パナソニックプログラミングコンテスト2020 Editorial

writer: rng58_admin

2020/3/14

こんにちは。Admin の rng_58 です。

今回は問題を解く方針によって、簡単に解けることも、大変になってしまうこともあるような問題を集めてみました。

ぜひ、いろいろな解答を見比べてみてください。

A. Kth Term

考えられる入力が 32 通りしかないので、たとえば次のように解くことができます。

```
#include <iostream>

using namespace std;

int main(void){
        int K;
        cin >> K;

        if(K == 1) cout << 1 << endl;
        if(K == 2) cout << 1 << endl;
        :
        if(K == 32) cout << 51 << endl

        return 0;
}
```

配列を使うと、よりシンプルに書くことができます。

```
#include <iostream>

using namespace std;

int main(void){
        int K;
        cin >> K;

        int a[] = {1, 1, 1, 2, 1, 2, 1, 5, 2, 2, 1, 5, 1, 2, 1, 14, 1, 5, 1, 5, 2, 2,
            1, 15, 2, 2, 5, 4, 1, 4, 1, 51};
        cout << a[K-1] << endl;

        return 0;
}
```

B. Bishop

サンプルの図を見ると、全体の半分のマス目に到達できることが分かります。また、全体に奇数マスある場合は切り上げます。

一般に、正整数 x, y に対し、x/y を切り上げた値は $(x+y-1)/y$ (ここで / は整数除算を表す) として実装することができます。overflow に注意すると、たとえば次のようにかけます。

```
#include <iostream>

using namespace std;
typedef long long ll;

int main(void){
        ll H,W;
        cin >> H >> W;

        cout << (H * W + 1) / 2 << endl;

        return 0;
}
```

残念ながら、これは WA となってしまいます。たとえば $(H, W) = (1, 10^9)$ のときを考えてみてください。上の解法では 500000000 が出力されますが、実際は角は全く動けないので答えは 1 となります。このような例外をコーナーケースといいます。

```
#include <iostream>

using namespace std;
typedef long long ll;

int main(void){
        ll H,W;
        cin >> H >> W;

        if(H == 1 || W == 1){
                cout << 1 << endl;
        } else {
                cout << (H * W + 1) / 2 << endl;
        }

        return 0;
}
```

コーナーケースを避けるコツは

- 解法をしっかりと証明する (この問題であれば、半分のマスに到達可能であるのはなぜか証明を書いてみる)
- Constraints の下限のほうにも注意する

などがあります。

C. Sqrt Inequality

自然に実装すると、たとえば次のようになります。

```
#include <iostream>
#include <cmath>

using namespace std;
typedef long long ll;

int main(void){
	ll a,b,c;
	cin >> a >> b >> c;

	if(sqrt(a) + sqrt(b) < sqrt(c)){
		cout << "Yes" << endl;
	} else {
		cout << "No" << endl;
	}

	return 0;
}
```

残念ながら、これはたとえば $(a, b, c) = (249999999, 250000000, 999999998)$ などのケースで WA となってしまいます。

次のコードで実験してみましょう。

```
#include <iostream>
#include <cmath>

using namespace std;
typedef long long ll;

int main(void){
	printf("%.30f\n", sqrt(249999999));
	printf("%.30f\n", sqrt(250000000));
	printf("%.30f\n", sqrt(249999999) + sqrt(250000000));
	printf("%.30f\n", sqrt(999999998));
	return 0;
}
```

実行結果は次の通りです。

```
15811.388269219120047637261450290680
15811.388300841896125348284840583801
31622.776570061017991974949836730957
31622.776570061017991974949836730957
```

本当は $\sqrt{249999999} + \sqrt{250000000} < \sqrt{999999998}$ なのですが、2 つの値が等しくなってしまっています。これはプログラムの内部では実数が近似的に扱われているためです。このように、実数を扱うときは誤差に注意する必要があります。

また、$\sqrt{a} + \sqrt{b} = \sqrt{c}$ であるときでも誤差により $\sqrt{a} + \sqrt{b} < \sqrt{c}$ と判定してしまうことがあります。

これらのことに注意して、さらにより精度の高い long double 型を使うと、AC を得ることができます。

```
#include <iostream>
#include <cmath>

using namespace std;
typedef long long ll;

int main(void){
	long double a,b,c;
	cin >> a >> b >> c;

	long double eps = 1.0E-14;

	if(sqrt(a) + sqrt(b) + eps < sqrt(c)){
		cout << "Yes" << endl;
	} else {
		cout << "No" << endl;
	}

	return 0;
}
```

ここで $\varepsilon = 10^{-14}$ の値を大きくしすぎたり小さくしすぎたりすると WA になってしまいます。なぜ $\varepsilon = 10^{-14}$ であるとうまくいくのかは説明できるのですが、より簡単で安全な方法は、全て整数でやってしまうことです。

$$\sqrt{a} + \sqrt{b} < \sqrt{c} \tag{1}$$
$$\Leftrightarrow \quad (\sqrt{a} + \sqrt{b})^2 < c \tag{2}$$
$$\Leftrightarrow \quad 2\sqrt{ab} < c - a - b \tag{3}$$
$$\Leftrightarrow \quad c - a - b > 0 \land 4ab < (c - a - b)^2 \tag{4}$$

```
#include <iostream>
#include <cmath>

using namespace std;
typedef long long ll;

int main(void){
	ll a,b,c;

	cin >> a >> b >> c;

	ll d = c - a - b;
	if(d > 0 && d * d > 4 * a * b){
		cout << "Yes" << endl;
	} else {
		cout << "No" << endl;
	}

	return 0;
}
```

D. String Equivalence

for ループや、複雑なものになると bitmask や next_permutation を使った全探索はよく出題されています。今回は、それより複雑な構造を全て生成する必要があります。このようなときは DFS が有用です。

文字列 $s = s_1 s_2 \ldots s_N$ が標準形であることの条件を言い換えると、

- $s_1 = a$
- $s_k \leq \max\{s_1, \ldots, s_{k-1}\} + 1 (\forall 2 \leq k \leq N)$

となります。(ここで $+1$ は辞書順で一つ次の文字を表します。)

これは、たとえば次のように実装できます。

```cpp
#include <iostream>
#include <string>

using namespace std;

int N;

void dfs(string s, char mx){
        if(s.length() == N){
                printf("%s\n", s.c_str());
        } else {
                for(char c='a';c<=mx;c++){
                        dfs(s + c, ((c == mx) ? (char)(mx + 1) : mx));
                }
        }
}

int main(void){
        cin >> N;
        dfs("", 'a');
        return 0;
}
```

E. Three Substrings

直感的に、次のような greedy が思いつきやすいかもしれません。

- s の中で a, b, c がどの順番に現れるかを考える。たとえば、左端が $a \to b \to c$ の順であるとする。(6 通りの順を全探索する。)
- a は s の左端であるとしてよい。
- b を a の情報と矛盾しないようにできる限り左に寄せておく。
- c を a, b の情報と矛盾しないようにできる限り左に寄せておく。

実はこれには反例があります。
たとえば $??c? \to ac?a \to ?b?a$ の順で左端から寄せておくと、

- $??c?$ (??c? を配置)
- $acca$ (ac?a を重ねて配置)
- $accab?a$ (?b?a をできる限り左に配置)

となりますが、最適解は

- $??c?$ (??c? を配置)
- $?ac?a$ (ac?a をわざと一つずらして配置)
- $?acbaa$ (?b?a をできる限り左に配置)

となります。
このように、直感に頼った未証明の Greedy は危険です。

- Greedy を証明してから使う
- 全探索や DP など他の方法を検討する

などが大切です。

今回の場合は、a の位置を基準に、b, c の相対的な位置を全探索すれば安全に解けることが分かります。a の左端の位置を 0 としたとき、他の文字列の左端の位置は -4000 から 4000 まで動く可能性があることに注意してください。

```
#include <iostream>
#include <string>

using namespace std;

#define REP(i,n) for((i)=0;(i)<(int)(n);(i)++)

#define M 2000

bool ab[100000],ac[100000],bc[100000];

bool match(char c1, char c2){
        return (c1 == '?' || c2 == '?' || c1 == c2);
}

int main(void){
        int i,j;

        string a,b,c;
        cin >> a >> b >> c;

        int A = a.length();
        int B = b.length();
        int C = c.length();

        REP(i,A) REP(j,B) if(!match(a[i], b[j])) ab[i-j+50000] = true;
        REP(i,A) REP(j,C) if(!match(a[i], c[j])) ac[i-j+50000] = true;
        REP(i,B) REP(j,C) if(!match(b[i], c[j])) bc[i-j+50000] = true;

        int ans = 3 * M;
        for(i=-2*M;i<=2*M;i++) for(j=-2*M;j<=2*M;j++) if(!ab[i+50000] && !ac[j+50000] &&
            !bc[j-i+50000]){
                int L = min(0, min(i, j));
                int R = max(A, max(B + i, C + j));
                ans = min(ans, R - L);
        }

        cout << ans << endl;

        return 0;
}
```

F. Fractal Shortest Path

一辺 3^{30} のグリッド上の二点間の距離を求める問題です。まずは一辺 3^{29} のブロック 9 個に分割して、次のように名前を付けます。

A	B	C
D	-	E
F	G	H

二点がどのブロックに属すかによって場合わけしていきます。

- 両方 A のとき、最短路が A 内で完結していることがわかるので、再帰的にフラクタルのレベルが 1 少ない問題を解けばよいです。
- A と E のときは、最短路は A の右下隅と E の左上隅を通るとしてよいことがわかるので、A, E 内の問題を解けばよいです。
- B と H のときは、対称性より A と E のときと同様であることがわかります (このように場合わけを減らす工夫をしていくとミス無く、短時間で解けるようになっていきます。)
- :

A と B のときなどは少し複雑ですがこのように場合わけしていけば解くことができます。
ただし、この後も方針の選択によって大変さが代わってきます。いろいろな方針が考えられると思いますが、たとえば次のようなものが比較的楽だと思います。
紙の上での場合わけの結果、次のような性質が成り立つことが分かります。

フ

ラクタル内の黒マスは正方形状の連結成分に分かれています。ここで、ある一つの連結成分のみに注目し、それ以外の黒マスをすべて白マスにしてしまったときの二点間の距離を考えます。実は、このような形で得られる二点間の距離の最大値が答えであることが (上記のように場合分けをしていくことによって) わかります。

各 t について、

- 二点間を妨害するような一辺 3^t の黒マスの連結成分はあるか？
- あるならば、それによってどれだけ迂回する必要があるか？

と考えていくと次のようにかけます。(get_dist は座標を 0-based に変換してから呼んでください。)

```
typedef long long ll;
#define _abs(x) ((x)>0?(x):-(x))

bool between(ll x, ll y1, ll y2){
	if(y1 > y2) swap(y1, y2);
	y1++; y2--;
	if(!(y1 <= y2)) return false;

	while(x > 0){
		if(x % 3 == 1){
			if(y2 - y1 > 3) return true;
			for(ll y=y1;y<=y2;y++) if(y % 3 == 1) return true;
		}
		x /= 3;
		y1 /= 3;
		y2 /= 3;
	}

	return false;
}

ll get_extra(ll x1, ll y1, ll x2, ll y2){
	ll ans = 0;
	int i;
	ll three = 1;

	REP(i,35){
		if(x1 / three == x2 / three && between(x1 / three, y1 / three, y2 / three)){
			ll tmp = min(min(x1 % three, x2 % three) + 1, three - max(x1 % three, x2 % three));
			ans = max(ans, tmp);
		}
		three *= 3;
	}

	return ans;
}

ll get_dist(ll x1, ll y1, ll x2, ll y2){
	return _abs(x1 - x2) + _abs(y1 - y2) + 2 * max(get_extra(x1, y1, x2, y2), get_extra(y1, x1, y2, x2));
}
```