**Batch- T7**

**Practical No. 5**

**Title of Assignment**

**Student Name: Om Kulkarni**

**Student PRN: 22510034**

**Problem Statement:**

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.

1. Algorithm/Pseudocode

**Input:**

* A: Number of houses.
* B: Number of localities.

**Output:**

* The minimum and maximum number of telephone connections.

**Steps:**

1. **Step 1: Calculate the number of connections for a given number of houses.**
   * Define a function connections(n) that calculates the number of connections in a locality with n houses using the formula: connections(n)=n×(n−1)2
2. **Step 2: Calculate the maximum number of connections.**
   * The maximum number of connections occurs when one locality has as many houses as possible, and the remaining localities have only one house each.
   * Set maxConnections = connections(A - B + 1) for the large locality (with A - B + 1 houses).
   * Add connections for the remaining B-1 localities, each of which has 1 house: maxConnections=connections(A−B+1)+(B−1)×connections(1)
3. **Step 3: Calculate the minimum number of connections.**
   * To minimize the connections, distribute the houses as evenly as possible among the B localities.
   * Compute:
     + housesPerLocality = A // B (integer division).
     + remainder = A % B (remainder of the division, meaning some localities will have one extra house).
   * For the remainder localities, there will be housesPerLocality + 1 houses, and for the remaining B - remainder localities, there will be housesPerLocality houses: minConnections=remainder×connections(housesPerLocality+1)+(B−remainder)×connections(housesPerLocality
4. **Step 4: Output the results.**
   * Print the minConnections and maxConnections.

Program Code

#include <iostream>

using namespace std;

// Function to calculate the number of connections for `n` houses

int connections(int *n*) {

    return (*n* \* (*n* - 1)) / 2;

}

int main() {

    int A, B;

    cout << "Enter the number of houses (A) and the number of localities (B): ";

    cin >> A >> B;

    // Maximum number of connections: One large locality, rest with one house each

    int maxConnections = connections(A - B + 1);  // (A-B+1) houses in one large locality

    for (int i = 1; i < B; ++i) {

        maxConnections += connections(1);  // Each of the other (B-1) localities has 1 house

    }

    // Minimum number of connections: Distribute houses as evenly as possible

    int housesPerLocality = A / B;

    int remainder = A % B;

    int minConnections = 0;

    // Add connections for the localities with (housesPerLocality + 1) houses

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

        minConnections += connections(housesPerLocality + 1);

    }

    // Add connections for the localities with housesPerLocality houses

    for (int i = 0; i < (B - remainder); ++i) {

        minConnections += connections(housesPerLocality);

    }

    // Output the results

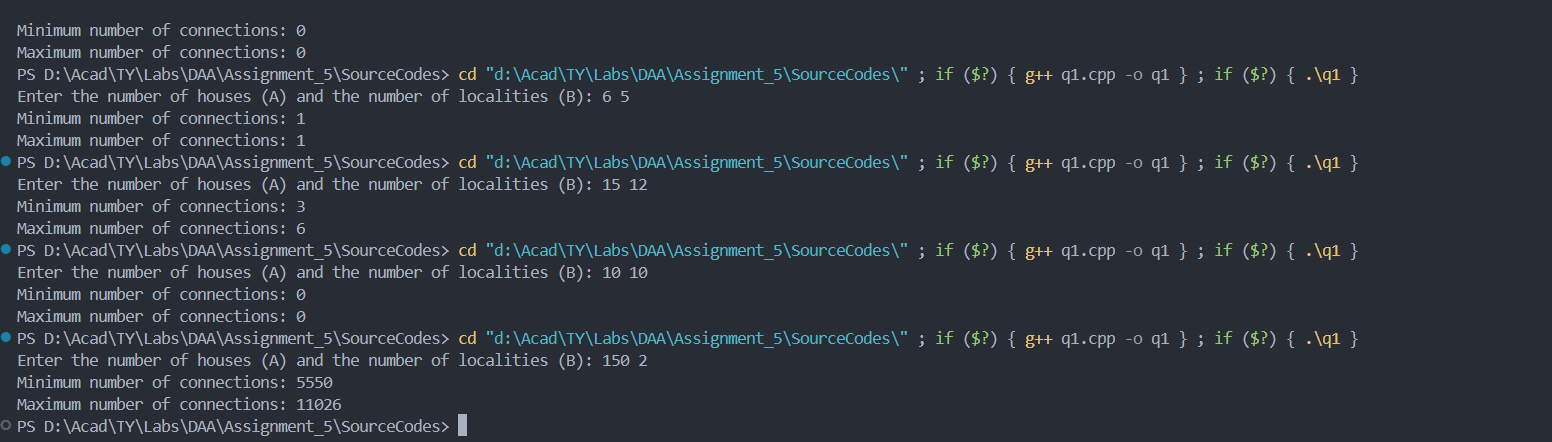
    cout << "Minimum number of connections: " << minConnections << endl;

    cout << "Maximum number of connections: " << maxConnections << endl;

    return 0;

}

1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

**Time Complexity:**

* All operations performed (arithmetic calculations and function calls) are constant-time operations.
* Therefore, the overall time complexity is O(1).

**Space Complexity:**

* The algorithm only uses a few variables to store intermediate values (housesPerLocality, remainder, minConnections, maxConnections), and does not use any additional data structures that grow with input size.
* Therefore, the space complexity is O(1)

**Problem Statement:**

2)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 • An integer K Task Determine the maximum number of separations that can be made in the array by spending no more than K coins.

1. Algorithm/Pseudocode

**Initialize oddCount = 0, evenCount = 0, totalCost = 0, and separations = 0.**

**Iterate through the array from i = 0 to N-2:**

* If arr[i] is odd, increment oddCount.
* If arr[i] is even, increment evenCount.

I**f oddCount == evenCount:**

* Calculate cost = |arr[i] - arr[i+1]|.
* If totalCost + cost <= K, increment separations and update totalCost.
* Otherwise, break out of the loop.

**Return the value of separations.**

#include <iostream>

#include <vector>

#include <cmath>  // For abs function

using namespace std;

int maxSeparations(int *N*, vector<int>& *arr*, int *K*) {

    int oddCount = 0, evenCount = 0;

    int totalCost = 0;

    int separations = 0;

    // Traverse the array and track odd/even count

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

        if (*arr*[i] % 2 == 0)

            evenCount++;

        else

            oddCount++;

        // If odd and even counts are equal, check if we can make a separation

        if (oddCount == evenCount) {

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

            if (totalCost + cost <= *K*) {

                totalCost += cost;

                separations++;

            } else {

                break;  // If adding the cost exceeds K, stop

            }

        }

    }

    return separations;

}

int main() {

    int N, K;

    cout << "Enter the size of the array: ";

    cin >> N;

    vector<int> arr(N);

    cout << "Enter the elements of the array: ";

    for (int i = 0; i < N; i++) {

        cin >> arr[i];

    }

    cout << "Enter the maximum cost K: ";

    cin >> K;

    int result = maxSeparations(N, arr, K);

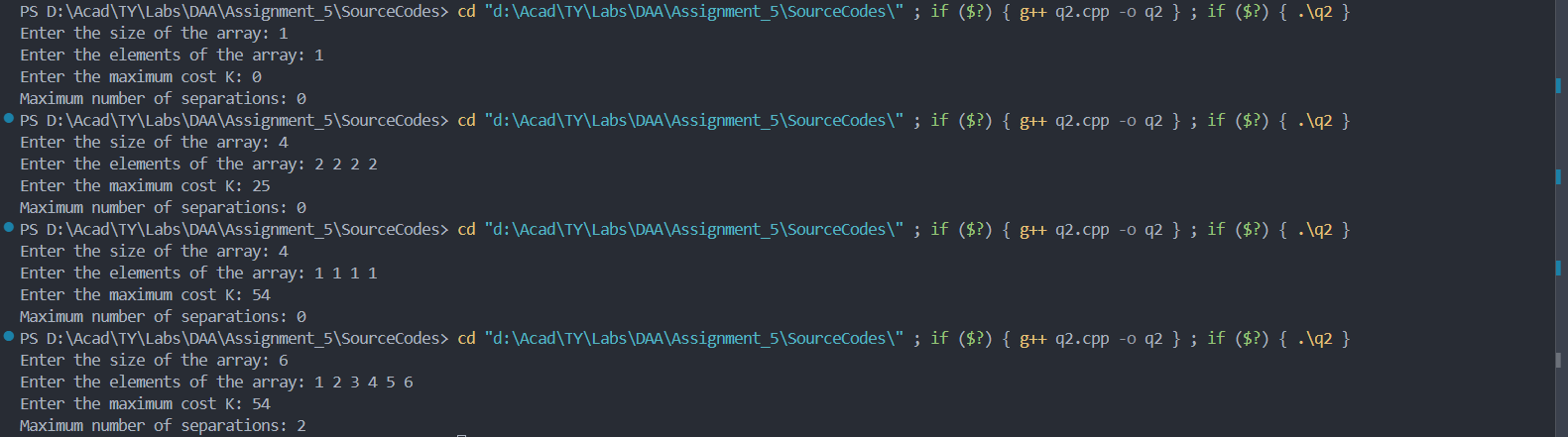
    cout << "Maximum number of separations: " << result << endl;

    return 0;

}

Program Code

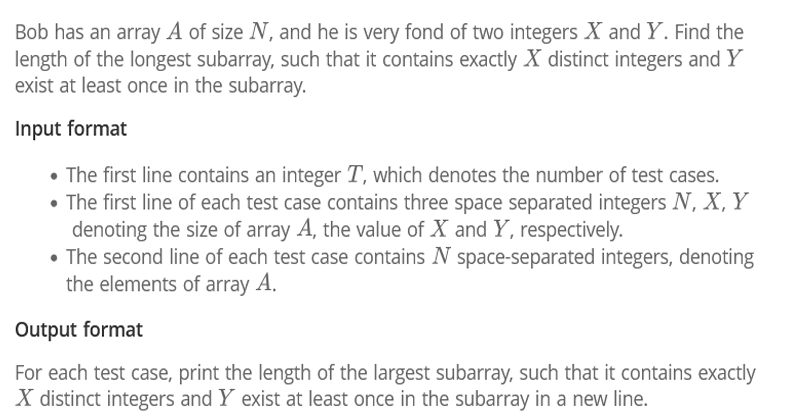
1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

* **Time Complexity:** O(N)
  + We traverse the array once, making calculations in constant time for each element.
* **Space Complexity:** O(1)
  + Only a few variables are used, regardless of the size of the input array.

**Problem Statement:**

****

1. Algorithm/Pseudocode

**Input Parsing**: Read the input values for multiple test cases.

**Initialize Variables**:

* A hash map (or dictionary) to keep track of element frequencies.
* Two pointers left and right for the sliding window.

**Sliding Window Logic**:

Start with both pointers at the beginning.

* Expand the window by moving the right pointer and update the frequencies.
* When the number of distinct elements exceeds X, shrink the window by moving the left pointer.
* Ensure that Y is present in the subarray and the number of distinct elements is exactly X.

**Store the Maximum Length** that satisfies the condition for each test case.

Program Code

#include <iostream>

#include <vector>

#include <unordered\_map>

using namespace std;

int longestSubarray(vector<int>& *arr*, int *N*, int *X*, int *Y*) {

    unordered\_map<int, int> freq;  // To store frequency of elements

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

    bool containsY = false;

    for (right = 0; right < *N*; ++right) {

        int current = *arr*[right];

        // Add the current element to the frequency map

        if (freq[current] == 0) {

            distinctCount++;

        }

        freq[current]++;

        // Check if Y exists in the subarray

        if (current == *Y*) {

            containsY = true;

        }

        // Shrink the window if distinct elements exceed X

        while (distinctCount > *X*) {

            freq[*arr*[left]]--;

            if (freq[*arr*[left]] == 0) {

                distinctCount--;

            }

            if (*arr*[left] == *Y* && freq[*arr*[left]] == 0) {

                containsY = false; // Y is no longer in the window

            }

            left++;

        }

        // Update the maxLength if the window contains exactly X distinct elements and Y is present

        if (distinctCount == *X* && containsY) {

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

        }

    }

    return maxLength;

}

int main() {

        int N, X, Y;

        cin >> N >> X >> Y;  // Size of array, value of X, and value of Y

        vector<int> arr(N);

        for (int i = 0; i < N; i++) {

            cin >> arr[i];  // Array elements

        }

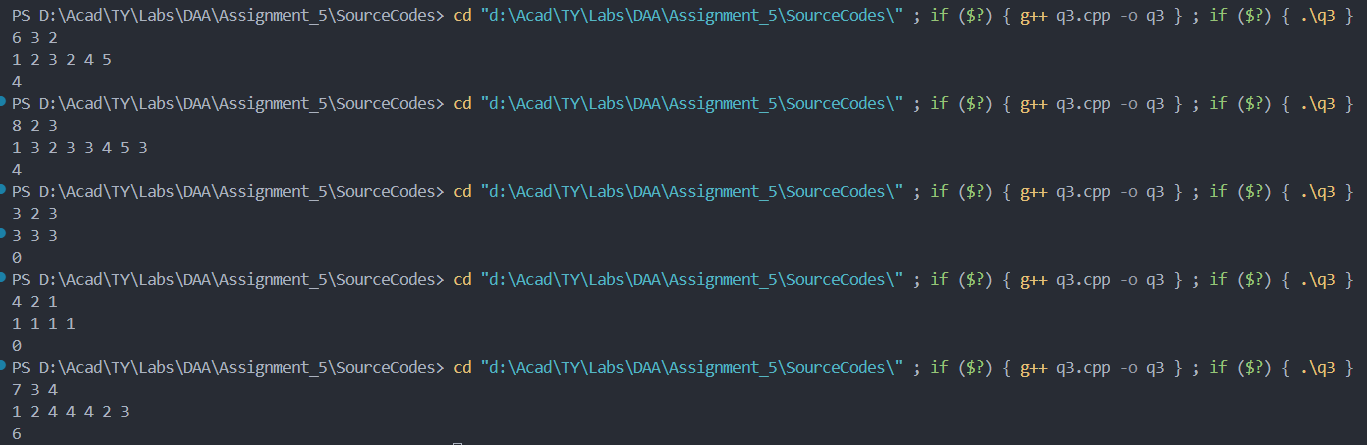
        int result = longestSubarray(arr, N, X, Y);

        cout << result << endl;  // Print the result for each test case

    return 0;

}

1. Output with verity of test cases



Analysis in terms of complexity wherever applicable.

 **Time Complexity**: O(N) per test case, since each element is processed at most twice (once when adding to the window and once when removing).

 **Space Complexity**: O(N) due to the space used by the frequency map.

**Problem Statement:**

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 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)

1. Algorithm/Pseudocode

**Initialization**:

* Create two DSU structures: one for tracking railway connections and one for tracking road connections.
* Read the input: number of cities n and number of queries q.

**Union-Find Operations**:

* For each query:
  + If the query type is 1 u v, union the cities u and v in the **railway DSU**.
  + If the query type is 2 u v, union the cities u and v in the **road DSU**.

**Check Balanced Condition**:

* After each union operation (query), iterate over all cities and check whether they have the same representative (root) in both DSUs.
  + If they do, the network is balanced; print "YES".
  + Otherwise, print "NO".

**Repeat the Process** for all queries and output the results.

Program Code

#include <iostream>

#include <vector>

using namespace std;

// Union-Find structure to handle connections

class UnionFind {

private:

    vector<int> parent, rank;

public:

    // Constructor to initialize DSU

    UnionFind(int *n*) {

        parent.resize(*n*);

        rank.resize(*n*, 1);

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

            parent[i] = i; // Every node is its own parent initially

        }

    }

    // Find function with path compression

    int find(int *u*) {

        if (parent[*u*] != *u*) {

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

        }

        return parent[*u*];

    }

    // Union function with union by rank

    void unionSets(int *u*, int *v*) {

        int root\_u = find(*u*);

        int root\_v = find(*v*);

        if (root\_u != root\_v) {

            if (rank[root\_u] > rank[root\_v]) {

                parent[root\_v] = root\_u;

            } else if (rank[root\_u] < rank[root\_v]) {

                parent[root\_u] = root\_v;

            } else {

                parent[root\_v] = root\_u;

                rank[root\_u]++;

            }

        }

    }

};

// Function to check if the network is balanced after each query

vector<string> isBalanced(int *n*, int *q*, vector<vector<int>>& *queries*) {

    UnionFind railwayUF(*n*); // DSU for railway tracks

    UnionFind roadUF(*n*);    // DSU for roads

    vector<string> result;

    for (const auto& query : *queries*) {

        int type = query[0], u = query[1] - 1, v = query[2] - 1; // Convert to 0-indexed

        if (type == 1) {

            railwayUF.unionSets(u, v); // Build railway track

        } else if (type == 2) {

            roadUF.unionSets(u, v);    // Build road

        }

        // Check if the network is balanced

        bool balanced = true;

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

            if (railwayUF.find(i) != roadUF.find(i)) {

                balanced = false;

                break;

            }

        }

        if (balanced) {

            result.push\_back("YES");

        } else {

            result.push\_back("NO");

        }

    }

    return result;

}

int main() {

    int n, q;

    cin >> n >> q;

    vector<vector<int>> queries(q, vector<int>(3));

    for (int i = 0; i < q; ++i) {

        cin >> queries[i][0] >> queries[i][1] >> queries[i][2];

    }

    vector<string> result = isBalanced(n, q, queries);

    for (const string& res : result) {

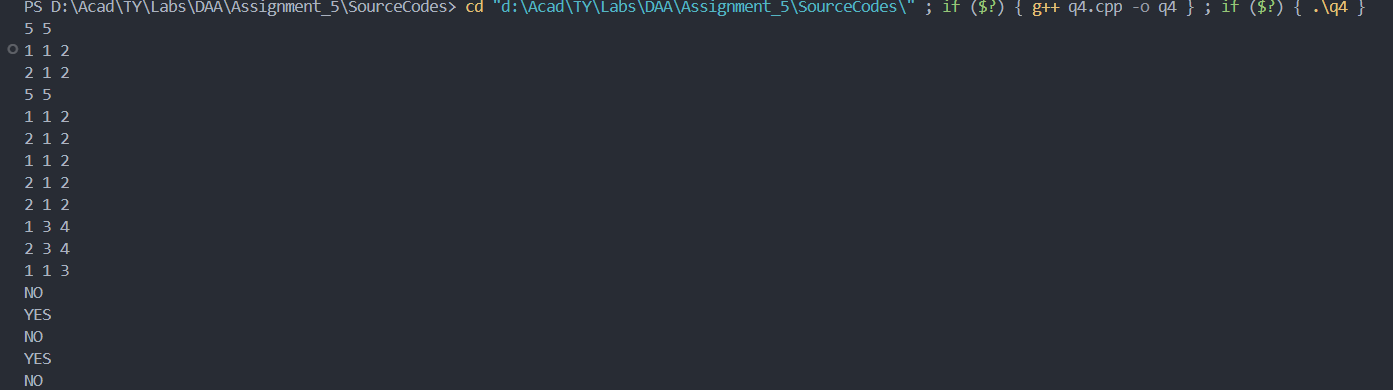
        cout << res << endl;

    }

    return 0;

}

1. Output with verity of test cases



1. Analysis in terms of complexity wherever applicable.

**Time Complexity:**

* **Union-Find Operations (Find and Union)**: Each operation takes **O(α(n))**, where **α(n)** is the inverse Ackermann function, which is nearly constant and can be considered **O(1)**.
* **Checking for Balanced Network**: For each query, we check if all cities are connected the same way in both DSUs, which takes **O(n)**.
* **Total Time Complexity**:
  + For q queries, the overall time complexity is **O(q \* n)**.

**Space Complexity:**

* We use two DSU structures (for railways and roads), each requiring **O(n)** space for storing the parent and rank arrays.
* **Total Space Complexity**: **O(n)**.