IMC Coding Competition 2024 Solutions

UNSW CPMSoc

25 March 2024

Contents

1 Minnemax [Beginner Division only]

Algorithm

In the case that $a \geq b$, we have $-a \leq -b$, and so $\max(a, b) + \min(-a, -b) = a - a = 0$.

Otherwise $a < b$, and so we have $-a > -b$, and so $\max(a, b) + \min(-a, -b) = b - b = 0$.

Thus the answer is always 0.

Implementation Notes

- We don't need to read the input.

Implementation in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << "0\n";
}
```

Implementation in Python

```python
a, b = map(int, input().split())
print(0)
```

Alternative Solution

We can compute $\max(a, b)$ and $\min(-a, -b)$ then compute their sum.

2 Tim Estable [Beginner Division only]

Algorithm

For every number in the table, we need to check if it's the same as the product.

Implementation Notes

- We don't need to store the input inside a two-dimensional array.
- Also, the starting and ending values of the `for` loop can be 1 and n.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

int n, num, ans = 0;
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cin >> num;
            if (num == i*j) {
                ans++;
            }
        }
    }
    cout << ans << '\n';
}
```

3 Glowing Trees [Beginner Division only]

Algorithm

It can be seen from the input that the width and the height of the whole tree are $2N-1$ and $N+\lfloor N/3 \rfloor+1$ respectively. In the top part of the tree, on the ath line (1-indexed), the number of `.` on each side is $\frac{1}{2}(width-a)$. The trunk is always the same as the first row of the tree.

Implementation Notes

- We need to use a string to get the input.
- We don't need to store the answer inside an array or string.
- Also we can create a function that draws a single line, and use it repeatedly to draw the tree.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

void printLine(int width, int tree) {
    for (int i = 0; i < (width-tree)/2; i++) cout << '.';
    for (int i = 0; i < tree; i++) cout << '*';
    for (int i = 0; i < (width-tree)/2; i++) cout << '.';
    cout << "\n";
}

int n;
int main () {
    cin >> n;
    for (int i = 1; i <= n; i++) printLine(n*2-1, i*2-1);
    for (int i = 1; i <= 1+(n/3); i++) printLine(n*2-1, 1);
}
```

4 Tringle

Algorithm

Let $a \leq b \leq c$ be the three side lengths.

First, check whether the three given sides can form a triangle. They can form a triangle iff $a + b > c$.

Then, check whether the triangle is equilateral. A triangle is equilateral iff $a = b = c$.

Then, check whether the triangle is isosceles. A triangle is isosceles iff $a = b$ or $a = c$ or $b = c$.

Then, check whether the triangle is right-angled. A triangle is right-angled iff $a^2 + b^2 = c^2$.

Otherwise, the triangle is scalene.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define int ll

int a,b,c;

signed main(){
    cin >> a >> b >> c;
    if(a+b<=c or b+c<=a or a+c<=b){
        cout << "NOT A TRIANGLE" << "\n";
    }
    else if(a==b and b==c){
        cout << "EQUILATERAL" << "\n";
    }
    else if(a==b or b==c or a==c){
        cout << "ISOSCELES" << "\n";
    }
    else if(a*a+b*b==c*c or a*a+c*c==b*b or b*b+c*c==a*a){
        cout << "RIGHT ANGLED" << "\n";
    }
    else{
        cout << "SCALENE" << "\n";
    }
    return 0;
}
```

5 Insertion

Algorithm

We see that if the answer is YES, then there exists some (possibly empty) prefix of Y (call this substring a), and some (possibly empty) suffix of Y (call this substring b) such that $a + b = X$ (where $+$ is the concatenation of strings).

Actually, we can just check how much of the prefix of Y matches X (let the length of the maximum prefix matching be i), and how much of the suffix of Y matches X (let the length of the maximum suffix matching be j). Then:

If $i + j \geq |X|$, the answer is YES.

Otherwise, the answer is NO.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define int ll

int a,b;
string x,y;
int pref,suff;

signed main(){
    cin >> a >> b;
    cin >> x >> y;
    pref = a;
    suff = a;
    for(int i=0;i<a;i++){
        if(x[i]!=y[i]){
            pref = i;
            break;
        }
    }
    for(int i=0;i<a;i++){
        if(x[a-i-1]!=y[b-i-1]){
            suff = i;
            break;
        }
    }
    if(pref+suff>=a){
        cout << "YES" << "\n";
    }
    else{
        cout << "NO" << "\n";
    }
    return 0;
}
```

6 Parcels

Algorithm

For each column, we can keep track of the highest position where there exists a parcel. Then, when dropping a vertical parcel, the value increases by 2, while when dropping a horizontal parcel, the value for both columns will be set to the maximum value of either column + 1.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define int ll

int n;
int arr[8];
bool ans[200005][8];

signed main(){
    cin >> n;
    for(int i=1;i<=n;i++){
        int a,b;
        cin >> a >> b;
        if(a==1){
            arr[b] += 2;
            ans[arr[b]][b] = true;
            ans[arr[b]-1][b] = true;
        }
        else{
            int x = max(arr[b],arr[b+1]);
            arr[b] = x+1;
            arr[b+1] = x+1;
            ans[arr[b]][b] = true;
            ans[arr[b+1]][b+1] = true;
        }
    }
    bool yes = false;
    for(int i=2e5;i>=1;i--){
        for(int j=1;j<=7;j++){
            if(ans[i][j]){
                yes = true;
            }
        }
        if(!yes){
            continue;
        }
        for(int j=1;j<=7;j++){
            if(ans[i][j]){
                cout << '*';
            }
            else{
                cout << '-';
            }
        }
        cout << "\n";
    }
    return 0;
}
```

7 Weird Numbers

Algorithm

Let $f(n)$ be the first integer greater than or equal to n which *might* be a non-weird number.

We will now define $f(n)$. If n is non-weird, then $f(n) = n$. Otherwise, let s be the string-form of n, where $s[0]$ is the least significant digit. Then, let i be the maximum integer such that $s[i] = s[i+1]$. In this case, $f(n) = n + 10^i - n\%10^i$ (where % is the modulo operation).

We can note some properties of $f(n)$:

- Any number between n and $f(n) - 1$ (inclusive) is weird. This is because $s[i] == s[i+1]$ for every number between n and $f(n) - 1$ (using s and i as defined above).
- Let $f^2(n) = f(f(n))$, $f^3(n) = f(f(f(n)))$ and so on. Then, $f^k(n)$ will be non-weird for some $k \leq 2|s|$. To prove this, note that if n is an integer with the special property $0 = s[0] = s[1] = s[2] = s[3]... = s[p] \neq s[p+1] \neq s[p+2]... \neq s[|s|-1]$, then k is approximately $p/2 < |s|$. Also, the value of i must increase until the above special property is reached, which means i increases at most $|s|$ times. So, k is at most $|s| + |s| = 2|s|$.

So, to find the first non-weird number greater than n, just take the first k such that $f^k(n+1)$ is non-weird.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define int ll

int n;
int pow10[19];

int weird(int x){
    string s = to_string(x);
    for(int i=0;i<s.length()-1;i++){
        if(s[i]==s[i+1]){
            return i+1;
        }
    }
    return 0;
}

signed main(){
    pow10[0] = 1;
    for(int i=1;i<=18;i++){
        pow10[i] = pow10[i-1]*10;
    }
    cin >> n;
    n++;
    while(weird(n)){
        int x = weird(n);
        string s = to_string(n);
        n += pow10[s.length()-x-1];
        n -= n%pow10[s.length()-x-1];
    }
    cout << n << "\n";
    return 0;
}
```

8 Nuts [Beginner Division only]

Algorithm

Since the total number of nuts taken after ath step is $\frac{1}{2}a(a+1)$ we can find the total number of turns using a binary search in $O(\log N)$ time. Then we can find the total nuts taken by the last squirrel using $\frac{1}{2} \times number\_first + number\_last \times steps$. This can be done in constant time. Also, don't forget to subtract the extra nuts counted towards the answer.

Implementation Notes

- Use two parameters for storing the left and right ends. Make sure to pick the correct side in the binary search.
- The first amount is equal to $last\_number \pmod 3$. Also, it is not necessary to make the first amount 3 if the number gives $0 \pmod 3$.
- Make sure to use `long long` for all of the variables if writing a solution in C++ since the values are bigger than 10^9.

Implementation in C++

```
#include <bits/stdc++.h>
using ll = long long;
using namespace std;

ll findSum(ll a) {
    return a * (a+1)/2;
}

ll n, cnt[3];
int main () {
    cin >> n;
    ll l = 1, r = 1e9, numberOfRotations = n;
    while (l <= r) {
        ll mid = (l+r)/2;
        if (findSum(mid) >= n) {
            numberOfRotations = mid;
            r = mid-1;
        } else {
            l = mid+1;
        }
    }
    ll startingNumber = numberOfRotations%3;
    ll numberOfStacks = (numberOfRotations - startingNumber) / 3 + 1;
    ll sum = (startingNumber+numberOfRotations)*numberOfStacks/2;
    ll ans = sum + n - findSum(numberOfRotations);
    cout << ans << '\n';
}
```

9 Paper [Advanced Division only]

Algorithm

The next fold in a valid folding will either be an UP fold or a LEFT fold. An UP fold will be a long line of all V or M creases running left to right in the center of the horizontal grid. A LEFT fold will be a long line of all V or M creases running top to bottom in center of the vertical grid. We use this to determine the type and direction of the next fold.

The creases of the paper will "reflect" across this fold line i.e it will appear symmetrical except all creases are inverted across the line of symmetry. That is, all V will be converted into M and all M will be converted into V. So if there is a central fold line, and all creases reflect across this fold line, then the current fold is valid.

We can check the next fold by discarding one of the halves of the paper and recursing. Creases will be accessed $O(RC + RC/2 + RC/4 + RC/8 + ... + 1) = O(2RC) = O(RC)$ times. Hence the algorithm runs in $O(RC)$ time.

Implementation Notes

- Checking whether the next fold is UP and if it is valid can be done with one sweep each of the horizontal and vertical grids. Likewise with LEFT.
- When checking UP for example, creases on opposite sides must be inverted, in other words if for any (i, j) in the vertical grid $v[i][j] = v[R - i][j]$ holds, then the fold does not reflect, hence the fold is invalid.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

int R, C, v[1505][1505], h[1505][1505];
vector<string> ans;

bool u(bool mountain) {
    if (R == 1) return false;
    for (int i = 1; i <= R; i++) {
        for (int j = 1; j <= C-1; j++) {
            if (v[i][j] == v[R-i+1][j]) return false;
        }
    }
    for (int i = 1; i <= R-1; i++) {
        for (int j = 1; j <= C; j++) {
            if (i == R-i) {
                if (h[i][j] != mountain) return false;
            } else {
                if (h[i][j] == h[R-i][j]) return false;
            }
        }
    }
    ans.push_back(mountain ? "UM" : "UV");
    R /= 2;
    return true;
}

bool l(bool mountain) {
    if (C == 1) return false;
    for (int i = 1; i <= R; i++) {
        for (int j = 1; j <= C-1; j++) {
            if (j == C-j) {
                if (v[i][j] != mountain) return false;
            } else {
```

```
                if (v[i][j] == v[i][C-j]) return false;
            }
        }
    }
    for (int i = 1; i <= R-1; i++) {
        for (int j = 1; j <= C; j++) {
            if (h[i][j] == h[i][C-j+1]) return false;
        }
    }
    ans.push_back(mountain ? "LM" : "LV");
    C /= 2;
    return true;
}

int main() {
    cin >> R >> C;
    for (int i = 1; i <= R; i++) {
        for (int j = 1; j <= C-1; j++) {
            cin >> v[i][j];
        }
    }
    for (int i = 1; i <= R-1; i++) {
        for (int j = 1; j <= C; j++) {
            cin >> h[i][j];
        }
    }
    while (u(0) || u(1) || l(0) || l(1));
    if (R == 1 && C == 1) {
        cout << "YES\n";
        for (string x: ans) {
            cout << x << "\n";
        }
    } else {
        cout << "NO\n";
    }
}
```

10 Seven Bag

Algorithm

The answer is always smaller than the length of the string. From this, we can use binary search to find the answer in $O(n \log n)$ time. To check if the bag size could be $7x$ or larger, we just have to break the original sequence into blocks of size $7x$ and count the number of occurrences for each letter. Every letter's frequency cannot exceed x.

Implementation Notes

- Use two parameters for storing the left and right ends. Make sure to pick the correct side in the binary search.
- We can use an array to count the occurrence of letters. Also a C++ `map` will do the job just fine. There are at most 7 different letters, which means the `map` will also work in almost constant time.
- If using a global variable to count the frequency, make sure to remove them after each check.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

int n;
string s;
int main () {
    cin >> n >> s;
    int ans = n;
    int cnt[200];
    memset(cnt, 0, sizeof(cnt));

    int l = 1, r = n;
    while (l <= r) {
        int mid = (l+r)/2;
        int curSize = mid*7;
        bool possible = true;
        for (int st = 0; st*curSize < n && possible; st++) {
                for (int j = st*curSize; j < min(n, st+curSize); j++) {
                        if (++cnt[s[j]] > mid) {
                                possible = false;
                        }
                }
                for (int j = st*curSize; j < st+curSize; j++) {
                        cnt[s[j]]--;
                }
        }
        if (possible) {
            r = mid-1;
            ans = mid;
        } else {
            l = mid+1;
        }
    }

    cout << ans*7 << '\n';
}
```

Alternative Solution

We can try every bag size which is a multiple of 7, and use prefix sums over the frequencies of each letter to check that each block has the right frequency of each letter. This gives a time complexity of $O(n/7 + n/14 + \ldots n/n)$, which is $O(n \log n)$ by the growth rate of the harmonic series.

Alternative Implementation in C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int N, pre[7][300005];
char S[300005], pieces[] = "IOTJSZL";
int main() {
    cin >> N >> (S+1);
    for (int piece = 0; piece < 7; piece++) {
        for (int i = 1; i <= N; i++) {
            pre[piece][i] = pre[piece][i-1] + (S[i] == pieces[piece]);
        }
    }
    int sz = 0;
    bool works = false;
    while (!works) {
        sz += 7;
        works = true;
        for (int i = 1; i <= N; i += sz) { // i is first element in bag
            for (int piece = 0; piece < 7; piece++) {
                if (pre[piece][min(i+sz-1, N)] - pre[piece][i-1] > sz/7) {
                    works = false;
                    break;
                }
            }
            if (!works) break;
        }
    }
    cout << sz << "\n";
}
```

11 Merchandise

Algorithm

We will start off with every item costing \$0 and adding \$1 each time to some item.

Lemma: When given an array of item costs, it is better to add \$1 to the cheapest item than any other item.

Proof: Assume the minimum cost in the array is x. Then, increasing x by 1 increases the total sum of squares by $(x+1)^2 - x^2 = 2x+1$. Increasing any other cost i by 1 will result in increasing the total sum of squares by $(i+1)^2 - i^2 = 2i+1$. Since $x \leq i$, we have $2x+1 \leq 2i+1$, and since by increasing either of x or i will both increase the total sum of costs by exactly 1, it is always better to choose x to increase as it increases the total sum of squares by the least amount.

With the lemma in mind, we can first sort the customers by c_i. Then, we can iterate through each customer, and assume that we only care about the first c_i items. We keep track of these items a set, where the set keeps track of all the costs of each of the first c_i items. When increasing from c_i to c_{i+1}, we can just add $c_{i+1} - c_i$ 0's to the set.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define int ll

int n,c,tot,ans;
vector<vector<int>> v;
multiset<int> s;

signed main(){
    cin >> n >> c;
    for(int i=1;i<=c;i++){
        int a,b;
        cin >> a >> b;
        v.push_back({a,b});
    }
    sort(v.begin(),v.end());
    for(vector<int> i:v){
        int a = i[0], b = i[1];
        while(s.size()!=a){
            s.insert(0);
        }
        while(tot<b){
            int p = *s.begin();
            s.erase(s.find(p));
            ans += 2*p+1;
            s.insert(p+1);
            tot++;
        }
    }
    cout << ans << "\n";
    return 0;
}
```

12 Buildings [Advanced Division only]

Algorithm

The simplest solution uses the slope trick. In fact, the problem is a classic application of the slope trick.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int N;

int main()
{
        cin >> N;
        ll ans = 0;
        priority_queue<ll> pq;
        for (int i = 0; i < N; i++)
        {
                ll x;
                cin >> x;
                pq.push(x);
                if (pq.top() > x) {
                        ans += pq.top() - x;
                        pq.pop();
                        pq.push(x);
                }
        }
        cout << ans;
        return 0;
}
```

Alternative Solution

Observe that for a non-increasing sequence, the best we can do is adjust all values to their median. This idea leads to a solution involving sweeping from left to right, keeping track of chunks of values we plan to adjust to their median, and merging chunks of values together if the left median is greater than the right median. To keep track of chunks of values, we use a stack of median-tracking sets, and to do this efficiently, we use small-to-large merging.

This solution was originally devised before realising that slope trick could be applied. The proof of its correctness is left as an exercise.

Alternative Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

int N, A;
multiset<int> ms[100005];
multiset<int>::iterator lowermed[100005], uppermed[100005];
vector<int> chunks;
int minmed[100005];

void insert(int i, int x) {
    ms[i].insert(x);
    if (ms[i].size() % 2 == 0) {
        if (x >= *uppermed[i]) uppermed[i]++;
        if (x < *lowermed[i]) lowermed[i]--;
    } else {
        if (x >= *lowermed[i]) lowermed[i]++;
```

```
        if (x < *uppermed[i]) uppermed[i]--;
    }
}

int main() {
    cin >> N;
    for (int i = 1; i <= N; i++) {
        cin >> A;
        ms[i].insert(A);
        lowermed[i] = ms[i].begin();
        uppermed[i] = ms[i].begin();
        minmed[i] = max(
            chunks.empty() ? 0 : minmed[chunks.back()],
            *lowermed[i]
        );
        chunks.push_back(i);
        while (true) {
            if (chunks.size() < 2) break;
            int a = chunks[chunks.size()-2];
            int b = chunks[chunks.size()-1];
            if (minmed[a] <= *uppermed[b]) break;
            if (ms[a].size() < ms[b].size()) swap(a, b);
            for (int x: ms[b]) {
                insert(a, x);
            }
            chunks.pop_back();
            chunks.pop_back();
            minmed[a] = max(
                chunks.empty() ? 0 : minmed[chunks.back()],
                *lowermed[a]
            );
            chunks.push_back(a);
        }
    }
    long long ans = 0;
    for (int i: chunks) {
        for (int x: ms[i]) {
            ans += abs(x - *lowermed[i]);
        }
    }
    cout << ans << "\n";
}
```

13 Tim's Dance [Advanced Division only]

Algorithm

If two distinct combination of switches make the grid equal, then there must exist a subset of these switches such that when they are turned on, nothing in the grid is turned on. (This argument can be formalised by viewing each switch as XOR operations of its row and column and then rearranging the equation formed by any 2 subset of switches that yield the same grid.) We seek to find properties of these special subsets.

These special subsets come in 2 types:

- A square grid with one light turned on in every column and row. For example, a square grid with the diagonal turned on.
- A cycle in the grid. For example, the lights on the corners of rectangle turned on. As another example, lights at $\{\{1,2\},\{1,3\},\{2,1\},\{2,2\},\{3,1\},\{3,3\}\}$ turned on will also do nothing. Notice that any cycle of odd length can always be reduced to even length due to the fact it's on a grid.

This problem then reduces to finding the largest subset of switches S such that none of these special subsets exist as a subset of S. Since for every cycle in the grid/graph, we must erase at least one node in the cycle, we can greedily loop through each node and add it to S (initially empty) if it does not form any cycles within S. This can be done via DSU in $O(n\alpha(n))$ time total. If it's a square grid and there's something in every row and column, the answer is $2^{|S|-1}$, otherwise it will be $2^{|S|}$ (since every subset of S creates a different configuration).

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 100010;
const int MOD = 1000000007;
int R, C, n;
struct UF
{
        int rep[MAXN];
        UF()
        {
                for (int i = 0; i < MAXN; i++)
                {
                        rep[i] = i;
                }
        }
        int findrep(int a)
        {
                if (rep[a] == a)
                        return a;
                return rep[a] = findrep(rep[a]);
        }
        void merge(int a, int b) { rep[findrep(a)] = findrep(b); }
};
UF uf;
vector<int> at[2][MAXN];
set<pair<int, int>> points;
int main()
{
        scanf("%d%d%d", &R, &C, &n);
        if (R == 1 || C == 1)
        {
                for (int i = 0; i < n; i++)
                {
                        int r, c;
                        scanf("%d%d", &r, &c);
                        points.insert({r, c});
```

```
                }
                int num = points.size();
                if ((R + C) % 2 == 0 && num == R + C - 1)
                        num--;
                int ans = 1;
                while (num--)
                {
                        ans *= 2;
                        ans %= MOD;
                }
                printf("%d\n", ans);
                return 0;
        }

        int ans = 1;
        int u = 0;
        set<int> rows, cols;
        for (int i = 0; i < n; i++)
        {
                int r, c;
                scanf("%d%d", &r, &c);
                rows.insert(r);
                cols.insert(c);
                if (points.find({r, c}) != points.end())
                        continue;
                points.insert({r, c});
                set<int> mergeWith;
                int extraEdges = 0;
                if (at[0][r].size())
                {
                        extraEdges++;
                        mergeWith.insert(uf.findrep(at[0][r][0]));
                }
                if (at[1][c].size())
                {
                        extraEdges++;
                        mergeWith.insert(uf.findrep(at[1][c][0]));
                }
                if (mergeWith.size() == extraEdges)
                {
                        for (auto a : mergeWith)
                                uf.merge(i, a);
                        u++;
                        at[0][r].push_back(i);
                        at[1][c].push_back(i);
                }
        }
        if (R == C) {
                if (rows.size() == R and cols.size() == C) u--;
        }
        while (u--)
        {
                ans *= 2;
                ans %= MOD;
        }
        printf("%d\n", ans);
}
```

14 IMC Banner V [Advanced Division only]

Algorithm

We shall use divide and conquer with convolution via FFT (Fast Fourier Transform). The number of evenly-spaced `IMC` that exist entirely within the range $[l, r]$ is equivalent to the sum of (where $m = \lfloor (l+r)/2 \rfloor$):

1. number of `IMC` that exist such that the `I` is within $[l, m]$ and the `C` is within $[m+1, r]$
2. the number of `IMC` that exist entirely within $[l, m]$
3. the number of `IMC` that exist entirely within $[m+1, r]$

We can solve for 2) and 3) recursively. We can also solve for 1) with one convolution, since if there exists a `M` at position i, then the number of `IMC` that uses that `M` is simply the convolution

$$\sum_{a+b=2i} A_a B_b$$

where A is an array that ranges from $[l, r]$ that has 0 everywhere except for positions i where $l \le i \le m$ and $S_i =$ `I`, in which case $A_i = 1$. B is an array that ranges from $[l, r]$ that has 0 everywhere except for positions i where $m+1 \le i \le r$ and $S_i =$ `C`, in which case $B_i = 1$. Hence, this expression finds the number of pairs of `I` that appear in the left half and `C` that appear in the second half that are equidistant to this `M`. To do this for every possible `M` fast, we use convolution via FFT, which is a common technique.

The recursive divide and conquer has $O(\log n)$ layers. Each layer in the divide and conquer in total takes $O(n \log n)$ time since FFT is $O(n \log n)$, and hence, the algorithm runs in $O(n \log^2 n)$.

Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()

typedef complex<double> C;
typedef vector<double> vd;
typedef vector<int> vi;
void fft(vector<C>& a) {
        int n = sz(a), L = 31 - __builtin_clz(n);
        static vector<complex<long double>> R(2, 1);
        static vector<C> rt(2, 1);  // (^ 10% faster if double)
        for (static int k = 2; k < n; k *= 2) {
                R.resize(n); rt.resize(n);
                auto x = polar(1.0L, acos(-1.0L) / k);
                rep(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
        }
        vi rev(n);
        rep(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
        rep(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
        for (int k = 1; k < n; k *= 2)
                for (int i = 0; i < n; i += 2 * k) rep(j,0,k) {
                        // C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
/// include-line
                        auto x = (double *)&rt[j+k], y = (double *)&a[i+j+k];
/// exclude-line
                        C z(x[0]*y[0] - x[1]*y[1], x[0]*y[1] + x[1]*y[0]);
/// exclude-line
                        a[i + j + k] = a[i + j] - z;
                        a[i + j] += z;
                }
}
vd conv(const vd& a, const vd& b) {
```

```
        if (a.empty() || b.empty()) return {};
        vd res(sz(a) + sz(b) - 1);
        int L = 32 - __builtin_clz(sz(res)), n = 1 << L;
        vector<C> in(n), out(n);
        copy(all(a), begin(in));
        rep(i,0,sz(b)) in[i].imag(b[i]);
        fft(in);
        for (C& x : in) x *= x;
        rep(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
        fft(out);
        rep(i,0,sz(res)) res[i] = imag(out[i]) / (4 * n);
        return res;
}

const int MAXN = 1000100;
char inp[2*MAXN];
long long ans;
void solve(int s, int e) {
    if (s >= e) return;
    int m = (s+e)/2;
    vd a, b;
    for (int i = s; i <= m; i++) {
        a.push_back(inp[i] == 'I');
        b.push_back(0);
    }
    for (int i = m+1; i <= e; i++) {
        b.push_back(inp[i] == 'C');
        a.push_back(0);
    }
    auto res = conv(a, b);
    for (int i = s; i <= e; i++) {
        int ind = i-s;
        if (inp[i] == 'M' && 2*ind < res.size()) {
            int add = round(res[2*ind]);

            ans += add;
        }
    }

    solve(s, m);
    solve(m+1, e);
}
int main() {
    int n;
    scanf("%d", &n);
    scanf(" %s", inp);
    solve(0, n-1);
    printf("%lld\n", ans);
}
```