**Batch- T8**

**Practical No. 5**

**Name: Siya Pondkule**

**PRN: 23520008**

**Assignment No 5**

1. **You are working on the city construction project. You have A houses in the city. You have to divide these houses into B localities such that every locality has at least one house. Also, every house in a locality should have a telephone connection wire with each of the other houses in the locality. You are given integers A and B.**

**Task**

**Print the minimum and the maximum number of telephone connections possible if you design the city accordingly.**

**Algorithm:**

**Step 1: Calculate Maximum Number of Connections**

* 1. The idea is to maximize the size of one locality and minimize the size of the others.
  2. To do this, place as many houses as possible in a single locality. Specifically, one locality should have A−B+ houses, and the remaining B−1 localities will each have one house (since each locality must have at least one house).
  3. The total number of connections will then be:

maxConnections=(A−B+1)(A−B)/2

This formula represents the number of connections in the largest locality (with A−B+1A - B + 1A−B+1 houses). The other B−1B - 1B−1 localities each have one house, which does not add any connections because a single house cannot form connections with itself.

**Step 2: Calculate Minimum Number of Connections**

* 1. The goal is to distribute the houses as evenly as possible across the B localities.
  2. Let the number of houses in each locality be as equal as possible:
  3. Let q=⌊AB⌋q (the number of houses in each locality if they are divided evenly).
  4. Let r=A% B (the remainder, representing the extra houses that cannot be evenly distributed across all localities).

**Step 3: The distribution of houses:**

* 1. r localities will have q+1 houses.
  2. The remaining B−r localities will have q houses.

**Step 4: The total number of connections in this case will be:**

minConnections=r×(q+1)q2+(B−r)×q(q−1)

This formula calculates the number of connections for the localities with q+1 houses and the localities with q houses.

**Step 5: Return the Results:** The algorithm now has both the maximum and minimum number of telephone connections, so the final step is to output both values.

import java.util.\*;

public class tel\_conn{

public static void main(String args[]){

    Scanner sc=new Scanner(System.in);

    System.out.print("Enter the number of houses (A): ");

    int A = sc.nextInt();

    System.out.print("Enter the number of localities (B): ");

    int B = sc.nextInt();

    int housesPerLocality = A / B;

    int extraHouses = A % B;

    int minConnections = (B - extraHouses) \* (housesPerLocality \* (housesPerLocality - 1)) / 2

                       + extraHouses \* (housesPerLocality \* (housesPerLocality + 1)) / 2;

    int maxConnections = ((A - B + 1) \* (A - B)) / 2;

    System.out.println("Minimum number of telephone connections: " + minConnections);

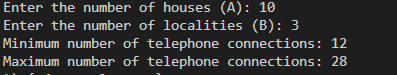
    System.out.println("Maximum number of telephone connections: " + maxConnections);

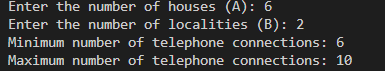
    sc.close();

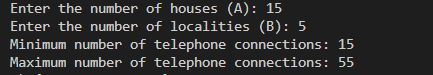
}

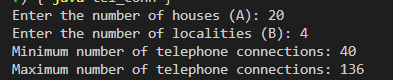
}

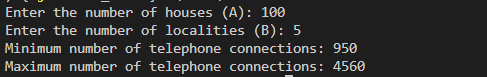
**Output:**











1. **You are working in the Data Consistency team of your company. You are allocated a task as follows:**

* **You have a data stream consisting of an equal number of odd and even numbers. You can make separations in the data stream but the number of odd elements should be equal to the number of even elements in both partitions after separation. Also, if you make a separation between a number *x* and number *y,* then the cost of this operation will be |x-y| coins.**

**You are given the following:**

* **An integer *N***
* **An array *arr***
* ***A*n integer *K***

**Task**

**Determine the maximum number of separations that can be made in the array by spending no more than *K* coins.**

**Step-by-Step Algorithm:**

* 1. An integer NNN (the number of elements in the array).
  2. An array arr of integers.
     1. An integer KKK (the maximum amount of coins available for separations).

1. **Initialize Variables:**
   1. Create two lists:
      1. odds: To store all odd numbers from the array.
      2. evens: To store all even numbers from the array.
   2. Create a variable separations to count the number of valid separations made.
2. **Classify Elements:**
   1. Traverse the array arr and classify each element as odd or even. Append it to the corresponding list (odds or evens).
3. **Check for Valid Partitions:**
   1. The number of separations can only occur when both partitions (before and after the separation) have equal counts of odd and even numbers.
   2. Since you have an equal number of odd and even numbers, the maximum number of separations possible is min⁡(length of odds,length of evens)−1\
4. **Calculate Costs:**
   1. For each possible separation (between consecutive elements), calculate the cost of separation as the absolute difference: cost=∣arr[i]−arr[i+1]∣ Maintain a list of costs for each valid separation.
5. **Sort Costs:**
   1. Sort the list of costs in ascending order to prioritize the cheapest separations first.
6. **Perform Separations Within Budget:**
   1. Initialize a variable total\_cost to keep track of the cumulative cost.
   2. Iterate through the sorted list of costs and for each cost:
      1. Check if adding this cost to total\_cost exceeds K.
      2. If it does not exceed, increment the separations count and add the cost to total\_cost.
      3. If it exceeds, break the loop as further separations will only incur equal or greater costs.
7. **Output the Result:**
   1. The value in separations will be the maximum number of separations that can be made without exceeding K.

import java.util.\*;

public class data\_stream {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of elements (N): ");

        int N = scanner.nextInt();

        int[] arr = new int[N];

        System.out.print("Enter the array elements: ");

        for (int i = 0; i < N; i++) {

            arr[i] = scanner.nextInt();

        }

        System.out.print("Enter the maximum coins (K): ");

        int K = scanner.nextInt();

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

        int oddCount = 0;

        int evenCount = 0;

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

            if (arr[i] % 2 == 0) {

                evenCount++;

            } else {

                oddCount++;

            }

            if (oddCount == evenCount) {

                int cost = Math.abs(arr[i] - arr[i + 1]);

                separationCosts.add(cost);

            }

        }

        Collections.sort(separationCosts);

        int totalCost = 0;

        int maxSeparations = 0;

        for (int cost : separationCosts) {

            if (totalCost + cost <= K) {

                totalCost += cost;

                maxSeparations++;

            } else {

                break;

            }

        }

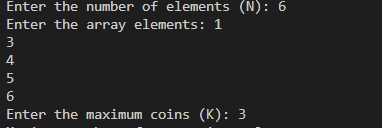
        System.out.println("Maximum number of separations: " + maxSeparations);

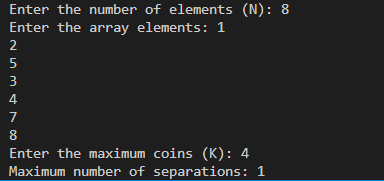
        scanner.close();

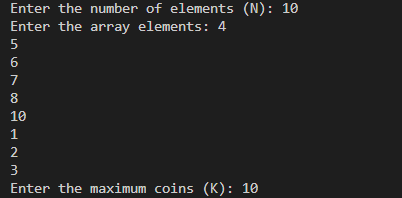
    }

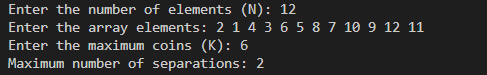
}

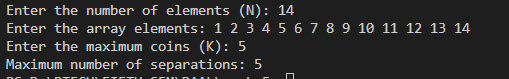
**Output:**

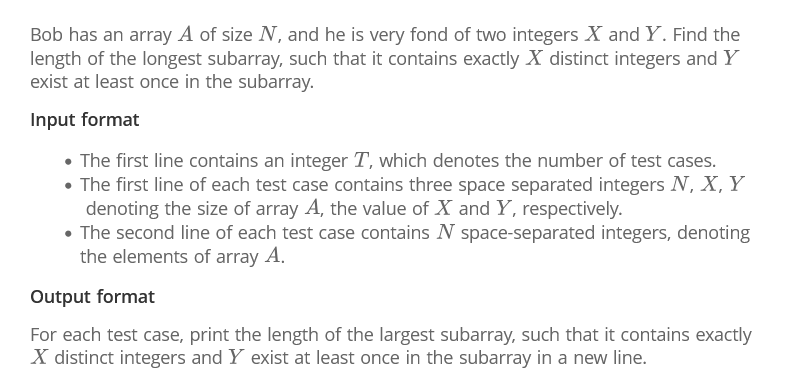










3)

### Step-by-Step Algorithm

1. **Initialize Variables:**
   1. Create a HashMap countMap to keep track of the count of each integer in the current window (subarray).
   2. Initialize left pointer to 0, which will represent the start of the current window.
   3. Initialize maxLength to store the maximum length of the valid subarrays found so far.
   4. Initialize distinctCount to count the number of distinct integers in the current window.
2. **Iterate Through the Array:**
   1. Use a loop with right pointer to iterate over the array A from the beginning to the end. The right pointer represents the end of the current window.
3. **Update the Count Map:**
   1. For each element A[right]:
      1. Check if it is a new distinct integer by looking it up in countMap. If its count is 0, increment distinctCount.
      2. Update the count of A[right] in countMap.
4. **Maintain the Distinct Count:**
   1. If distinctCount exceeds xxx (the allowed number of distinct integers), shrink the window from the left:
      1. Decrement the count of the integer at A[left] in countMap.
      2. If the count of A[left] becomes 0, decrement distinctCount and remove the integer from countMap.
      3. Increment the left pointer to narrow the window.
5. **Check for Valid Subarray:**
   1. After adjusting the window, check if distinctCount is exactly xxx and whether y exists in the countMap (i.e., it must appear at least once in the current window).
   2. If both conditions are met, update maxLength to the maximum of its current value and the size of the current window: current window size=right−left+1
6. **Return Result:**
   1. After iterating through the array, return maxLength, which contains the length of the longest subarray that meets the conditions.

import java.util.HashMap;

import java.util.Scanner;

public class LongestSubarray {

    public static int longestSubarray(int[] A, int x, int y) {

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

        int left = 0, maxLength = 0, distinctCount = 0;

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

            if (countMap.getOrDefault(A[right], 0) == 0) {

                distinctCount++;

            }

            countMap.put(A[right], countMap.getOrDefault(A[right], 0) + 1);

            while (distinctCount > x) {

                countMap.put(A[left], countMap.get(A[left]) - 1);

                if (countMap.get(A[left]) == 0) {

                    distinctCount--;

                    countMap.remove(A[left]);

                }

                left++;

            }

            if (distinctCount == x && countMap.getOrDefault(y, 0) > 0) {

                maxLength = Math.max(maxLength, right - left + 1);

            }

        }

        return maxLength;

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the size of the array: ");

        int n = scanner.nextInt();

        int[] A = new int[n];

        System.out.println("Enter the elements of the array:");

        for (int i = 0; i < n; i++) {

            A[i] = scanner.nextInt();

        }

        System.out.print("Enter the number of distinct integers (x): ");

        int x = scanner.nextInt();

        System.out.print("Enter the integer that must exist at least once (y): ");

        int y = scanner.nextInt();

        int result = longestSubarray(A, x, y);

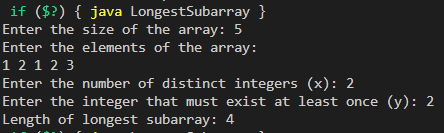
        System.out.println("Length of longest subarray: " + result);

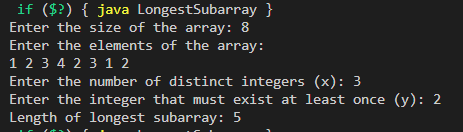
        scanner.close();

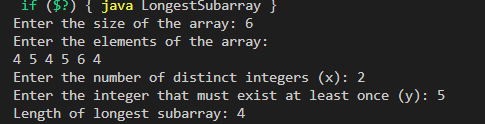
    }

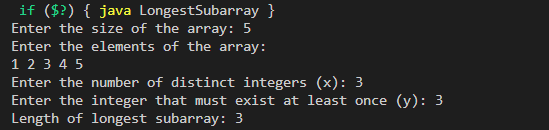
}

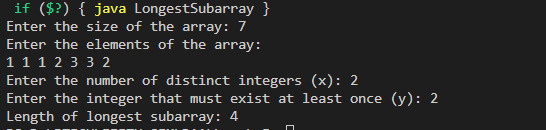
Output:











4**) The country of Byteland consists of n cities. Between any 2 cities it is possible to have a railway track and a road. Railway tracks are bidirectional, meaning if there exists a railway track between u and v then you can take a train from u to v as well as from v to u. Similarly, roads are bidirectional, meaning if there exists a route between u and v then you can drive from u to v as well as from v to u.**

**2 cities, u and v are called railway-connected if it is possible to travel between u and v using railway tracks.**

**2 cities, u and v are called road-connected if it is possible to travel between u and v using roads. The transportation network is called balanced if for all pairs of cities u, v:**

**u,v are railway-connected** [**if and only if**](https://en.wikipedia.org/wiki/If_and_only_if#Definition) **u,v are road-connected.**

**Initially, there are n cities and no roads or railways in Byteland. You will be given q instructions asking you to build either a railway track or a road between some 2 cities. After each instruction, you must report whether the transportation network is balanced.**

**Input format**

**The first line of input will contain 2 integers, n and q. q lines will follow. Each line will contain 3 space-separated integers in one of the following formats:**

**1 u v : build a railway track between u and v**

**2 u v : build a road between u and v**

**Output format**

**You must print q lines. The ith line contains an answer to the question whether the transport network is balanced after the ith instruction. If it is balanced print "YES" (without quotes) otherwise print "NO" (without quotes)**

**Pseudo code:**

FUNCTION BytelandTransport(n, q):

// Initialize Union-Find structures for railways and roads

railways = UnionFind(n)

roads = UnionFind(n)

// Process each instruction

FOR i FROM 1 TO q DO:

PRINT "Enter instruction (type, u, v):"

READ type, u, v

IF type == 1 THEN

railways.union(u, v)

PRINT "Added railway track between u and v"

ELSE IF type == 2 THEN

roads.union(u, v)

PRINT "Added road between u and v"

// Check if the transport network is balanced

balanced = TRUE

FOR j FROM 1 TO n DO:

IF railways.connected(j, 1) != roads.connected(j, 1) THEN

balanced = FALSE

BREAK

// Output the result

IF balanced THEN

PRINT "After instruction i: YES"

ELSE

PRINT "After instruction i: NO"

PRINT "All instructions processed."

END FUNCTION

// Union-Find data structure

FUNCTION UnionFind(n):

parent = ARRAY of size n+1

size = ARRAY of size n+1

FOR i FROM 1 TO n DO:

parent[i] = i

size[i] = 1

RETURN OBJECT with methods find(u), union(u, v), and connected(u, v)

END FUNCTION

// Method to find the root of a node

FUNCTION find(u):

IF parent[u] != u THEN

parent[u] = find(parent[u]) // Path compression

RETURN parent[u]

END FUNCTION

// Method to union two nodes

FUNCTION union(u, v):

rootU = find(u)

rootV = find(v)

IF rootU != rootV THEN

IF size[rootU] < size[rootV] THEN

parent[rootU] = rootV

size[rootV] += size[rootU]

ELSE

parent[rootV] = rootU

size[rootU] += size[rootV]

END FUNCTION

// Method to check if two nodes are connected

FUNCTION connected(u, v):

RETURN find(u) == find(v)

END FUNCTION

import java.util.Scanner;

public class BytelandTransport {

    static class UnionFind {

        private int[] parent;

        private int[] size;

        public UnionFind(int n) {

            parent = new int[n + 1];

            size = new int[n + 1];

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

                parent[i] = i;

                size[i] = 1;

            }

        }

        public int find(int u) {

            if (parent[u] != u) {

                parent[u] = find(parent[u]);

            }

            return parent[u];

        }

        public void union(int u, int v) {

            int rootU = find(u);

            int rootV = find(v);

            if (rootU != rootV) {

                if (size[rootU] < size[rootV]) {

                    parent[rootU] = rootV;

                    size[rootV] += size[rootU];

                } else {

                    parent[rootV] = rootU;

                    size[rootU] += size[rootV];

                }

            }

        }

        public boolean connected(int u, int v) {

            return find(u) == find(v);

        }

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the number of cities (n) and number of instructions (q): ");

        int n = scanner.nextInt();

        int q = scanner.nextInt();

        UnionFind railways = new UnionFind(n);

        UnionFind roads = new UnionFind(n);

        System.out.println("Processing instructions...");

        for (int i = 0; i < q; i++) {

            System.out.print("Enter instruction (type, u, v): ");

            int type = scanner.nextInt();

            int u = scanner.nextInt();

            int v = scanner.nextInt();

            if (type == 1) {

                railways.union(u, v);

                System.out.println("Added railway track between " + u + " and " + v);

            } else {

                roads.union(u, v);

                System.out.println("Added road between " + u + " and " + v);

            }

            boolean balanced = true;

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

                if (railways.connected(j, 1) != roads.connected(j, 1)) {

                    balanced = false;

                    break;

                }

            }

            System.out.println("After instruction " + (i + 1) + ": " + (balanced ? "YES" : "NO"));

        }

        scanner.close();

        System.out.println("All instructions processed.");

    }

}

**Output:**

