Batch:T6

Practical No.5

Title of Assignment: Greedy approach

Student Name: Parshwa Herwade

Student PRN: 22510064

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.

ANS.

Pseudocode:  
function minimum\_connections(A, B):

x = A // B # Minimum houses per locality

y = A % B # Extra houses to distribute

min\_connections = (B - y) \* (x \* (x - 1)) // 2 + y \* (x \* (x + 1)) // 2

return min\_connections

function maximum\_connections(A):

max\_connections = A \* (A - 1) // 2

return max\_connections

input A, B

print minimum\_connections(A, B)

print maximum\_connections(A)

Code:

#include <iostream>

using namespace std;

int minimum\_connections(int A, int B) {

    int x = A / B;

    int y = A % B;

    int min\_connections = (B - y) \* (x \* (x - 1)) / 2 + y \* (x \* (x + 1)) / 2;

    return min\_connections;

}

int maximum\_connections(int A) {

    return A \* (A - 1) / 2;

}

int main() {

    int A, B;

    cin >> A >> B;

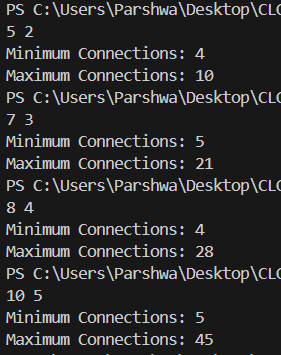
    cout << "Minimum Connections: " << minimum\_connections(A, B) << endl;

    cout << "Maximum Connections: " << maximum\_connections(A) << endl;

    return 0;

}

Output:



**Time Complexity:**

* Best/Average/Worst: O(1) since the operations are constant-time arithmetic.

**Space Complexity:**

* O(1) as we only store a few variables.

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.

ANS.

Pseudocode

function max\_separations(arr, N, K):

odd\_count = 0

even\_count = 0

separations = 0

cost = 0

for i = 0 to N-2:

if arr[i] is odd:

odd\_count += 1

else:

even\_count += 1

if odd\_count == even\_count:

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

if cost <= K:

separations += 1

else:

break

return separations

input N, K, arr[]

print max\_separations(arr, N, K)

Code:  
#include <iostream>

#include <cmath>

using namespace std;

int max\_separations(int arr[], int N, int K) {

    int odd\_count = 0, even\_count = 0, separations = 0, cost = 0;

    for (int i = 0; i < N - 1; i++) {

        if (arr[i] % 2 == 0)

            even\_count++;

        else

            odd\_count++;

        if (odd\_count == even\_count) {

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

            if (cost <= K) {

                separations++;

            } else {

                break;

            }

        }

    }

    return separations;

}

int main() {

    int N, K;

    cin >> N >> K;

    int arr[N];

    for (int i = 0; i < N; i++) {

        cin >> arr[i];

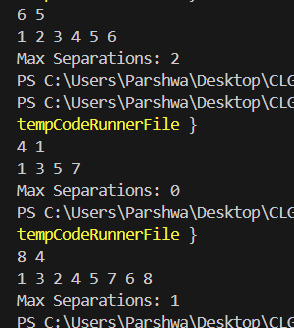
    }

    cout << "Max Separations: " << max\_separations(arr, N, K) << endl;

    return 0;

}

Output:

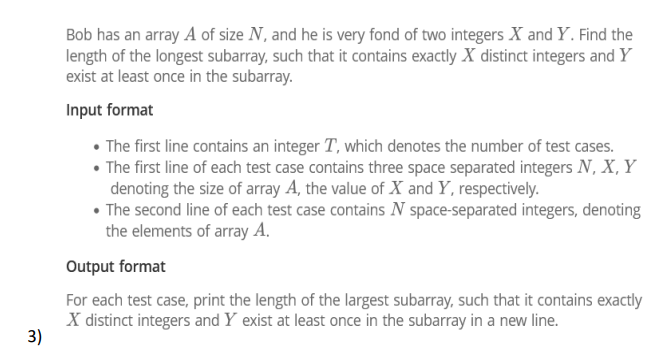


**Time Complexity:**

* Best/Average/Worst: O(N) since we traverse the array once.

**Space Complexity:**

* O(1) as we use a fixed number of variables.



ANS.

Pseudocode:

function longest\_subarray\_with\_conditions(A, N, X, Y):

freq\_map = empty dictionary

left = 0

max\_len = 0

distinct\_count = 0

y\_present = false

for right = 0 to N-1:

# Add A[right] to the window

if A[right] not in freq\_map or freq\_map[A[right]] == 0:

distinct\_count += 1

freq\_map[A[right]] += 1

# Check if Y is in the current window

if A[right] == Y:

y\_present = true

# Shrink window if distinct\_count exceeds X

while distinct\_count > X:

freq\_map[A[left]] -= 1

if freq\_map[A[left]] == 0:

distinct\_count -= 1

if A[left] == Y:

y\_present = false

left += 1

# If the window has exactly X distinct elements and contains Y, update max\_len

if distinct\_count == X and y\_present:

max\_len = max(max\_len, right - left + 1)

return max\_len

# Input handling

input T

for each test case:

input N, X, Y

input array A[N]

result = longest\_subarray\_with\_conditions(A, N, X, Y)

print result

Code:

#include <iostream>

#include <unordered\_map>

#include <vector>

using namespace std;

int longest\_subarray\_with\_conditions(const vector<int>& A, int N, int X, int Y) {

    unordered\_map<int, int> freq\_map;

    int left = 0, max\_len = 0, distinct\_count = 0;

    bool y\_present = false;

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

        if (freq\_map[A[right]] == 0)

            distinct\_count++;

        freq\_map[A[right]]++;

        if (A[right] == Y)

            y\_present = true;

        while (distinct\_count > X) {

            freq\_map[A[left]]--;

            if (freq\_map[A[left]] == 0)

                distinct\_count--;

            if (A[left] == Y)

                y\_present = false;

            left++;

        }

        if (distinct\_count == X && y\_present)

            max\_len = max(max\_len, right - left + 1);

    }

    return max\_len;

}

int main() {

    int T;

    cin >> T;

    while (T--) {

        int N, X, Y;

        cin >> N >> X >> Y;

        vector<int> A(N);

        for (int i = 0; i < N; ++i)

            cin >> A[i];

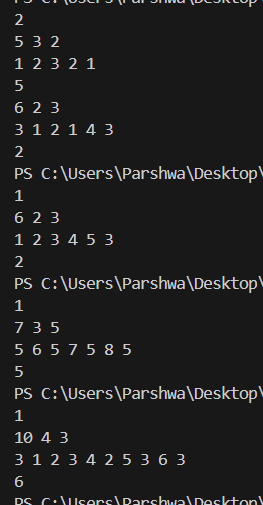
        cout << longest\_subarray\_with\_conditions(A, N, X, Y) << endl;

    }

    return 0;

}

Output:



**Time Complexity:**

* **Best/Average/Worst Case**: O(N) per test case due to the sliding window approach where both pointers (left and right) only traverse the array once.
  + Overall complexity for T test cases: O(T \* N).

**Space Complexity:**

* O(N) due to the frequency map that stores counts of up to N elements.

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)

ANS.

Pseudocode

function find(parent[], u):

if parent[u] != u:

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

return parent[u]

function union(parent[], rank[], u, v):

root\_u = find(parent, u)

root\_v = find(parent, 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] += 1

function is\_balanced(rail\_parent[], road\_parent[], u, v):

return find(rail\_parent, u) == find(road\_parent, u)

input n, q

initialize rail\_parent[], road\_parent[], rail\_rank[], road\_rank[]

for each query:

if type == 1:

union(rail\_parent, rail\_rank, u, v)

else:

union(road\_parent, road\_rank, u, v)

if is\_balanced(rail\_parent, road\_parent, u, v):

print "YES"

else:

print "NO"

Code:

#include <iostream>

#include <vector>

using namespace std;

int find(vector<int>& parent, int u) {

    if (parent[u] != u)

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

    return parent[u];

}

void union\_sets(vector<int>& parent, vector<int>& rank, int u, int v) {

    int root\_u = find(parent, u);

    int root\_v = find(parent, 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]++;

        }

    }

}

bool is\_balanced(vector<int>& rail\_parent, vector<int>& road\_parent, int u, int v) {

    return find(rail\_parent, u) == find(rail\_parent, v) &&

           find(road\_parent, u) == find(road\_parent, v);

}

int main() {

    int n, q;

    cin >> n >> q;

    vector<int> rail\_parent(n + 1), road\_parent(n + 1), rail\_rank(n + 1, 0), road\_rank(n + 1, 0);

    for (int i = 1; i <= n; i++) {

        rail\_parent[i] = i;

        road\_parent[i] = i;

    }

    for (int i = 0; i < q; i++) {

        int type, u, v;

        cin >> type >> u >> v;

        if (type == 1) {

            union\_sets(rail\_parent, rail\_rank, u, v);

        } else {

            union\_sets(road\_parent, road\_rank, u, v);

        }

        if (is\_balanced(rail\_parent, road\_parent, u, v)) {

            cout << "YES" << endl;

        } else {

            cout << "NO" << endl;

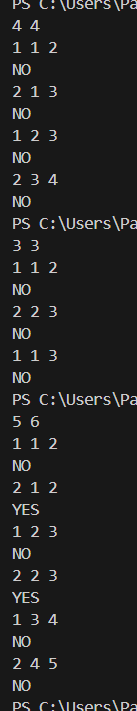
        }

    }

    return 0;

}

Output:



**Time Complexity:**

* Best/Average/Worst: O(q log n) due to union-find with path compression.

**Space Complexity:**

* O(n) for storing the parent and rank arrays.