Indigo Coding Contest Unofficial Editorial

Stefan Dascalescu

April 2023

I decided to write an unofficial editorial for this contest as I tested it and I found tasks to be interesting and enjoyable for people of levels up to USACO Gold. If you want to check more of my work or you're looking for a tutor, visit my website for more details. You can upsolve the problems here.

1 The Adventures of Curious George

We start by processing the strings according to the rule described in the statement. First, for each string read, we count the number of letters equal to 'y' and attach the remaining letters as they come. After we traverse the string, we attach a number of letters 'y' equal to the count we found earlier on.

Once we processed everything, we are going to sort the strings using any sorting algorithm according to the criterias given in the statement. In my solution I used std::sort together with a comparator to sort them in O(n log n) time.

```
1
2  #include <bits/stdc++.h>
3
4  using namespace std;
5
6  bool cmp(string a, string b)
7  {
8      if(a.size() != b.size())
9          return a.size() > b.size();
10     return a < b;
11 }
12 int main() {
13     int t;
14     cin >> t;
15     vector<string> words;
16     for(; t; t--)
17     {
18         string s;
19         int cntY = 0;
20         cin >> s;
21         string s2;
22         for(int i = 0; i < s.size(); i++)
23         {
24             if(s[i] == 'x')
25                 continue;
26             if(s[i] == 'y')
27                 cntY++;
28             else
29                 s2 += s[i];
30         }
31         for(int i = 0; i < cntY; i++)
32             s2 += 'y';
33         words.push_back(s2);
34     }
35     sort(words.begin(), words.end(), cmp);
36     for(auto x : words)
37         cout << x << '\n';
38     return 0;
39 }
```

2 Delicious Math

Finding PI using various functions or in built functions can lead to precision errors, so the method I used to solve this task was to google search the first 200 digits of PI, add them in a string and for each string, check how many positions are matched with the positions at the beginning of the string which has PI.

```
1
2  #include <cmath>
3  #include <cstdio>
4  #include <vector>
5  #include <iostream>
6  #include <algorithm>
7  using namespace std;
8
9
10 int main() {
11
12     // PI, the string is actually on the same line but I put it like that for
       readability reasons
13
14     string s = "31415926535897932384626433832795028841971693993751058209 7
15     94459230781640628620899862803482534211706798214808651328230 6
16     64709384460955058223172535940812848111 7
17     45028410270193852110555964462294895493038196";
18
19     int t;
20     cin >> t;
21     for(; t; t--)
22     {
23         string s2;
24         cin >> s2;
25
26         int ans = 0;
27         while(ans < s2.size() && s2[ans] == s[ans])
28             ans++;
29         cout << ans << '\n';
30     }
31     return 0;
32 }
```

3 Breaking Ground

For this problem one has to find the number of squares at the edge of the board, which is $4 * n - 4$ (there are four sides, each with n squares, but corners show up on two edges so we need to subtract 4 to not count them twice) and the remaining squares which are not on the edges. Then we need to check if the total cost is smaller or equal than the budget required.

```
1
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      long long n, b, x, y;
7      cin >> n >> b >> x >> y;
8
9      long long cnt = 4 * n - 4;
10
11     if(cnt * y + (n * n - cnt) * x <= b)
12         cout << "Walter will like this lab";
13     else
14         cout << "Sorry Mr. Fring, no can do";
15
16     return 0;
17 }
```

4 Ship Navigation

For this problem we need to traverse the string and count the length of each streak of consecutive equal characters. Then, depending on the character and on the length of the streak, we can compute the contribution to the final answer using the formulas given in the statement.

```
#include <cmath>
#include <cstdio>
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;

pair<int, int> sol;

void calc(int len, char tip)
{
    if(tip == '<')
    {
        if(len <= 2)
            sol.first -= 2 * len;
        else
        {
            sol.first -= 20;
            sol.first -= 5 * (len - 3);
        }
    }

    if(tip == '>')
    {
        if(len <= 2)
            sol.first += 2 * len;
        else
        {
            sol.first += 20;
            sol.first += 5 * (len - 3);
        }
    }

    if(tip == '^')
    {
        if(len <= 2)
            sol.second += 2 * len;
        else
        {
            sol.second += 20;
            sol.second += 5 * (len - 3);
        }
    }

    if(tip == 'v')
    {
        if(len <= 2)
            sol.second -= 2 * len;
        else
        {
            sol.second -= 20;
            sol.second -= 5 * (len - 3);
        }
    }
}
int main() {

    string s;
    cin >> s;

    int str = 1;
    char tip = s[0];

    for(int i = 1; i < s.size(); i++)
```

```
    {
        if(s[i] == s[i-1])
            str++;
        else
        {
            calc(str, tip);
            str = 1;
            tip = s[i];
        }
    }
    calc(str, tip);

    cout << "(" << sol.first << "," << sol.second << ")" << '\n';
    return 0;
}
```

5 Building Blocks

Let's start from a naive solution which simulates the wall additions in reverse order. In order to reconstruct the additions, we would find the maximum value from the array and subtract one from that value, as well as from all of the other values adjacent to that maximum which are equal to it. If there are more such intervals, we can subtract 1 from either of them.

For example, if the array is $[2, 1, 3, 3, 1, 1, 2]$, we would subtract 1 from each of the values equal to 3 and get $[2, 1, 2, 2, 1, 1, 2]$.

Now, the main observation we get from this idea is we have two adjacent values a and b, with b ¿ a, we will have $b - a$ operations we will use on b which won't be used on a. Therefore, we will add to the answer $b - a$ every time we have a pair of values such that the current value is greater than the previous value, which leads us to the optimal solution in $O(n)$.

```

#include <iostream>

using namespace std;


int main()
{
    int n;
    cin >> n;

    int prv = 0;

    int ans = 0;

    for(int i = 1; i <= n; i++)
    {
        int x;
        cin >> x;
        if(x >= prv)
            ans += x - prv;
        prv = x;
    }

    cout << ans;
    return 0;
}
```

6 Burning Trees

For this problem we can run a BFS or a DFS from the starting node and at the end, find the farthest away node using a for loop, while making sure to print the node with the largest value. The complexity will be $O(n+m)$

```
#include <vector>
#include <iostream>
#include <queue>
using namespace std;

int b, n;
vector<int> v[1030];
vector<int> viz(1030), dist(1030);

int main() {
    cin >> b;
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        int a, b;
        cin >> a >> b;
        v[a].push_back(b);
        v[b].push_back(a);
    }

    viz[b] = 1;
    queue<int> q;
    q.push(b);

    while(!q.empty())
    {
        int node = q.front();
        q.pop();
        for(auto x : v[node])
            if(!viz[x])
            {
                viz[x] = 1;
                dist[x] = dist[node] + 1;
                q.push(x);
            }
    }

    int maxi = b;

    for(int i = 1; i <= 1000; i++)
        if(viz[i] && dist[i] >= dist[maxi])
            maxi = i;

    cout << maxi;
    return 0;
}
```

7 Octopus Shenanigans

For this problem we can observe that we don't have many distinct characters and the size of the grid isn't big either so we can use a brute-force like solution where we keep in each state of the queue the position we are currently at, as well as the bitmask of the letters we have chosen so far. Basically for each letter I codified it with a number from 0 to 25 and then I created a mask with the sums of powers of two corresponding to each letter's position in the alphabet.

In order to simulate this brute force I used BFS in order to keep track of the maximum answer as well. The complexity is $O(2^{sigma} * sigma)$, where $sigma$ is the number of letters in the alphabet.

```cpp

#include<bits/stdc++.h>
#pragma GCC optimize("O3")
#define fi first
#define se second
#define pb push_back
#define pf push_front

using namespace std;

typedef long long ll;

const int mod = 1000000007;
const double PI = 3.14159265359;
const double eps = 1e-9;

char a[22][22];

int ox[] = {0, 1, 0, -1};
int oy[] = {-1, 0, 1, 0};


int main()
{

    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m;
    cin >> n >> m;

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
            cin >> a[i][j];

    queue<pair<int, pair<int, int> > > q;

    q.push({(1<<(a[1][1] - 'A')), {1, 1}});

    int maxi = 0;

    while(!q.empty())
    {
        pair<int, pair<int, int> > node = q.front();
        q.pop();

        maxi = max(maxi, __builtin_popcount(node.first)); // how many bits equal to 1
    the mask has

        for(int dir = 0; dir <= 3; dir++)
        {
            int nxt_x = node.second.first + ox[dir];
            int nxt_y = node.second.second + oy[dir];

            if(nxt_x >= 1 && nxt_x <= n && nxt_y >= 1 && nxt_y <= m)
            {
                if(node.first & (1<<(a[nxt_x][nxt_y] - 'A')))
                    continue;
                q.push({node.first + (1<<(a[nxt_x][nxt_y] - 'A')), {nxt_x, nxt_y}});
            }
        }
    }

    cout << maxi << '\n';
    return 0;
}
```

8 Bracket Reconstruction

This problem can be solved using a rather standard dp approach, by keeping states for each pair of (position, difference), where the difference is the difference between the count of '(' and the count of ')'. Let's define $dp[i][j]$ as the number of ways to construct a string up to position i from the string S if the difference is equal to j. Then, for each state, we will update the states based on how we fill the characters at positions $i * 2$ and $i * 2 + 1$, the cases being shown in the code.

In the end we find the answer in $dp[n][0]$.

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

typedef long long ll;

const int mod = 1000000007;
int n, dp[2002][4002];

int main()
{

    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    cin >> n;
    string s;
    cin >> s;
    s = ' ' + s;

    dp[0][0] = 1;

    for(int i = 0; i < n; i++)
        for(int dif = 0; dif <= 2*n; dif++)
        {
            if(s[i+1] == '?' || s[i+1] == '(')
            {
                dp[i+1][dif+2] += dp[i][dif];  // ((
                if(dp[i+1][dif+2] >= mod)
                    dp[i+1][dif+2] -= mod;
                dp[i+1][dif] += dp[i][dif]; // ()
                if(dp[i+1][dif] >= mod)
                    dp[i+1][dif] -= mod;
            }

            if(s[i+1] == '?' || s[i+1] == ')')
            {

                if(dif >= 2)
                {
                    dp[i+1][dif-2] += dp[i][dif];  // ))
                    if(dp[i+1][dif-2] >= mod)
                        dp[i+1][dif-2] -= mod;
                    if(i + 1 != n)
                    {
                        dp[i+1][dif] += dp[i][dif]; // )(
                        if(dp[i+1][dif] >= mod)
                            dp[i+1][dif] -= mod;
                    }
                }
            }
        }

    cout << dp[n][0] << '\n';
    return 0;
}
```

9 Bracket Reconstruction 2: Electric Boogaloo

I want to thank Mohamed Bakry for explaining the editorial for this problem. His code is attached below.

We begin by constructing the complete string, in order to do that we add a ? character after each given character in S.

A key observation consists in the fact that in order to make S a balanced bracket sequence, every '(', '', and '[' in an odd position will be matched with a character in an even position, and every ')', '', and ']' in an odd position will be matched with a character in an even position.

If an open bracket matched a closed bracket and both are placed in positions of the same parity, then S can't be a balanced bracket sequence because the length of the string would be odd and a balanced bracket sequence must have an even length. Since all the characters at even positions are '?', then we can just change them to make each character match its opposing character in an odd position.

Thus, we can basically consider all '' and '[' as '(' and '' and ']' as ').

The problem now becomes similar to its easy version, except that if a matched pair is ('?', '?') then there are three ways, otherwise, there's only one way.

So basically we will use a similar dp approach to the one used in the easy version, except that if we choose '?' in an odd position then we multiply the ways by 3 since we know from the previous observations that we will match it with another '?' from an even position.

```

#pragma GCC optimize("Ofast,no-stack-protector,unroll-loops,fast-math,O3")

#include <bits/stdc++.h>

using namespace std ;

const int mod = 1e9 + 7 ;

int Add(int x , int y)
{
    int z = x + y ;
    if(z >= mod)
        z -= mod ;
    return z ;
}

int Sub(int x , int y)
{
    int z = x - y ;
    if(z < 0)
        z += mod ;
    return z ;
}

int Mul(int x , int y)
{
    return (x * 1ll * y) % mod ;
}

int powlog(int base , int power)
{
    if(power == 0)
        return 1 ;
    int x = powlog(base , power / 2) ;
    x = Mul(x , x) ;
    if(power & 1)
        x = Mul(x , base) ;
    return x ;
}

int modinv(int x)
{
    return powlog(x , mod-2) ;
}

int Div(int x , int y)
```

```
{
    return Mul(x , modinv(y)) ;
}

const int MAX = 2004 ;

int arr[MAX] ;
int n ;

string s2 , s ;

int dp[2*MAX][MAX] ;

int solve(int idx , int cnt)
{
    if(cnt < 0 || cnt > n)
        return 0 ;
    if(idx == 2*n)
        return (cnt == 0) ;
    int &ret = dp[idx][cnt] ;
    if(ret != -1)
        return ret ;
    ret = 0 ;
    if(s[idx] == '(')
        ret = Add(ret , solve(idx+1 , cnt+1)) ;
    if(s[idx] == ')')
        ret = Add(ret , solve(idx+1 , cnt-1)) ;
    if(s[idx] == '?')
    {
        if(idx%2 == 0)
        {
            ret = Add(ret , Mul(3 , solve(idx+1 , cnt+1))) ;
            ret = Add(ret , Mul(3 , solve(idx+1 , cnt-1))) ;
        }
        else
        {
            ret = Add(ret , solve(idx+1 , cnt+1)) ;
            ret = Add(ret , solve(idx+1 , cnt-1)) ;
        }
    }
    return ret ;
}

int main()
{
    memset(dp , -1 , sizeof(dp)) ;
    ios_base::sync_with_stdio(0) ;
    cin.tie(0) ;
    cin>>n ;
    cin>>s2 ;
    s = "" ;
    for(auto &c : s2)
    {
        if(c == '(' || c == '[' || c == '{')
            s.push_back('(') ;
        else if(c == ')' || c == ']' || c == '}')
            s.push_back(')') ;
        else
            s.push_back('?') ;
        s.push_back('?') ;
    }
    return cout<<solve(0 , 0)<<"\n" , 0 ;
}
```

10 Go Forth and Attack

For this problem we can observe that the number of columns is very small, so we can think at something similar to a DP on broken profiles. Thus, in order to make it easier for me while solving, I rotated the grid in order to turn the solution into something more similar to this CSES problem.

Then, given that we can't place a haybale within the previous two columns of the previous one on the same line, I used a ternary mask in order to know for each row the last column at which I added the haybale. For each row, 0, 1 and 2 marked the distance at which the last haybale was placed.

Knowing this, I used DP and $dp[i][j]$ represented the maximum answer if we filled the first i columns in such a way that the ternary mask of the column was j. In order to update the states on line i, I brute forced the valid configurations of haybales from the same column (there can be at most 4 haybales used on the same column) and then I iterated through the masks in order to check whichever masks were valid, thus updating the states.

For more details, please check the code attached below.

```

#include<bits/stdc++.h>
#pragma GCC optimize("O3")
#define fi first
#define se second
#define pb push_back
#define pf push_front

// #define fisier 1

using namespace std;

typedef long long ll;

const int mod = 1000000007;
const double PI = 3.14159265359;
const double eps = 1e-9;

char str[102][12], rotstr[12][102];

int n, m, dp[102][60002]; // dp[i][j] = max number of haybales if the last filled haybales have mask j

int p3; // max_mask;

// mask j is a ternary mask with each digit on position x being the number of places last haybale was placed on line x (0, 1, 2 places before)

// O(n * valid_configs * valid_masks)

void bkt(int col, int pos, vector<int> positions)
{
    for(int x = pos; x <= m; x++)
        if(rotstr[x][col] == 'D')
        {
            positions.push_back(x);
            bkt(col, x + 3, positions);
            positions.pop_back();
        }

    bool mrkd[11] = {0};

    for(auto x : positions)
        mrkd[x-1] = 1;

    for(int msk = 0; msk <= p3; msk++)
    {
        int val = msk;
        bool valid = 1;
        int totmsk = 0;
        int pp = 1;
        for(int ln = 0; ln < m; ln++)
        {
            if(!mrkd[ln])
```

```
            {
                int rest = val % 3 + 1;
                if(rest == 3)
                    rest = 2;
                totmsk += rest * pp;
            }
            else
                if(val % 3 < 2)
                    valid = 0;
            val /= 3;
            pp *= 3;
        }


        if(valid)
        {
            dp[col][totmsk] = max(dp[col][totmsk], dp[col-1][msk] + (int) positions.
    size());
        }

    }
}
int main()
{
    #ifdef fisier
        ifstream cin("input.in");
        ofstream cout("output.out");
    #endif

    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    cin >> n >> m;

    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= m; j++)
        {
            cin >> str[i][j];
            rotstr[j][i] = str[i][j];
        }

    for(int i = 1; i <= m; i++)
        p3 = p3 * 3 + 2;

    for(int i = 1; i <= n; i++) // iterate through columns
    {
        vector<int> valid;
        bkt(i, 1, valid);
    }

    int ansmax = 0;
    for(int i = 1; i <= n; i++)
        for(int j = 0; j <= p3; j++)
            ansmax = max(ansmax, dp[i][j]);

    cout << ansmax;
    return 0;
}
```