**VIRTUAL CONTEST ON CODEFORCES**

|  |  |  |
| --- | --- | --- |
| Problem Number | Problem Name | Page |
| 1 | Weird Sum | 2-3 |
| 2 | Almost Ternary Matrix | 3-4 |
| 3 | Imbalanced Array | 5-7 |
| 4 | Vlad and a Sum of Sum of Digits | 7-8 |
| 5 | Closest Cities | 9-11 |

Problem-1 : Weird Sum

Code:

#include<bits/stdc++.h>

#define ll long long

using namespace std;

ll calculate\_contri (vector<int>& pos) {

sort(pos.begin(), pos.end()); // Sort pos

ll total = 0;

ll prefix\_sum = 0;

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

total += (ll)pos[i] \* i - prefix\_sum;

prefix\_sum += pos[i];

}

return total;

}

int main() {

int n, m;

cin >> n >> m;

// Table input

vector<vector<int>> table(n, vector<int>(m));

// Map to store pos for each color

unordered\_map<int, vector<pair<int, int>>> color\_pos;

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

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

cin >> table[i][j];

color\_pos[table[i][j]].emplace\_back(i, j);

}

}

ll total\_distance = 0;

// Process each color

for (auto& entry : color\_pos) {

vector<int> rows, cols;

for (auto& pos : entry.second) {

rows.push\_back(pos.first);

cols.push\_back(pos.second);

}

total\_distance += calculate\_contri (rows);

total\_distance += calculate\_ contri (cols);

}

cout << total\_distance << endl;

return 0;

}

**Analysis:**

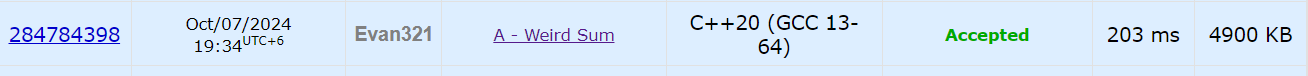
This C++ program calculates the total "distance contribution" for all cells in a grid based on the positions of each unique color. The program reads a grid of size `n x m`, where each cell contains a color value. It stores the positions of each color in an unordered map `color\_pos`, where the key is the color, and the value is a vector of pairs representing the row and column positions where the color appears.

For each color, it calculates the contribution to the total distance by separately analyzing the row and column positions. The function `calculate\_contribution` is responsible for computing the sum of Manhattan distances between all pairs of positions along one dimension (either rows or columns). It first sorts the positions, then iterates through them, calculating the distance contribution by considering the prefix sums (partial sums of the positions). This avoids the need for a brute-force double loop over all pairs of positions, making the calculation more efficient. Finally, the program outputs the total distance contribution across all colors.

The code uses optimizations such as sorting and prefix sum to calculate contributions efficiently and leverages standard C++ features like `vector`, `unordered\_map`, and custom macros for compact code.

**Time Complexity:** O(n \*m log (n \* m)).

**Submission:**



**Problem-2 : Almost Ternary Matrix**

**Code:**

#include <bits/stdc++.h>

#define ll long long int

using namespace std;

void Lakshyasolve(){

ll n, m;

cin >> n >> m;

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

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

ll x1 = i / 2, x2 = j / 2;

ll y1 = (x1 + x2) & 1, y2 = ((i & 1) + (j & 1)) & 1;

cout << (y1 ^ y2) << " ";

}

cout << endl;

}

}

int main(){ ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

ll T = 1;

cin >> T;

while (T--) {

Lakshyasolve();

}

}

**Analysis:**

 **Input Parsing**:

* The program reads the number of test cases T.
* For each test case, it reads two integers n and m, representing the dimensions of a grid (with n rows and m columns).

 **Pattern Generation Logic**:

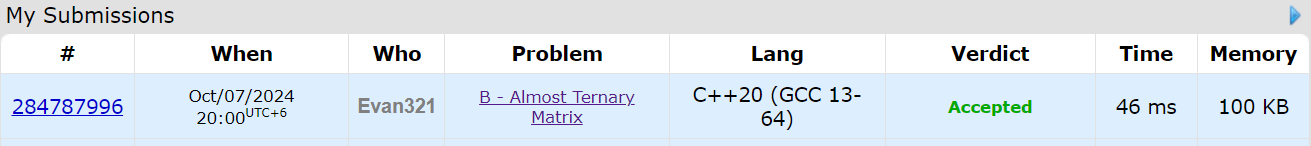
* For each test case, the program iterates over each cell (i, j) in the n x m grid. For each cell:
  + It calculates x1 as the integer division i / 2 and x2 as j / 2.
  + Then, it calculates y1 as (x1 + x2) & 1, which adds x1 and x2, and takes the bitwise AND with 1. This operation determines whether the sum is odd or even.
  + Next, it calculates y2 as ((i & 1) + (j & 1)) & 1, which adds the parity bits of i and j and checks if the result is odd or even.
  + Finally, the program outputs the result of y1 ^ y2, where ^ is the bitwise XOR, effectively determining whether the final output for that cell is 0 or 1.

 **Output**:

* For each test case, the program prints an n x m grid of 0 and 1 based on the above logic.

**Time Complexity: O(T \* n \* m)**

**Submission:**

****

**Problem-3 : Imbalanced Array**

**Code:**

#include <bits/stdc++.h>

#define ll long long

using namespace std;

ll Imbalance(int n, vecll& a) {

vector<int> left\_max(n), right\_max(n), left\_min(n), right\_min(n);

stack<int> st;

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

while (!st.empty() && a[st.top()] < a[i]) st.pop();

left\_max[i] = st.empty() ? 0 : st.top() + 1;

st.push(i);

}

while (!st.empty()) st.pop();

for (int i = n - 1; i >= 0; --i) {

while (!st.empty() && a[st.top()] <= a[i]) st.pop();

right\_max[i] = st.empty() ? n - 1 : st.top() - 1;

st.push(i);

}

while (!st.empty()) st.pop();

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

while (!st.empty() && a[st.top()] > a[i]) st.pop();

left\_min[i] = st.empty() ? 0 : st.top() + 1;

st.push(i);

}

while (!st.empty()) st.pop();

for (int i = n - 1; i >= 0; --i) {

while (!st.empty() && a[st.top()] >= a[i]) st.pop();

right\_min[i] = st.empty() ? n - 1 : st.top() - 1;

st.push(i);

}

ll total\_imbalance = 0;

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

ll max\_contrib = (ll)(i - left\_max[i] + 1) \* (right\_max[i] - i + 1);

ll min\_contrib = (ll)(i - left\_min[i] + 1) \* (right\_min[i] - i + 1);

total\_imbalance += a[i] \* (max\_contrib - min\_contrib);

}

return total\_imbalance;

}

int main() {

int n;

cin >> n;

vecll a(n);

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

cin >> a[i];

}

cout << Imbalance(n, a) << endl;

return 0;

}

**Analysis:**

 **Input**:

* The input consists of a single integer n, the size of the array, followed by n integers representing the array a.

 **Key Concept**:

* The "imbalance" is calculated by determining how much each element contributes to the total imbalance. The contribution of each element depends on how many subarrays treat that element as the maximum and as the minimum.
* The program uses two main ideas:
  + **Maximum Contribution**: For each element a[i], determine the number of subarrays in which a[i] is the maximum element.
  + **Minimum Contribution**: For each element a[i], determine the number of subarrays in which a[i] is the minimum element.

 **Stack-based Calculation**:

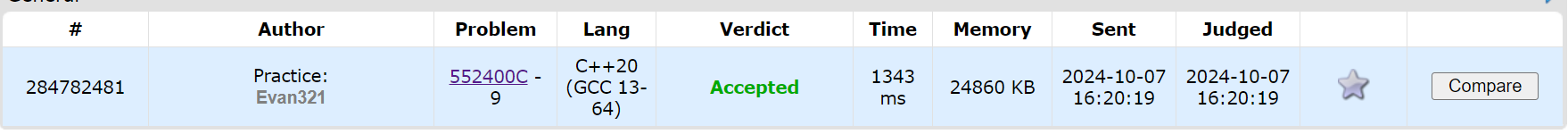
* The program uses **monotonic stacks** to efficiently compute the range of subarrays where each element is either the maximum or the minimum.
* **Left and Right Maximum**:
  + The arrays left\_max and right\_max store the boundaries of subarrays where each element a[i] is the maximum.
  + left\_max[i] represents the starting index of the leftmost subarray where a[i] is the maximum.
  + right\_max[i] represents the ending index of the rightmost subarray where a[i] is the maximum.
* **Left and Right Minimum**:
  + Similarly, the arrays left\_min and right\_min store the boundaries where each element a[i] is the minimum.

 **Calculating Total Imbalance**:

* For each element a[i], the program calculates how many subarrays treat a[i] as the maximum (max\_contrib) and how many subarrays treat it as the minimum (min\_contrib).
* The contribution of each element to the total imbalance is then proportional to the difference between max\_contrib and min\_contrib.
* This contribution is added to the total\_imbalance.

**Time Complexity: O(n)**

**Submission:**

****

**Problem-4:** **Vlad and a Sum of Sum of Digits**

**Code:**

#include<bits/stdc++.h>

#define vecll vector<long long>

#define ll long long

using namespace std;

int SOD(int a){

int sum = 0;

while(a){

sum += a%10;

a/=10;

}

return sum;

}

vecll preSum(200005);

int main(){

preSum[0] = 0;

for(int i=1; i<200005; i++){

preSum[i] = preSum[i-1] + SOD(i);

}

int t;

cin >> t;

while(t--){

int n;

cin >> n;

cout << preSum[n] << endl;

}

}

**Analysis:**

 **Sum of Digits (SOD) Function**:

* The function SOD(int a) calculates the sum of the digits of a number a. It repeatedly extracts the last digit of a using a % 10, adds it to a running sum (sum), and then removes the last digit using a /= 10. The function returns the sum of digits of a.

 **Prefix Sum Array**:

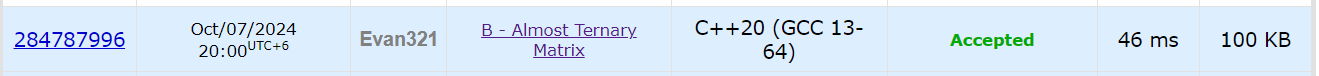
* The program defines a global vector preSum of size 200005 to store the prefix sums of digit sums for all integers from 1 to 200004. This means preSum[i] will store the sum of the digit sums from 1 to i.
* The prefix sum array is initialized in a loop before handling the test cases. For each index i, the value preSum[i] is set to preSum[i-1] + SOD(i), ensuring that preSum[i] contains the cumulative sum of digit sums up to i.

 **Main Program Execution**:

* The program first computes the preSum array.
* It then reads the number of test cases t.
* For each test case, it reads an integer n and directly outputs the value of preSum[n], which is the sum of digit sums for all integers from 1 to n.

**Time Complexity: O(t)**

**Submission:**

****

**Problem-5 : Closest Cities**

**Code:**

#include<bits/stdc++.h>

#define vec vector<int>

#define ll long long

using namespace std;

int main(){

int t;

cin >> t;

while(t--){

int n;

cin >> n;

vec c(n);

for(auto &i:c) cin >> i;

vector<int> right(n);

right[0] = 0;

right[1] = 1;

for(int i=2; i<n; i++){

if(c[i]-c[i-1] < c[i-1]-c[i-2]){

right[i] = right[i-1] + 1;

}else{

right[i] = right[i-1] + c[i]-c[i-1];

}

}

vector<int> left(n);

left[n-1] = 0;

left[n-2] = 1;

for(int i=n-3; i>=0; i--){

if(c[i+1]-c[i] < c[i+2]-c[i+1]){

left[i] = left[i+1] + 1;

}else{

left[i] = left[i+1] + c[i+1]-c[i];

}

}

int m; cin >> m;

while(m--){

int x,y; cin >> x >> y;

x -= 1;

y -= 1;

if(x<y) cout << right[y] - right[x] << endl;

else cout << left[y]-left[x] << endl;

}

}

}

**Analysis:**

 **Input Parsing**:

* The program reads multiple test cases (t). For each test case, the size of the city n is input, followed by an array c of length n that represents certain characteristics of the city (possibly elevation, distances, or other factors).
* For each test case, a number of queries m are also input, where each query consists of two points x and y, indicating a subrange of the city on which computations are to be performed.

 **Precomputing Arrays**:

* Two helper arrays, right and left, are precomputed:
  + right[i]: This array helps to compute the number of coins or difficulty when traversing from the start (leftmost) towards the right (i to i+1).
  + left[i]: This array helps to compute the difficulty when traversing from the end (rightmost) towards the left (i to i-1).

The logic in these precomputations seems to consider:

* If the difference in city elevation (or a property of consecutive elements) is decreasing (i.e., c[i] - c[i-1] < c[i-1] - c[i-2]), the difficulty increases. Otherwise, the increase is based on the actual difference between consecutive elements.

 **Query Handling**:

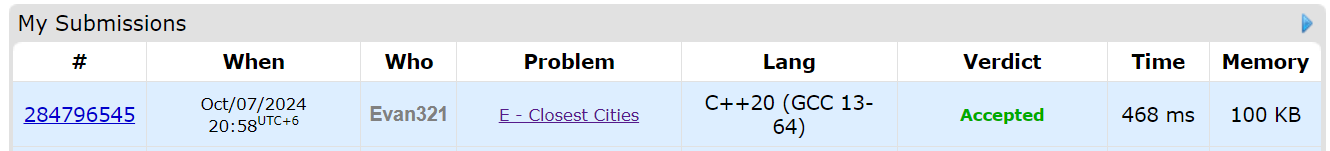
* For each query (x, y), the program checks whether the direction of travel is from left to right (x < y) or from right to left (x > y):
  + **If x < y**: It uses the right array to compute the number of coins required, calculating right[y] - right[x] (coins needed from position x to y).
  + **If x > y**: It uses the left array, computing left[y] - left[x].

 **Efficiency**:

* The key optimization is the use of the right and left arrays, which allow the program to compute the number of coins for any subrange in constant time **O(1)** per query, once the arrays are precomputed.
* The precomputation of the right and left arrays takes **O(n)** time for each test case, where n is the size of the city.

**Time Complexity: O(t \* (n + m))**

**Submission:**

****