then answer is 1 and,
if A = [11, 12, 13, 12, 13] and B = [12, 11, 13, 12, 13] then answer is 1 and,
if A = [11, 12, 13] and B = [12, 13, 12] then answer is 0.
Complete the function **arraysEqualorNot()** given in the editor, which takes two arrays as input and returns the answer(0/1) as output.
Input Format
The 1st line contains an integer N, the number of elements in A and B.
The 2nd line contains N integers separated by space.
The 3rd line contains N integers separated by space.
Output Format
Print 1 or 0 as per the condition.
Constraints
 1 <= N <= 10^5
 -50000 <= A[i], B[i] <= 50000
Sample Input

```
3        // N
11 12 13  // A[]
12 11 13  // B[]
```

Sample Output

```
1
```

Solution:

```
import java.util.*;
class Result {
  static int arraysEqualorNot(int[] A, int[] B) {
    if (A.length != B.length)
      return 0;
    HashMap<Integer, Integer> map = new HashMap<>();
    // Count elements of A
    for (int x : A) {
      map.put(x, map.getOrDefault(x, 0) + 1);
    }
    // Subtract using elements of B
    for (int x : B) {
      if (!map.containsKey(x))
        return 0;
      map.put(x, map.get(x) - 1);
      if (map.get(x) < 0)
        return 0;
    }
    return 1;
  }
}
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: Largest subarray with 0 (ZERO) sum

Given an unsorted array of integers. Find the largest sub-array which adds to 0 (ZERO). For example,
Given **A = {11, 2, -2, 10, 1, -4, -7, 19}** , Print "**6**" as the sum 0 is found between indexes 1 and 6. Note that from index 1 to index 2 is also resulting in 0, but we need to print the largest.
If no sub-array found print **-1**.
Input Format
First Line will contain an integer N denoting the number of elements.
Second line contains N integers separated by space.
Output Format
Print the length of the largest sub-array as shown if found otherwise print -1.
Sample Input
8
11 2 -2 10 1 -4 -7 19
Sample Output
6

Solution:

```
import java.util.*;
class Result {
  static int largSubArray(int ar[], int n) {
    HashMap<Integer, Integer> map = new HashMap<>();
    int sum = 0;
    int maxLen = 0;
    for (int i = 0; i < n; i++) {
      sum += ar[i];
      // If sum becomes 0
      if (sum == 0) {
        maxLen = i + 1;
      }
      // If sum seen before
      if (map.containsKey(sum)) {
        maxLen = Math.max(maxLen, i - map.get(sum));
      } else {
        // Store first occurrence only
        map.put(sum, i);
      }
    }
    return maxLen == 0 ? -1 : maxLen;
  }
}
```

Aim: Find K largest elements in array

Given an array of integer elements and an integer k, print the k largest elements from array separated by space in descending sorted order. For example, if array a = [2, 56, 3, 87, 42, 1, 45, 8, 2, 98] and k=3 then output must be 98 87 56.
Input Format:
First line of input contain T = number of test cases. Each test case contain two lines, in first line, total number of elements in array and value of k are provided and next line will contain the elements.
The array of numbers and integer k is given to you.
Note: Do not write the whole program, just complete the functions void printKLargest(int array[], int k) which takes the array and integer k as parameters and print the k largest elements in descending order separated by space.
Note: Do not read any input from stdin/console. Just complete the functions provided. You can write more functions if required, but just above the given function.
Output Format:
Print the k largest element in descending order separated by space.
Do not print the space after last element.
Sample Input
```
2
10 3
2 56 3 87 42 1 45 8 2 98
6 2
1 87 2 67 3 45
```
Sample Output
```
98 87 56
87 67
```

Solution:

```
static void printKLargest(int array[], int n, int k) {
    Arrays.sort(array);
    for (int i = n - 1; i >= n - k; i--) {
        System.out.print(array[i]);
        if (i != n - k)
            System.out.print(" ");
    }
}
```

Aim: Convert min Heap to max Heap

Given array representation of a min-heap convert it in a max-heap representation. For example, if array a = [13 15 19 16 18 120 110 112 118] the array must be modified as a = [120 118 110 112 18 19 13 15 16].
The array of numbers is given to you. Do not write the whole program, just complete the functions **void modifyMintoMax(int array[], int n)** which takes the array and total number of elements n as parameters and modify the array elements with **heapify()** to convert it in max-heap.
Note: Do not read any input from stdin/console. Just complete the functions provided. You can write more functions if required, but just above the given function.
Input Format:
First line of input contain T = number of test cases.
Each test case contain two lines, in first line, total number of elements in array and next line will contain the elements.
Output Format:
For each test case, print the max-heap elements separated by space in new lines.
Sample Input

```
2
9
13 15 19 16 18 120 110 112 118
6
1 2 3 4 5 6
```

Sample Output

```
120 118 110 112 18 19 13 15 16
6 5 3 4 2 1
```

Solution:

```
// Heapify function for Max Heap
    static void heapify(int array[], int n, int i)
    {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        if (left < n && array[left] > array[largest])
            largest = left;
        if (right < n && array[right] > array[largest])
            largest = right;
        if (largest != i)
        {
            int temp = array[i];
            array[i] = array[largest];
            array[largest] = temp;
            heapify(array, n, largest);
        }
    }
    // Function to convert Min Heap to Max Heap
    static void modifyMintoMax(int array[], int n)
    {
        // Start from last non-leaf node
        for (int i = (n / 2) - 1; i >= 0; i--)
        {
            heapify(array, n, i);
        }
    }
```

Aim: Check if a given tree is max-heap or not

A max-heap is a kind of tree which must satisfy the property as “Every node’s value should be greater than its children’s value”.
Your task is given level wise array representation of a binary tree, check if it is a max-heap or not.
Complete the functions **isMaxHeap()** which takes the array and total number of elements n as parameters and return 1 if array represents a max-heap and 0 otherwise.
Input Format
First line of input contain T = number of test cases. Each test case contain two lines, in first line, total number of elements in tree and next line will contain the elements in level order fashion.
Output Format
For each test case, print 1 if array represents max-heap or 0 otherwise in new lines.
Sample Input

```
2
9
120 118 110 112 18 19 13 15 16
6
1 2 3 4 5 6
```

Sample Output

```
1
0
```

Explanation:
First tree is like below:

```
         120
        /   \
      118     110
     /   \    /  \
   112   18  19   13
   / \
  15   16
```

It satisfies the properties of max-heap.

Solution:

```
static int isMaxHeap(int array[], int n) {
        // Check all internal nodes
        for (int i = 0; i <= (n / 2) - 1; i++) {
            int left = 2 * i + 1;
            int right = 2 * i + 2;
            // If left child exists and is greater than parent
            if (left < n && array[i] < array[left]) {
                return 0;
            }
            // If right child exists and is greater than parent
            if (right < n && array[i] < array[right]) {
                return 0;
            }
        }
        return 1; // Valid Max Heap
    }
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: Connect the sticks of different lengths with minimum cost

Given n sticks of different length you have to connect them. The cost of connecting sticks is the sum of their length. You have to find the minimum cost for connecting all sticks. For example, if 3 sticks are there having length respectively 3, 6, 1 then we can connect them in many ways like:
Connect 3 and 6 (Cost 9), then we have 2 sticks of length 9 and 1 connect them (cost 10), So total cost is 19.
Connect 3 and 1 (Cost 4), then we have 2 sticks of length 4 and 6 connect them (cost 10), So total cost is 14.
So your function must return 14 in this case.
Complete the function **connectCost()** which takes the array and total number of sticks as input and returns the minimum cost of connecting all these sticks.

Input Format

First line of input contain T = number of test cases.
Each test case contain two lines, in first line, total number of sticks and next line will contain the lengths of each stick.

Output Format

For each test case, print the total cost of connecting all sticks in new lines.

Constraints

1 <= T <= 10
1 <= n <= 10^5
1 <= lengths[i] <= 1000

Sample Input

```
2
3
3 6 1
4
4 2 3 6
```

Sample Output

```
14
29
```

Solution:

```
import java.util.PriorityQueue;
class Result
{
    static int connectCost(int lengths[], int n)
    {
        // Min-heap
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        // Add all stick lengths
        for (int i = 0; i < n; i++) {
            pq.add(lengths[i]);
        }
        int totalCost = 0;
        // Keep connecting two smallest sticks
        while (pq.size() > 1)
        {
            int first = pq.poll();
            int second = pq.poll();
            int cost = first + second;
            totalCost += cost;
            pq.add(cost);
        }
        return totalCost;
    }
}
```

Aim: Implement Priority Queue using Linked List

Implement the priority queue using linked list representation.
Input Format:
First line of input contains an integer Q denoting the number of queries. A query can be of 1 of the below 2 types: -
(a) 1 item p (a query of this type means insert 'item' into the queue with priority p)
(b) 2 (a query of this type means to delete highest priority element from queue and print it, Assume lowest number has highest priority i.e. 0 is highest priority and 9 is lowest).
Next lines of each test case contains Q queries as shown below.
The functions void enqueue() & int dequeue() will take the head of list and modify list and head appropriately. The head pointer is passed by reference to both of them. You are required to complete the methods given.
Output Format:
The output for each test case will be space separated integers having -1 if the queue is empty else the element deleted from the queue.
Sample Input

```
1   // No. of test cases
4   // No. of queries
1   // Query type insert
10  // value
1   // priority
1   // Query type insert
20  // value
0   // priority
2   // Query type delete
2   // Query type delete
```

Sample Output

```
20 10
```

Explanation:
1st query is 1 10 1, which inserts element 10 with priority 1 in queue,
2nd query is 1 20 0, which inserts element 20 with priority 0 in queue,
3rd query is 2, which will delete the highest priority element and print it i.e. 20
4th query is 2, which will delete the highest priority element and print it i.e. 10

Solution:

```
class PQueueLL
{
    public QueueNode front, rear;
    // Insert element based on priority (ascending order)
    public void EnQueue(int data, int priority)
    {
        QueueNode newNode = new QueueNode(data, priority);
        // Case 1: Empty queue
        if (front == null)
        {
            front = rear = newNode;
            return;
        }
        // Case 2: New node has higher priority than front
        if (priority < front.priority)
```

```
            {
                newNode.next = front;
                front = newNode;
                return;
            }
            // Case 3: Find correct position
            QueueNode temp = front;
            while (temp.next != null && temp.next.priority <= priority)
            {
                temp = temp.next;
            }
            newNode.next = temp.next;
            temp.next = newNode;
            // Update rear if inserted at end
            if (newNode.next == null)
            {
                rear = newNode;
            }
        }
        // Remove highest priority element
        public int DeQueue()
        {
            if (front == null)
                return -1;
            int value = front.data;
            front = front.next;
            if (front == null)
                rear = null;
            return value;
        }
    }
```

Aim: Sort an array using heap sort

Given an array of N integers, sort them in ascending order using heap sort. Write the two functions **heapify()** and **heapSort(),** to implement the heap.

Input Format

First line contains the number of elements

Second line contains the elements of array separated by space.

Output Format

Print the elements of sorted array in ascending order.

Constraints

1 <= N <= 10^5

-(10^9) <= arr[i] <= 10^9

Sample Input

```
7
1 3 5 7 2 4 9
```

Sample Output

```
1 2 3 4 5 7 9
```

Solution:

```
class Result {
    static void heapify(int array[], int n, int i) {
        int largest = i;          // root
        int left = 2 * i + 1;     // left child
        int right = 2 * i + 2;    // right child
        // If left child is larger than root
        if (left < n && array[left] > array[largest]) {
            largest = left;
        }
        // If right child is larger than largest so far
        if (right < n && array[right] > array[largest]) {
            largest = right;
        }
        // If largest is not root
        if (largest != i) {
            int temp = array[i];
            array[i] = array[largest];
            array[largest] = temp;
            // Recursively heapify the affected subtree
            heapify(array, n, largest);
        }
    }
    static void heapSort(int array[], int n) {
        // Step 1: Build Max Heap
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(array, n, i);
        }
        // Step 2: Extract elements one by one
        for (int i = n - 1; i > 0; i--) {
            // Swap root with last element
            int temp = array[0];
            array[0] = array[i];
            array[i] = temp;
            // Heapify reduced heap
            heapify(array, i, 0);
        }
    }
}
```

Aim: Create a binary tree from array

Given an array of integer elements, create a binary tree from this array in level order fashion.
Input Format
The array of elements is given and you have to build the tree from it in level order i.e. elements from left in the array will be filled in the tree level wise starting from level 0.
Do not write the whole program, just complete the function buildTree(int arr[], int n) which takes the array and total number of nodes as parameter and return the root of the binary tree.

Note: Do not read any input from stdin/console. Just complete the functions provided. You can write more functions if required, but just above the given function.
þÿOutput Format
Print the tree in inorder.
Constraints
0 <= n <= 10^5
Sample Input
```
6  // n
1
2
3
4
5
6
```
Sample Tree
```
        1
       / \
      2    3
     / \  /
    4  5 6
```
Sample Output
```
4 2 5 1 6 3
```

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
// Return the root node of the tree
// Return the root node of the tree
static Node buildTree(int arr[], int n) {
    if (n == 0) return null;
    Node[] nodes = new Node[n];
    // Create all nodes
    for (int i = 0; i < n; i++) {
        nodes[i] = new Node(arr[i]);
    }
    // Link children
    for (int i = 0; i < n; i++) {
        int leftIndex = 2 * i + 1;
        int rightIndex = 2 * i + 2;
        if (leftIndex < n)
            nodes[i].left = nodes[leftIndex];
        if (rightIndex < n)
            nodes[i].right = nodes[rightIndex];
    }
    return nodes[0]; // root
}
```

Aim: Complete the inorder(), preorder() and postorder() functions for traversal with recursion

Given a binary tree, print it in inorder, preorder and postorder fashion with recursion.
Input
The root node of binary tree is given to you, and you have to complete the function
void inorder(Node root), void preorder(Node root) & void postorder(Node root)
to print the nodes of tree. (The function should use recursion).
Note: Do not read any input from stdin/console. Just complete the function provided. You can write more functions if required, but just above the given function.
Output
Print the nodes of tree separated by space by all three traversals in new lines.
Sample Input

```
6
1 2 3 4 5 6
```

Above input corresponds to below tree.

```
        1
       / \
      2    3
     / \  /
    4  5 6
```

Sample Output

```
4 2 5 1 6 3
1 2 4 5 3 6
4 5 2 6 3 1
```

Solution:

```
/* class Node {
  int data; // data used as key value
  Node leftChild;
  Node rightChild;
  public Node()  {
    data=0;  }
  public Node(int d)  {
    data=d;  }
 } Above class is declared for use. */
static void inorder(Node root)
{
    if (root == null)
        return;
    inorder(root.leftChild);
    System.out.print(root.data + " ");
    inorder(root.rightChild);
}
static void preorder(Node root)
{
    if (root == null)
        return;
    System.out.print(root.data + " ");
    preorder(root.leftChild);
    preorder(root.rightChild);
}
static void postorder(Node root)
{
    if (root == null)
        return;
    postorder(root.leftChild);
    postorder(root.rightChild);
    System.out.print(root.data + " ");
}
```

Aim: Find the right node of a given node

Given a binary tree and a key in it, find the node which is on same level and on right of this given key. If no such node present return -1.
Complete the function **findRightSibling()** which takes the address of the root node and an integer key as a parameter and return the right sibling (if exists, otherwise return -1).
þÿInput Format
First Line contains an integer N, denoting the number of integers to follow in the serialized representation of the tree.
Second line contains N space separated integers, denoting the level order description of left and right child of nodes, where -1 signifies a NULL child.
Third line contains an integer key whose right sibling is desired.
Output Format
Print the right sibling if any, otherwise print -1.
Sample Input 1
7
1 2 3 5 -1 -1 8
5
Sample Output 1
8
Explanation 1
```
        1
       / \
      2    3
     /      \
    5        8
```
þÿFor 5, the right sibling is 8.
Sample Input 2
6
1 2 3 5 6 7
7
Sample Output 2
-1
Explanation 2
```
        1
       / \
      2    3
     / \  /
    5  6 7
```
For 7, there is no node present in its right on the same level.
Therefore, the answer is -1.

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
import java.util.*;
class Result {
  static int findRightSibling(Node root, int key) {
    if (root == null)
      return -1;
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    while (!q.isEmpty()) {
      int size = q.size();
      for (int i = 0; i < size; i++) {
        Node curr = q.poll();
        // If key found
        if (curr.data == key) {
          // If not last node in this level
          if (i < size - 1)
            return q.peek().data;
          else
            return -1;
        }
        if (curr.left != null)
          q.add(curr.left);
        if (curr.right != null)
          q.add(curr.right);
      }
    }
    return -1;
  }
}
```

Aim: Print cousins of a given node in Binary Tree

Given a binary tree and a key value from this tree, print all the cousins of this node separated by space. If no cousin exists print -1.
Complete the function **printCousins()** which takes the address of the root node and a key k as a parameter and print the cousins of k separated by space or -1 if no cousin exists.
Input Format
First line contains the total number of nodes, second line contains the node labels separated by space. Third line contains an integer key k whose cousins are desired.
Output Format
For each test case, print the cousins of given key separated by space in new lines.
Sample Input
6
1 2 3 4 5 6
2
Sample Output
-1
Explanation:

```
        1
       / \
      2    3
     / \  /
    4  5 6
```

The parent of node 2 is 1, who have no siblings, no cousins for node 2.
If key is 6, then 2 is the sibling of parent i.e. 3. So the cousins are 4 and 5þÿ.

Solution:

```
/* class Node {
  int data; // data used as key value
  Node leftChild;
  Node rightChild;
  public Node()  {
    data=0;  }
  public Node(int d)  {
    data=d;  }
 } Above class is declared for use. */
import java.util.*;
class Result {
  static void printCousins(Node root, int k) {
    if (root == null || root.data == k) {
      System.out.print("-1");
      return;
    }
    Queue<Node> q = new LinkedList<>();
    q.add(root);
    boolean found = false;
    while (!q.isEmpty() && !found) {
      int size = q.size();
      for (int i = 0; i < size; i++) {
        Node curr = q.poll();
        if ((curr.leftChild != null && curr.leftChild.data == k) ||
            (curr.rightChild != null && curr.rightChild.data == k)) {
          found = true;
        } else {
          if (curr.leftChild != null)
            q.add(curr.leftChild);
          if (curr.rightChild != null)
            q.add(curr.rightChild);
        }
      }
    }
    if (!found || q.isEmpty()) {
      System.out.print("-1");
      return;
    }
    while (!q.isEmpty()) {
      System.out.print(q.poll().data + " ");
    }
  }
}
```

Aim: Find out if the tree can be folded or not

Given a binary tree, find out if it can be folded or not. A tree can be folded if left and right sub-trees of root are structure wise mirror images.
Note: A tree with zero or one node is foldable inherently.
Complete the function **isFoldable()** which takes the address of the root as parameter and return 1 if tree can be folded and 0 otherwise.
Input Format
First line contains an integer T, denoting the number of test cases.
For each test case:
First Line contains an integer N, denoting the number of integers to follow in the serialized representation of the tree.
Second line contains N space separated integers, denoting the level order description of left and right child of nodes, where -1 signifies a NULL child.
Output Format
For each test case, print 1 if tree is foldable and 0 otherwise, in new lines.
Sample Input 1

```
   1
  / \
 2   3
```

Sample Output 1
1
Sample Input 2

```
     1
    / \
   2    3
  / \
 4  5
```

Sample Output 2
0

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
class Result {
  static int isFoldable(Node root) {
    if (root == null)
      return 1;
    return isMirror(root.left, root.right) ? 1 : 0;
  }
  static boolean isMirror(Node n1, Node n2) {
    if (n1 == null && n2 == null)
      return true;
    if (n1 == null || n2 == null)
      return false;
    return isMirror(n1.left, n2.right) &&
           isMirror(n1.right, n2.left);
  }
}
```

Aim: Given two trees are identical or not

Given two binary trees, find out both are same or not. Two trees are considered same if they have same nodes and same structure.
So complete the function **areSameTree()** which takes the address of the root nodes of two trees as parameters and return **1** if both are same and **0** otherwise.
Input Format
The first line contains the number of testcases, T.
For each Testcase:
First line contains the number of nodes in first tree and second line contains the node labels in level-wise order.
Third line contains the number of nodes in second tree and fourth line contains the node labels in level-wise order.
Output Format
For each test case, print 1 if both tree are same and 0 otherwise.
Sample Input 1
1 // Number of testcases
3
1 2 3
3
1 2 3
Sample Output 1
1
Explanation 1

```
      1              1
     / \            / \
    2   3          2   3
```

As both the trees have same number of nodes and all same labels so trees are same.
Sample Input 2
1 // Number of testcases
5
1 2 3 4 5
3
1 2 3
Sample Output 2
0
Explanation 2

```
      1               1
     / \             / \
    2   3           2   3
   / \
  4   5
```

In this case, trees are not same

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
class Result {
    static int areSameTree(Node t1, Node t2) {
        // Both nodes are null -> identical
        if (t1 == null && t2 == null)
            return 1;
        // One node is null -> not identical
        if (t1 == null || t2 == null)
            return 0;
        // Node data mismatch -> not identical
        if (t1.data != t2.data)
            return 0;
        // Recursively check left and right subtrees
        int leftCheck = areSameTree(t1.left, t2.left);
        int rightCheck = areSameTree(t1.right, t2.right);
        return (leftCheck == 1 && rightCheck == 1) ? 1 : 0;
    }
}
```

Aim: Find maximum depth or height of a binary tree

Height of a tree is defined as the length of the longest downward path from root node to any leaf. If tree is empty, height is considered as -1 and for tree with only one node height is considered as 0.
Your task is that given a binary tree, find out the maximum depth of tree (also called height of tree).
Complete the function **treeHeight()** which takes the address of the root node of tree as parameter and return the height of the tree.
Input Format
First Line contains an integer N, denoting the number of integers to follow in the serialized representation of the tree.
Second line contains N space separated integers, denoting the level order description of left and right child of nodes, where -1 signifies a NULL child.
Output Format
For each test case, print the height of tree.
Constraints
0 <= N <= 10^5
Sample Input
5
1 2 3 4 5
Sample Output
2
Explanation:

```
        1
       / \
      2   3
     / \
    4  5
```

The longest path are 1-2-4 and 1-2-5, both have a length of 2, so tree height is 2.

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
class Result {
    static int treeHeight(Node root) {
        // Base case: empty tree
        if (root == null)
```

```
            return -1;
        // Recursive case
        int leftHeight = treeHeight(root.left);
        int rightHeight = treeHeight(root.right);
        return 1 + Math.max(leftHeight, rightHeight);
    }
}
```

Aim: Given binary tree is binary search tree or not

A binary tree is called binary search tree if it holds following three properties: -
a. The left subtree of a node contains nodes whose keys are less than that node’s key.
b. The right subtree of a node contains nodes whose keys are greater than that node’s key.
c. Both the left and right subtrees must also be binary search trees.
Your task is that, given a binary tree check if it is binary search tree or not. Complete the function **isBinarySearchTree()** which takes the address of the root node of tree as parameter and return **1** if the tree is binary search tree and **0** otherwise.
Input Format
First line contains an integer T, denoting the number of test cases.
For each test case:
  First Line contains an integer N, denoting the number of integers to follow in the serialized representation of the tree.
  Second line contains N space separated integers, denoting the level order description of left and right child of nodes, where -1 signifies a NULL child.
Output Format
For each test case, print 0 or 1, in new lines.
Constraints
1 <= T <= 10
0 <= N <= 10^5þÿ
Sample Input
1 // testcases
7 // N
4 2 7 1 3 5 8
Sample Output
1
Explanation:
Given tree is:

```
        4
      /   \
     2     7
   / \    / \
  1   3  5   8
```

Which satisfies the property of binary search tree, so return value is 1.

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
class Result {
    static int isBinarySearchTree(Node root) {
        return isBST(root, Long.MIN_VALUE, Long.MAX_VALUE) ? 1 : 0;
    }
    static boolean isBST(Node node, long min, long max) {
        if (node == null) return true;
        if (node.data <= min || node.data >= max) return false;
        return isBST(node.left, min, node.data) &&
               isBST(node.right, node.data, max);
    }
}
```

Aim: Find a lowest common ancestor of a given two nodes in a binary search tree

The Lowest Common Ancestor (LCA) of two nodes in a Binary Search Tree (BST) is defined as the deepest node from the root which lies in the path of both the nodes from the root.
So, given a binary search tree, find the Lowest Common Ancestor of two given nodes in it.
Complete the function **lowestCommonAncestor()** which takes the address of the root node of tree and two integer keys as parameters and return the lowest common ancestor of these two nodes (For empty tree return -1).
You can assume that both the given keys are present in the tree and are different from each other.
Input Format
First line contains an integer N, denoting the number of integers to follow in the serialized representation of the tree.
Second line contains N space separated integers, denoting the level order description of left and right child of nodes, where -1 signifies a NULL child.
Third line contains two integers k1 and k2 separated by space.
Output Format
Print the lowest common ancestor of given two nodes from the binary search tree.
Constraints
0 <= N <= 10^5
Sample Input

```
7
4 2 7 1 3 5 8
1 5
```

Sample Output

```
4
```

Explanation:

```
       4
      /  \
     2    7
    / \  / \
   1  3 5   8
```

Solution:

```
/*
 *  class Node {
 *    int data;
 *    Node left;
 *    Node right;
 *    public Node() {
 *      data = 0;
 *    }
 *    public Node(int d)  {
 *      data = d;
 *    }
 *  }
 *
 *  The above class defines a tree node.
 */
class Result {
    static int lowestCommonAncestor(Node root, int k1, int k2) {
        // Empty tree
        if (root == null)
            return -1;
        // Both keys are smaller than root -> go left
        if (k1 < root.data && k2 < root.data)
            return lowestCommonAncestor(root.left, k1, k2);
        // Both keys are greater than root -> go right
        if (k1 > root.data && k2 > root.data)
            return lowestCommonAncestor(root.right, k1, k2);
        // Keys are on different sides or one equals root -> root is LCA
        return root.data;
    }
}
```

Aim: Fractional Knapsack problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
In this problem we will solve a variant of it, in which the items may have a fractional decision i.e. you can pick the full item or partial depending on the capacity you have. Partial items are allowed to fill the bag at fullest. This is also known as fractional knapsack problem.
Given two integer arrays values[0..n-1] and weight[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of values[] such that sum of the weights of this subset is equal to W.
Input Format:
First Line will contain an integer N denoting the number of items.
Second line contains N integers separated by space as values of N items.
Third line contains N integers separated by space as weights of N items.
Fourth line contains an integer denoting the capcacity of knapsack.
Output Format:
Print the maximum value that can be earned with above knapsack.
Constraints:
1 <= N <= 10^5
1 <= val[i] <= 10^3
1 <= weight[i] <= 10^3
1 <= capacity <= 10^7
Sample Input

```
3
25 24 15
18 15 10
20
```

Sample Output

```
31.50
```

Explanation:
Pick 2nd item in full and 3rd item half which makes total weight as (15 + 10/2) = 20 and profit as (24 + 15/2) = 31.5

Solution:

```
import java.util.*;
class Result
{
    static double fractionalKnapsack(int val[], int weight[], int n, int capacity)
    {
        // Create array to store item index
        Integer[] idx = new Integer[n];
        for (int i = 0; i < n; i++) {
            idx[i] = i;
        }
        // Sort items by value/weight ratio (descending)
        Arrays.sort(idx, (i, j) ->
            Double.compare((double)val[j] / weight[j], (double)val[i] / weight[i])
        );
        double maxValue = 0.0;
        int remainingCapacity = capacity;
        // Pick items greedily
        for (int i = 0; i < n && remainingCapacity > 0; i++) {
            int item = idx[i];
            if (weight[item] <= remainingCapacity) {
                // Take full item
                maxValue += val[item];
                remainingCapacity -= weight[item];
            } else {
                // Take fractional part
                maxValue += ((double) val[item] / weight[item]) * remainingCapacity;
                remainingCapacity = 0;
            }
        }
        return maxValue;
    }
}
```

Aim: Interval scheduling Problem

Interval is defined as a span of time with start and end time. You are given **N** intervals with their start and finish times.
Two intervals can overlap each other if they have any time in common, e.g. Interval(5, 15) and Interval(10,20) are overlapping as time 10 to 15 is shared between them,
whereas Interval(5, 15) and Interval(20, 30) are non-overlapping as they do not have any common time.
Your task is to find the maximum number of intervals that can be scheduled, obviously they need to be non-overlapping.
Input Format:
The 1st line contains an integer N, the number of intervals.
The 2nd line contains N integers separated by space denoting starting times of N intervals.
The 3rd line contains N integers separated by space denoting finishing times of N intervals.
Output Format:
Print the number of non-overlapping intervals that can be scheduled.
Constraints:
1 <= N <= 10^5
Sample Input
```
4
10 25 12 40
26 32 22 50
```
Sample Output
```
3
```
Explanation:
We can start with the interval no 3 i.e. having start time 12 and finish time 22, then
with the interval no 2 i.e. having start time 25 and finish time 32 and last with the interval no 4 i.e. having start time 40 and finish time 50.

Solution:

```
import java.util.*;
class Result {
  static int intervalScheduling(int[] start, int[] end) {
    int n = start.length;
    // Create list of intervals
    int[][] intervals = new int[n][2];
    for (int i = 0; i < n; i++) {
      intervals[i][0] = start[i];
      intervals[i][1] = end[i];
    }
    // Sort by end time
    Arrays.sort(intervals, (a, b) -> a[1] - b[1]);
    int count = 1;
    int lastEnd = intervals[0][1];
    // Select non-overlapping intervals
    for (int i = 1; i < n; i++) {
      if (intervals[i][0] >= lastEnd) {
        count++;
        lastEnd = intervals[i][1];
      }
    }
    return count;
  }
}
```

Aim: Job Scheduling with deadlines

Mr. Amit have some jobs to be scheduled. Each job must meet the deadline to be counted as completed i.e. scheduling a job after its deadline is of no use.
Each job has some profit associated with it. Mr. Amit wants to schedule the as much jobs as he can and also want to earn the maximum profit.
Note: Each job needs 1 time unit for execution.
Input Format:
The 1st line contains an integer N, the number of jobs.
The 2nd line contains N integers separated by space denoting deadline times of N jobs.
The 3rd line contains N integers separated by space denoting profits of N jobs.
Output Format:
Print the maximum profit that can be earned.
Constraints:
1 <= N <= 10^5
1 <= Deadline <= 100
1 <= Profit <= 500
Sample Input
```
4
2 1 1 2
6 8 5 10
```
Sample Output
```
18
```
Explanation:
Job2 can be scheduled to execute at time 1 with profit 8.
Job4 can be scheduled to execþÿute at time 2 with profit 10.
So maximum profit that can be earned is 18.

Solution:

```
import java.util.*;
class Result {
  static int jobScheduling(int[] deadlines, int[] profits) {
    int n = deadlines.length;
    // Create jobs array: [deadline, profit]
    int[][] jobs = new int[n][2];
    int maxDeadline = 0;
    for (int i = 0; i < n; i++) {
      jobs[i][0] = deadlines[i];
      jobs[i][1] = profits[i];
      maxDeadline = Math.max(maxDeadline, deadlines[i]);
    }
    // Sort jobs by profit in descending order
    Arrays.sort(jobs, (a, b) -> b[1] - a[1]);
    boolean[] slot = new boolean[maxDeadline + 1];
    int totalProfit = 0;
    // Schedule jobs
    for (int i = 0; i < n; i++) {
      int deadline = jobs[i][0];
      // Find a free slot from deadline backwards
      for (int t = deadline; t > 0; t--) {
        if (!slot[t]) {
          slot[t] = true;
          totalProfit += jobs[i][1];
          break;
        }
      }
    }
    return totalProfit;
  }
}
```

Aim: Activity Selection Problem

Activity is defined as a interval of time for its execution. Every activity has a start time and a finish time. You are given **N** activities with their start and finish times.
Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.
Note: Remember that you should take care of overlapping of activity times, and always choose the very next activity instead of jump if that does not make any difference to the maximum number of activities.
Input Format:
The 1st line contains an integer N, the number of activities.
The 2nd line contains N integers separated by space denoting starting times of N activities.
The 3rd line contains N integers separated by space denoting finishing times of N activities.
Output Format:
Print the starting time of picked activities in sorted order separated by space.
Constraints:
1 <= N <= 10^5
Sample Input

```
3         // N
20 22 30  // start[]
30 35 40  // finish[]
```

Sample Output

```
20 30
```

Explanation:
First activity will finish at time 30, thereafter we can not select the 2nd activity.
However we can pick the 3rd activity as it will start after 1st will over.
So in this way we can select a maximum of 2 activities and their starting time are printed.

Solution:

```
import java.util.*;
class Result {
  // Print the starting time of selected activities in sorted order
  static void activitySelection(int[] start, int[] finish) {
    int n = start.length;
    // Store activities as [start, finish]
    int[][] activities = new int[n][2];
    for (int i = 0; i < n; i++) {
      activities[i][0] = start[i];
      activities[i][1] = finish[i];
    }
    // Sort by finish time
    Arrays.sort(activities, (a, b) -> a[1] - b[1]);
    List<Integer> selected = new ArrayList<>();
    // Pick first activity
    selected.add(activities[0][0]);
    int lastFinish = activities[0][1];
    // Pick remaining activities
    for (int i = 1; i < n; i++) {
      if (activities[i][0] >= lastFinish) {
        selected.add(activities[i][0]);
```

```
          lastFinish = activities[i][1];
        }
      }
      // Sort starting times as required in output
      Collections.sort(selected);
      // Print result
      for (int i = 0; i < selected.size(); i++) {
        System.out.print(selected.get(i));
        if (i < selected.size() - 1) System.out.print(" ");
      }
    }
  }
```

Aim: Count number of ways to cover a distance

Mr. Amit has a puzzle to solve. He needs your help. The puzzle is to find total number of ways to cover a distance of D steps where he can take on only 1,2,....,K steps in one go. For example, if D=3 & K=3, he can do it in 4 ways: {1+1+1, 1+2, 2+1, 3}, total 4 ways to complete 3 steps.
Input Format
First Line will contain an integer D denoting the distance to be covered.
Second Line will contain an integer K denoting the steps can be taken at most in one go.
Output Format
Print the total number of ways.
Sample Input
3
3
Sample Output
4

Solution:

```
class Result
{
    static int totalWaysToDistance(int d, int k){
        int[] dp = new int[d + 1];
        dp[0] = 1;
        for (int i = 1; i <= d; i++) {
            for (int step = 1; step <= k; step++) {
                if (i - step >= 0) {
                    dp[i] += dp[i - step];
                }
            }
        }
        return dp[d];
    }
}
```

Aim: Minimum Cost Path to last element of matrix

Given a cost matrix cost[M][N], write a function that returns cost of minimum cost path to reach (M, N) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (M, N) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1) and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

Input Format

First Line will contain two integers M and N denoting the size of matrix.
Second line contains M*N integers as matrix elements row-wise separated by space.

Output Format

Print the minimum cost to reach from source to destination in matrix.

Sample Input

```
3 3
1 2 3 4 8 2 1 5 3
```

Sample Output

```
8
```

Explanation

We can traverse from elements (0,0) -> (0,1) -> (1,2) -> (2,2)

Solution:

```
class Result
{
    static int minCostPath(int cost[][], int m, int n){
        int[][] dp = new int[m][n];
        dp[0][0] = cost[0][0];
        // Initialize first row
        for (int j = 1; j < n; j++) {
            dp[0][j] = dp[0][j-1] + cost[0][j];
        }
        // Initialize first column
        for (int i = 1; i < m; i++) {
            dp[i][0] = dp[i-1][0] + cost[i][0];
        }
        // Fill the rest
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                dp[i][j] = cost[i][j] + Math.min(
                    dp[i-1][j-1],
                    Math.min(dp[i-1][j], dp[i][j-1])
                );
            }
        }
        return dp[m-1][n-1];
    }
}
```

Aim: Longest Common Subsequence (LCS)

Overlapping Subproblems and Optimal Substructure properties are the prerequisites for solving a problem with dynamic programming. In this problem you need to figure them out. LCS is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.
A subsequence is defined as an ordered subset of the array's elements having the same sequential ordering as the original array.
Given two sequences, find the length of longest subsequence present in both of them. For example,
For sequences “**ABAZDC**” and “**BACBAD**” the LCS is “**ABAD**” of length 4.
For sequences “**AGGTAB**” and “**GXTXAYB**” the LCS is “**GTAB**” of length 4.
Note: A string of length n has 2^n-1 different possible subsequences, which implies that the time complexity of the brute force approach will be O(n * 2^n). It takes O(n) time to check if a subsequence is common to both the strings. This time complexity can be improved using dynamic programming.
Input Format
The first line of input contains an integer T denoting the no of test cases. Then T test cases follow.
Each test case contains two strings in two lines.
Output Format
For each test case, print the length of the Longest Common Subsequence in new lines.
Sample Input

```
2
ABAZDC
BACBAD
AGGTAB
GXTXAYB
```

Sample Output

```
4
4
```

Solution:

```
class Result
{
    static int longestCommonSubsequence(String str1, String str2){
        int n = str1.length();
        int m = str2.length();
        int[][] dp = new int[n + 1][m + 1];
        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= m; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (str1.charAt(i - 1) == str2.charAt(j - 1)) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[n][m];
    }
}
```

Aim: The Subset Sum problem

The subset sum problem is an important problem in computer science.
The challenge is to determine if there is some subset of numbers in an given array that can sum up to some given number S.
For example, if the array is {1, 4, 5, 10, 16} and sum = 9, you should return 1 and if sum = 7, you should return 0.

Input Format

First Line will contain an integer M denoting the sum.
Second Line will contain an integer N denoting the total number of elements.
Third line contains N integers separated by space.

Output Format

Print 1 if some subset found and 0 otherwise.

Sample Input

```
9
5
1 4 5 10 16
```

Sample Output

```
1
```

Solution:

```
class Result
{
    static int subsetSum(int a[], int n, int sum){
        boolean[][] dp = new boolean[n + 1][sum + 1];
        // sum 0 is always possible
        for (int i = 0; i <= n; i++)
            dp[i][0] = true;
        for (int i = 1; i <= n; i++) {
            for (int s = 1; s <= sum; s++) {
                if (a[i - 1] <= s) {
                    dp[i][s] = dp[i - 1][s] || dp[i - 1][s - a[i - 1]];
                } else {
                    dp[i][s] = dp[i - 1][s];
                }
            }
        }
        return dp[n][sum] ? 1 : 0;
    }
}
```

Aim: Matrix Chain Multiplication problem

A matrix can be represented as a 2-dimensional array. Two matrices are eligible for multiplication if the number of columns in 1st matrix is equal to the number of rows in 2nd matrix.
The total number of operations required for multiplying two matrices A [m x n] and B [n x o] will be (m * n * o).
We have many ways to multiply a chain of matrices because matrix multiplication is associative, each way require different number of operations to complete. Mr. Amit has a sequence of matrices and interested to find the way in which minimum operations are required to multiply all matrices. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.
Example :
If we had four matrices A, B, C, and D, we would have:
(ABC)D or (AB)(CD) or A(BCD) and we can use the same kind of parenthesizes for inner problems to get ((A(BC))D) or (((AB)C)D) or (AB)(CD) or (A((BC)D)) or (A(B(CD))).
Suppose A is a 10 × 20 matrix, B is a 20 × 30 matrix, and C is a 30 × 40 matrix. Then,
For 3 matrices we have 2 options
(A(BC)) = To Multiply BC we need 20*30*40=24000 and we get 20x40 matrix, then it is multiplied with A and take 10*20*40=8000 operations, so total 32000 operations needed.
((AB)C) = To Multiply AB we need 10*20*30=6000 and we get 10x30 matrix, then it is multiplied with C and take 10*30*40=12000 operations, so total 18000 operations needed.
Input Format:
First Line will contain an integer N denoting the number of matrices.
Second line contains N+1 integers denoting the matrix sizes separated by space.
Output Format:
Print the minimum operations needed for matrix multiplication as specified above.
Constraints
2 <= N <= 100
1 <= arr[i] <= 500
Sample Input
3
10 20 30 40
Sample Output
18000

Solution:

```
class Result
{
    // Matrix A[i] has dimension p[i-1] x p[i] for i = 1..n
    static int matrixChainMultiplication(int p[], int n){
        int[][] dp = new int[n + 1][n + 1];
        // length is chain length
        for (int len = 2; len <= n; len++) {
            for (int i = 1; i <= n - len + 1; i++) {
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k < j; k++) {
                    int cost = dp[i][k] + dp[k + 1][j]
                               + p[i - 1] * p[k] * p[j];
                    dp[i][j] = Math.min(dp[i][j], cost);
                }
            }
        }
        return dp[1][n];
    }
}
```

Aim: 0-1 Knapsack problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
In this problem we will solve a variant of it, in which the items can have a binary decision only i.e. either you can pick the item or leave it. No partial items allowed to fill the bag at fullest. This is also known as 0-1 knapsack problem.
Given two integer arrays values[0..n-1] and weight[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of values[] such that sum of the weights of this subset is smaller than or equal to W.
Note: You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).
Input Format
First Line will contain an integer N denoting the number of items.
Second line contains N integers separated by space as values of N items.
Third line contains N integers separated by space as weights of N items.
Fourth line contains an integer denoting the capcacity of knapsack.
Output Format
Print the maximum value that can be earned with above knapsack.
Constraints
1 <= N <= 10^3
1 <= val[i] <= 10^3
1 <= weight[i] <= 10^3
1 <= capacity <= 10^3
Sample Input

```
3
60 100 120
10 20 30
50
```

Sample Output

```
220
```

Explanation
Pick 2nd and 3rd item with weight 20 and 30 and profit 100 and 120.

Solution:

```
class Result
{
    static int zeroOneKnapsack(int val[], int weight[], int n, int capacity){
        int[][] dp = new int[n + 1][capacity + 1];
        for (int i = 1; i <= n; i++) {
            for (int w = 1; w <= capacity; w++) {
                if (weight[i - 1] <= w) {
                    dp[i][w] = Math.max(
                        dp[i - 1][w],
                        val[i - 1] + dp[i - 1][w - weight[i - 1]]
                    );
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
        return dp[n][capacity];
    }
}
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: Tom And Permutations

Tom loves to generate new strings by using various different methods. So, this time he decided to generate new strings by just rearranging the characters of some given string. In other words, he decided to find all the permutations for a given string and print them.
You task is to help Tom write a function to find all the permutations of a given string, that contains distinct characters.

Input Format:

First lines contains a string

Output Format:

Print all permutations of given string in lexicographical order

Sample Input

```
ABC
```

Sample Output

```
ABC
ACB
BAC
BCA
CAB
CBA
```

Solution:

```
import java.util.ArrayList;
import java.util.Arrays;
class Solve {
    ArrayList<String> permute(String str) {
        ArrayList<String> result = new ArrayList<>();
        char[] chars = str.toCharArray();
        Arrays.sort(chars); // for lexicographical order
        boolean[] used = new boolean[chars.length];
        backtrack(chars, used, new StringBuilder(), result);
        return result;
    }
    private void backtrack(char[] chars, boolean[] used,
                           StringBuilder current,
                           ArrayList<String> result) {
        if (current.length() == chars.length) {
            result.add(current.toString());
            return;
        }
        for (int i = 0; i < chars.length; i++) {
            if (used[i]) continue;
            used[i] = true;
            current.append(chars[i]);
            backtrack(chars, used, current, result);
            // backtrack
            current.deleteCharAt(current.length() - 1);
            used[i] = false;
        }
    }
}
```

Aim: Print all strings of n-bits

Given n-bits we can generate 2^n different strings from it. For example, with 3 bits we can generate 2^3=8 strings i.e. 000, 001, 010, 011, 100, 101, 110, 111.
You need to write a function to find all strings for a given number of bits.
Input Format:
First lines contains an integer denoting the number of bits
Output Format:
Print all strings for given number of bits in ascending order.
Sample Input

```
3
```

Sample Output

```
000
001
010
011
100
101
110
111
```

Solution:

```
import java.util.*;
class Solve
{
void generateAllStrings(int n, int i, char currStr[], ArrayList<String> strs){
    if(i==n){
        strs.add(new String(currStr));
        return;
    }
    currStr[i] = '0';
    generateAllStrings(n,i+1,currStr,strs);
    currStr[i] = '1';
    generateAllStrings(n,i+1,currStr,strs);
}
}
```

Aim: Solve N Queen problem

The N Queen problem is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.
You must know that a queen in chess can attack in all 8 directions.
Given the size of board as N*N, you need to count how many ways are there to place N queens on the board and save all such solutions in the given array
Input Format:
First lines contains an integer denoting the number of queens
Output Format:
Print all solutions in there column number for all queens.
If no solution exists, print -1.
Sample Input
4
Sample Output
```
1 3 0 2
2 0 3 1
```
Explanation:
Solutions are for per row starting from row 0 to row N-1.
1 3 0 2 means For 0th row pick column1, for 1st row pick column 3, for 2nd row pick column 0 and for 3rd row pick column 2.

There are total two solutions:
First is 1 3 0 2, means 1st queen at column 1, 2nd queen at column 3, 3rd queen at column 0 and 4th queen at column 2
Second is 2 0 3 1, means 1st queen at column 2, 2nd queen at column 0, 3rd queen at column 3 and 4th queen at column 1

Solution:

```
import java.util.ArrayList;
class Result {
    // Check if placing queen at board[row][col] is safe
    int isSafe(int board[][], int row, int col, int N) {
        // Check same row on left side
        for (int j = 0; j < col; j++) {
            if (board[row][j] == 1)
                return 0;
        }
        // Check upper left diagonal
        for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 1)
                return 0;
        }
        // Check lower left diagonal
        for (int i = row, j = col; i < N && j >= 0; i++, j--) {
            if (board[i][j] == 1)
                return 0;
        }
        return 1; // Safe
    }
    // Solve N Queen and store all solutions
    boolean solveNQUtil(int board[][], int col, int N,
                    ArrayList<ArrayList<Integer>> sol) {
        // If all queens are placed
        if (col == N) {
            ArrayList<Integer> temp = new ArrayList<>();
            for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++) {
                    if (board[i][j] == 1) {
                        temp.add(j); // store column index
                    }
                }
            }
            sol.add(temp);
            return true;
        }
        boolean res = false;
        // Try placing queen in all rows of this column
        for (int i = 0; i < N; i++) {
            if (isSafe(board, i, col, N) == 1) {
                board[i][col] = 1;
                res = solveNQUtil(board, col + 1, N, sol) || res;
                // Backtrack
                board[i][col] = 0;
            }
        }
        return res;
    }
}
```

Aim: Solve Sudoku

Sudoku is one of the most popular puzzle games of all time. The goal of Sudoku is to fill a 9×9 grid with numbers so that each row, column and 3×3 section contain all of the digits between 1 and 9. As a logic puzzle, Sudoku is also an excellent brain game. If you play Sudoku daily, you will soon start to see improvements in your concentration and overall brain power.
So, given a partially filled 9×9 matrix i.e. mat[9][9], Your goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and sub-grid of size 3×3 contains exactly one instance of the digits from 1 to 9.
If the given sudoku is not solvable, or having wrong prefilled entries, you should return -1, otherwise on successful solution return 1.
Input Format
9 lines contains 9 integers (0-9) separated by space as elements of the sudoku matrix.
Where 0 indicates null entry and other numbers are prefilled entries.
Output Format
Print the solved sudoku as shown.
Sample Input
```
2 9 0 8 7 3 0 1 0
4 0 0 0 0 5 9 2 0
0 1 0 0 2 4 0 0 0
0 0 0 0 8 9 6 0 0
0 0 4 0 0 0 8 3 0
0 8 2 3 1 0 5 0 0
0 0 9 2 3 8 0 0 7
8 0 0 0 4 7 0 0 0
3 0 5 0 9 0 2 8 4
```
Sample Output
```
2 9 6 8 7 3 4 1 5
4 3 7 1 6 5 9 2 8
5 1 8 9 2 4 7 6 3
1 5 3 4 8 9 6 7 2
9 6 4 7 5 2 8 3 1
7 8 2 3 1 6 5 4 9
6 4 9 2 3 8 1 5 7
8 2 1 5 4 7 3 9 6
3 7 5 6 9 1 2 8 4
```

Solution:

```
class Result {
    static final int N = 9;
    static int solveSudoku(int mat[][]) {
        if (solve(mat))
            return 1; // Sudoku solved
        else
            return -1; // No solution exists
    }
    // Recursive backtracking function
    private static boolean solve(int[][] mat) {
        for (int row = 0; row < N; row++) {
            for (int col = 0; col < N; col++) {
                if (mat[row][col] == 0) { // Empty cell
                    for (int num = 1; num <= 9; num++) {
                        if (isSafe(mat, row, col, num)) {
                            mat[row][col] = num;
                            if (solve(mat))
                                return true;
                            mat[row][col] = 0; // backtrack
                        }
                    }
                    return false; // no number can be placed here
                }
            }
        }
        return true; // all cells filled
    }
    // Check if placing 'num' at (row,col) is valid
    private static boolean isSafe(int[][] mat, int row, int col, int num) {
        // Check row
        for (int x = 0; x < N; x++)
            if (mat[row][x] == num)
                return false;
        // Check column
        for (int x = 0; x < N; x++)
            if (mat[x][col] == num)
                return false;
        // Check 3x3 subgrid
        int startRow = row - row % 3;
        int startCol = col - col % 3;
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                if (mat[startRow + i][startCol + j] == num)
                    return false;
        return true;
    }
}
```

Aim: Rat in a Maze Problem

A Maze is given as N*N binary matrix of blocks. A rat starts from **source** and has to reach the **destination**.
The rat can move only in two directions, i.e. **rightwards** and **downwards**.
In the maze matrix, -1 means the block is a dead end and 0 means the block can be used in the path from **source** to **destination**.
Note: Source block is the upper left most block i.e., maze[0][0] and **destination** block is lower rightmost block i.e., maze[N-1][N-1]
Your task is to find the number of possible solutions for the rat to reach the destination.
Input Format
First line contains number of rows i.e. N
Next N lines contains N binary numbers each denoting the maze
Constraints
2 <= N <= 15
Output Format
Print the number of solutions
Sample Input

```
4
 0 -1 -1 -1
 0  0 -1 -1
-1  0  0 -1
-1  0  0  0
```

Sample Output
2
Explanation:
Solution 1: From cell (0,0) -> down -> (1,0) -> forward -> (1,1) -> down -> (2,1) -> down -> (3,1) -> forward -> (3,2) -> forward -> (3,3)
Solution 2: From cell (0,0) -> down -> (1,0) -> forward -> (1,1) -> down -> (2,1) -> forward -> (2,2) -> down -> (3,2) -> forward -> (3,3)
Total 2 solutions possible for this maze.

Solution:

```
class Result {
    public static int solveMaze(int maze[][], int size) {
        return countPaths(maze, 0, 0, size);
    }
    // Recursive function to count all paths from (row,col) to destination
    private static int countPaths(int[][] maze, int row, int col, int N) {
        // Out of bounds or dead-end cell
        if (row >= N || col >= N || maze[row][col] == -1)
            return 0;
        // Reached destination
        if (row == N - 1 && col == N - 1)
            return 1;
        // Move Down
        int downPaths = countPaths(maze, row + 1, col, N);
        // Move Right
        int rightPaths = countPaths(maze, row, col + 1, N);
        // Total paths from this cell
        return downPaths + rightPaths;
    }
}
```

Aim: Count word in the board

Problem
Given a **m** x **n** 2D board of characters and a word, find how many times the word is present in the board.
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring.
Note: The same letter cell may not be used more than once.
Input Format
First line contains two space separated integers m, n that denotes the number of rows and columns in the board respectively.
Next m lines contain n characters each.
Last line contains the word to be searched.
Constraints
1 <= m, n <= 5
1 <= lengthOf(word) <= 30
Output Format
Print the total count of the given word in the 2-D board.
Sample Input 1
```
4 4
COXG
ADTA
BERM
ACDE
CODER  // word
```
Sample Output 1
```
1
```

Sample Input 2
```
3 4
HDST
ANEO
BENY
NEST  // word
```
Sample Output 2
```
2
```

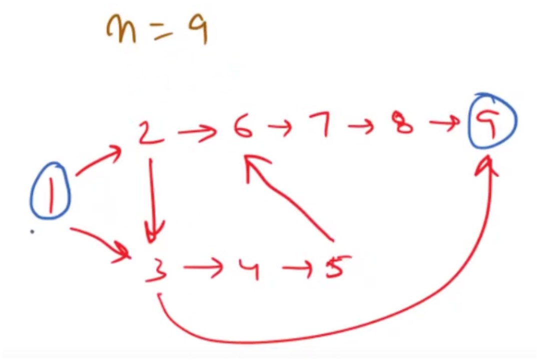
Solution:

```
// Do not change the class name
class Result {
    static int countWord(char board[][], String word, int m, int n) {
        boolean[][] visited = new boolean[m][n];
        int count = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (board[i][j] == word.charAt(0)) {
                    count += dfs(board, word, i, j, 0, visited, m, n);
                }
            }
        }
        return count;
    }
    private static int dfs(char[][] board, String word, int i, int j,
                           int index, boolean[][] visited, int m, int n) {
        // Boundary and validity checks
        if (i < 0 || j < 0 || i >= m || j >= n ||
            visited[i][j] || board[i][j] != word.charAt(index)) {
            return 0;
        }
        // If last character matched
        if (index == word.length() - 1) {
            return 1;
        }
        visited[i][j] = true;
        int count = 0;
        // Explore 4 directions
        count += dfs(board, word, i + 1, j, index + 1, visited, m, n);
        count += dfs(board, word, i - 1, j, index + 1, visited, m, n);
        count += dfs(board, word, i, j + 1, index + 1, visited, m, n);
        count += dfs(board, word, i, j - 1, index + 1, visited, m, n);
        visited[i][j] = false; // backtrack
        return count;
    }
}
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

Aim: Find the minimum number of edges in a path of a graph

Consider a directed graph whose vertices are numbered from 1 to n. There is an edge from a vertex i to a vertex j iff either j = i + 1 or j = 3i. The task is to find the minimum number of edges in a path in G from vertex 1 to vertex n.
Input Format:
The first line of input contains an integer T denoting the number of test cases. The description of T test cases follows.
Each test case contains a single line of input which contains an integer, n.
Output Format:
Print the number of edges in the shortest path from 1 to n.
Sample Input
1
9
Sample Output
2
Explanation:
The below given graph is formed



The minimum number of edges from vertex 1 to vertex 9 is 2.

Solution:

```
class Result {
    static int number_of_edges(int n) {
        int count = 0;
        while (n > 1) {
            if (n % 3 == 0) {
                n = n / 3;
            } else {
                n = n - 1;
            }
            count++;
        }
        return count;
    }
    }
```

Aim: Find path in a directed graph

Given a **directed graph** and two vertices(say source and destination vertex), check whether there exists some path from the given source vertex to the destination vertex or not. If it exists then print "YES", else print "NO".

Input Format:

First line contains two space separated integers V, E denoting the number of vertices and edges in the graph.

Following E lines contain two space separated integers u, v denoting a directed edge from u to v.

Last line contains two space separated integers src, dest denoting the source and destination vertex respectively.

Constraints:

1 <= V <= 100
0 <= E < 10000
0 <= u, v, src, dest < 100

Output Format:

Print "YES" if a path exists from the given source vertex to the destination vertex, else print "NO".

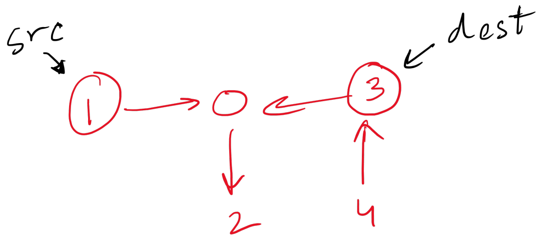
Sample Input 1

5 4 // V E
1 0 // directed edge from u to v
0 2
3 0
4 3
1 3 // source destination

Sample Output 1

NO

Explanation 1

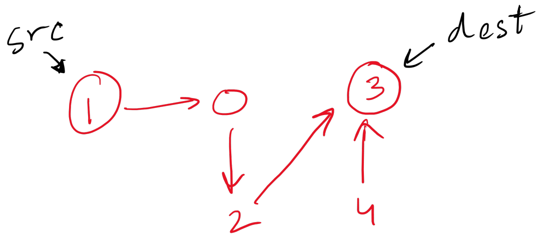


No path exists, from the given source vertex to the destination vertex

Sample Input 2

5 4 // V E
1 0
0 2
2 3

4 3
1 3
Sample Output 2
YES
Explanation 2



There is a path, from the given source vertex to the destination vertex

Solution:

```
import java.io.*;
import java.util.*;
// Do NOT change the class name
class Main {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        StringTokenizer st;
        st = new StringTokenizer(br.readLine());
        int V = Integer.parseInt(st.nextToken());
        int E = Integer.parseInt(st.nextToken());
        // As per constraints: 0 <= vertex < 100
        boolean[][] graph = new boolean[100][100];
        for (int i = 0; i < E; i++) {
            st = new StringTokenizer(br.readLine());
            int u = Integer.parseInt(st.nextToken());
            int v = Integer.parseInt(st.nextToken());
            graph[u][v] = true;
        }
        st = new StringTokenizer(br.readLine());
        int src = Integer.parseInt(st.nextToken());
        int dest = Integer.parseInt(st.nextToken());
        boolean[] visited = new boolean[100];
        Queue<Integer> queue = new LinkedList<>();
```

PRANSHU SHARMA
pranshu2259.be23@chitkara.edu.in

```
        queue.add(src);
        visited[src] = true;
        boolean found = false;
        while (!queue.isEmpty()) {
            int node = queue.poll();
            if (node == dest) {
                found = true;
                break;
            }
            for (int i = 0; i < 100; i++) {
                if (graph[node][i] && !visited[i]) {
                    visited[i] = true;
                    queue.add(i);
                }
            }
        }
        System.out.println(found ? "YES" : "NO");
    }
}
```

Aim: Number of Islands

Given a **m** x **n** binary matrix in which 1 represents 'land' and 0 represents 'water', find the number of islands.
An island is surrounded by water(**0s**) and is formed by connecting adjacent lands(**1s**) horizontally or vertically. You may assume all four edges of the given matrix are all surrounded by water.
Input Format:
First line contains two space separated integers m, n that denotes the number of rows and columns in the matrix respectively.
Each of the next m lines contain n space-separated integers representing the matrix.
Constraints:
1 <= m, n <= 150
Output Format:
Print the number of islands.
Sample Input

```
4 4
1 0 1 0
1 1 0 0
0 0 0 1
1 1 1 1
```

Sample Output

```
3
```

Explanation:
First island consists of following cells : (0, 0), (1, 0), (1,1)
Second island consists of following cells : (0, 2)
Third island consists of following cells : (2, 3), (3, 0), (3, 1), (3, 2), (3, 3)

Solution:

```
class Result {
    static int countIslands(int mat[][], int m, int n) {
        int count = 0;
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                // If land found, it's a new island
                if (mat[i][j] == 1) {
                    count++;
                    dfs(mat, i, j, m, n);
                }
            }
        }
        return count;
    }
    static void dfs(int mat[][], int i, int j, int m, int n) {
        // Boundary & water check
        if (i < 0 || j < 0 || i >= m || j >= n || mat[i][j] == 0)
            return;
        // Mark visited
        mat[i][j] = 0;
        // Explore all 4 directions
        dfs(mat, i + 1, j, m, n); // down
        dfs(mat, i - 1, j, m, n); // up
        dfs(mat, i, j + 1, m, n); // right
        dfs(mat, i, j - 1, m, n); // left
    }
}
```

Aim: Shortest path in a binary maze

Given a **m** x **n** binary matrix, find the length of the shortest path in the matrix from a given source cell to a destination cell. The path can only be created out of a cell if its value is 1. If there is no path from a given source cell to a destination cell, return -1.
Note: At any point of time, you can either move horizontally or vertically in the four directions.
Input Format:
First line contains two space separated integers m, n that denotes the number of rows and columns in the matrix respectively.
Each of the next m lines contain n space-separated integers representing the matrix.
Second Last line contains two space separated integers, denoting the position of the source cell.
Last line contains two space separated integers, denoting the position of the destination cell.
Constraints:
1 <= m, n <= 150
Output Format:
Print the length of the shortest path from source to destination.
Sample Input

```
5 5
0 1 1 1 1
1 1 0 0 1
1 0 0 1 1
1 1 1 1 0
0 1 0 1 0
1 1
3 2
```

Sample Output

```
5
```

Explanation:
The shortest path from (1, 1) to (3, 2) is as follows:
(1, 1) -> (1, 0) -> (2, 0) -> (3, 0) -> (3, 1) -> (3, 2)

Solution:

```
import java.util.*;
class Result {
    static int shortestPath(int mat[][], int srcR, int srcC,
                            int destR, int destC, int m, int n) {
        // If source or destination is blocked
        if (mat[srcR][srcC] == 0 || mat[destR][destC] == 0)
            return -1;
        boolean[][] visited = new boolean[m][n];
        Queue<int[]> q = new LinkedList<>();
        // row, col, distance
        q.add(new int[]{srcR, srcC, 0});
        visited[srcR][srcC] = true;
        int[] dr = {-1, 1, 0, 0};
        int[] dc = {0, 0, -1, 1};
        while (!q.isEmpty()) {
            int[] curr = q.poll();
            int r = curr[0];
            int c = curr[1];
            int dist = curr[2];
            // Destination reached
            if (r == destR && c == destC)
                return dist;
            // Explore 4 directions
            for (int i = 0; i < 4; i++) {
                int nr = r + dr[i];
                int nc = c + dc[i];
                if (nr >= 0 && nr < m && nc >= 0 && nc < n &&
                    mat[nr][nc] == 1 && !visited[nr][nc]) {
                    visited[nr][nc] = true;
                    q.add(new int[]{nr, nc, dist + 1});
                }
            }
        }
        return -1;
    }
}
```

Aim: Depth First Traversal of Graph

Graphs can be traversed popularly in following two ways: Depth First Traversal (DFT) and Breadth First Traversal (BFT). Traversing Graphs can be little bit more tricky than traversing tree because of possibilities of a cycle in graph. So, you need to to write the function for doing a depth first traversal over the given graph.

Input Format:

First line contains an integer V i.e. number of nodes.
Second line contains an integer E i.e. number of edges.
Next E lines contains two integers each v1, v2 denoting the edge from v1 to v2.
Last line of input contains an integer denoting the starting vertex for DFT.

Output Format:

Print the vertex labels starting from given vertex separated by space.

Sample Input

```
4
5
0 1
0 2
1 2
2 0
2 3
2
```

Sample Output

```
2 0 1 3
```

Solution:

```
/* The class is defined with below variables
  class Graph
{
  private Map<Integer, List<Integer>> adjVertices;
  public Graph() {
    this.adjVertices = new HashMap<Integer, List<Integer>>();
  }
  public void addVertex(int vertex) {
    adjVertices.putIfAbsent(vertex, new ArrayList<>());
  }
  public void addEdge(int src, int dest) {
    adjVertices.get(src).add(dest);
  }       */
// Recursive helper function
void DFSUtil(int v, boolean visited[])
{
    visited[v] = true;
    System.out.print(v + " "); // print when first visited
    for (int adj : adjVertices.get(v)) {
        if (!visited[adj]) {
            DFSUtil(adj, visited);
        }
    }
}
// DFS function called externally
```

```
void DFS(int v)
{
   boolean visited[] = new boolean[adjVertices.size()];
   DFSUtil(v, visited);
}
```

Aim: Breadth First Traversal of Graph

Graphs can be traversed popularly in following two ways: Depth First Traversal (DFT) and Breadth First Traversal (BFT). Traversing Graphs can be little bit more tricky than traversing tree because of possibilities of a cycle in graph. So, you need to to write the function for doing a breadth first traversal over the given graph.

Input Format:

First line contains an integer V i.e. number of nodes.
Second line contains an integer E i.e. number of edges.
Next E lines contains two integers each v1, v2 denoting the edge from v1 to v2.
Last line of input contains an integer denoting the starting vertex for BFT.

Output Format:

Print the vertex labels starting from given vertex separated by space.

Sample Input

```
4
5
0 1
0 2
1 2
2 0
2 3
2
```

Sample Output

```
2 0 3 1
```

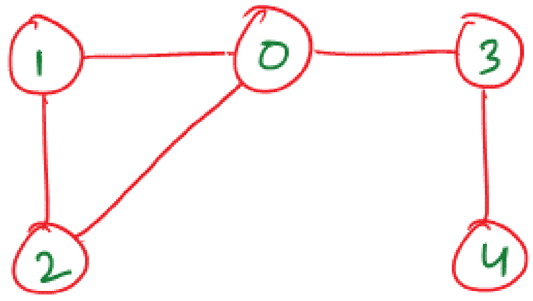
Solution:

```
/* The class is defined with below variables
class Graph {
  private int V;
  private Map<Integer, List<Integer>> adjVertices;
  public Graph(int V) {
      this.V = V;
    this.adjVertices = new HashMap<Integer, List<Integer>>();
  }
  public void addVertex(int vertex) {
    adjVertices.putIfAbsent(vertex, new ArrayList<>());
  }
  public void addEdge(int src, int dest) {
    adjVertices.get(src).add(dest);
  }        */
void BFS(int v)
{
    boolean visited[] = new boolean[V];
    Queue<Integer> q = new LinkedList<>();
    // Start from vertex v
    visited[v] = true;
    q.add(v);
    while (!q.isEmpty()) {
        int curr = q.poll();
        System.out.print(curr + " ");
        // Add all unvisited neighbors
```

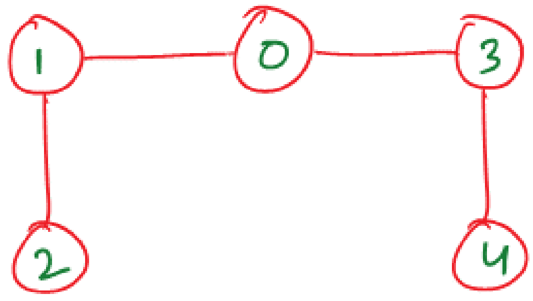
```
            for (int adj : adjVertices.get(curr)) {
                if (!visited[adj]) {
                    visited[adj] = true;
                    q.add(adj);
                }
            }
        }
    }
```

Aim: Find the cycle in undirected graph

Given a un-directed graph find out if there is a cycle in it or not.
Note: The given graph **may** or **may not** be connected.
Example 1: The following graph contains a cycle



Example 2: The following graph does **not** contain a cycle



Input Format
The first line contains the Number of vertices, V.
The second line contains the Number of edges, E.
Then E lines follow , containing 2 numbers,x & y, seperated by space denoting an edge from vertex x to y.
Output Format
Print Yes or No depending on graph has cycle or not.
Sample Input
5 // Vertices
5 // Edges
0 1 // Edge from v1 to v2
0 2
1 2

0 3
3 4

Sample Output

Yes

Explanation:

The graph is having a cycle from vertices 0, 1 and 2

Solution:

```
import java.util.*;
class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int V = sc.nextInt(); // Number of vertices
        int E = sc.nextInt(); // Number of edges
        // Build adjacency list
        List<List<Integer>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++)
            adj.add(new ArrayList<>());
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            adj.get(u).add(v);
            adj.get(v).add(u); // undirected graph
        }
        boolean[] visited = new boolean[V];
        boolean hasCycle = false;
        for (int i = 0; i < V; i++) {
            if (!visited[i] && dfs(i, -1, visited, adj)) {
                hasCycle = true;
                break;
            }
        }
        System.out.println(hasCycle ? "Yes" : "No");
    }
    // DFS helper function
    private static boolean dfs(int node, int parent, boolean[] visited, List<List<Integer>> adj) {
        visited[node] = true;
        for (int neighbor : adj.get(node)) {
            if (!visited[neighbor]) {
                if (dfs(neighbor, node, visited, adj))
                    return true;
            } else if (neighbor != parent) {
                // Visited neighbor that is not parent -> cycle found
                return true;
            }
        }
        return false;
    }
}
```