

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

For International Readers: English editorial starts on page 10.

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。

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void){
6     int K;
7     cin >> K;
8
9     if(K == 1) cout << 1 << endl;
10    if(K == 2) cout << 1 << endl;
11    :
12    if(K == 32) cout << 51 << endl
13
14    return 0;
15 }
```

配列を使うと、よりシンプルに書くことができます。

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void){
6     int K;
7     cin >> K;
8
9     int a[] = {1, 1, 1, 2, 1, 2, 1, 5, 2, 2, 1, 5, 1, 2, 1, 14, 1, 5, 1, 5, 2, 2,
10              1, 15, 2, 2, 5, 4, 1, 4, 1, 51};
11    cout << a[K-1] << endl;
12
13    return 0;
14 }
```

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、たとえば次のようかけます。

```
1 #include <iostream>
2
3 using namespace std;
4 typedef long long ll;
5
6 int main(void){
7     ll H,W;
8     cin >> H >> W;
9
10    cout << (H * W + 1) / 2 << endl;
11
12    return 0;
13 }
```

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。

```
1 #include <iostream>
2
3 using namespace std;
4 typedef long long ll;
5
6 int main(void){
7     ll H,W;
8     cin >> H >> W;
9
10    if(H == 1 || W == 1){
11        cout << 1 << endl;
12    } else {
13        cout << (H * W + 1) / 2 << endl;
14    }
15
16    return 0;
17 }
```

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。

```
1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     ll a,b,c;
9     cin >> a >> b >> c;
10
11     if(sqrt(a) + sqrt(b) < sqrt(c)){
12         cout << "Yes" << endl;
13     } else {
14         cout << "No" << endl;
15     }
16
17     return 0;
18 }
```

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。

```
1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     printf("%.30f\n", sqrt(249999999));
9     printf("%.30f\n", sqrt(250000000));
10    printf("%.30f\n", sqrt(249999999) + sqrt(250000000));
11    printf("%.30f\n", sqrt(999999998));
12    return 0;
13 }
```

実行結果は次の通りです。

```
1 15811.388269219120047637261450290680
2 15811.388300841896125348284840583801
3 31622.776570061017991974949836730957
4 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。

```

1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     long double a,b,c;
9     cin >> a >> b >> c;
10
11     long double eps = 1.0E-14;
12
13     if(sqrt(a) + sqrt(b) + eps < sqrt(c)){
14         cout << "Yes" << endl;
15     } else {
16         cout << "No" << endl;
17     }
18
19     return 0;
20 }

```

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。

$$\sqrt{a} + \sqrt{b} < \sqrt{c} \quad (1)$$

$$\Leftrightarrow (\sqrt{a} + \sqrt{b})^2 < c \quad (2)$$

$$\Leftrightarrow 2\sqrt{ab} < c - a - b \quad (3)$$

$$\Leftrightarrow c - a - b > 0 \wedge 4ab < (c - a - b)^2 \quad (4)$$

```

1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     ll a,b,c;
9
10    cin >> a >> b >> c;
11
12    ll d = c - a - b;
13    if(d > 0 && d * d > 4 * a * b){
14        cout << "Yes" << endl;
15    } else {
16        cout << "No" << endl;
17    }
18
19    return 0;
20 }

```

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 $s = s_1s_2 \dots s_N$ が標準形であることの条件を言い換えると、

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

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。

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int N;
7
8 void dfs(string s, char mx){
9     if(s.length() == N){
10         printf("%s\n", s.c_str());
11     } else {
12         for(char c='a';c<=mx;c++){
13             dfs(s + c, ((c == mx) ? (char)(mx + 1) : mx));
14         }
15     }
16 }
17
18 int main(void){
19     cin >> N;
20     dfs("", 'a');
21     return 0;
22 }
```

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 $a \rightarrow b \rightarrow c$ 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 $??c? \rightarrow ac?a \rightarrow ?b?a$ 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。

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 #define REP(i,n) for((i)=0;(i)<(int)(n);(i)++)
7
8 #define M 2000
9
10 bool ab[100000],ac[100000],bc[100000];
11
12 bool match(char c1, char c2){
13     return (c1 == '?' || c2 == '?' || c1 == c2);
14 }
15
16 int main(void){
17     int i,j;
18
19     string a,b,c;
20     cin >> a >> b >> c;
21
22     int A = a.length();
23     int B = b.length();
24     int C = c.length();
25
26     REP(i,A) REP(j,B) if(!match(a[i], b[j])) ab[i-j+50000] = true;
27     REP(i,A) REP(j,C) if(!match(a[i], c[j])) ac[i-j+50000] = true;
28     REP(i,B) REP(j,C) if(!match(b[i], c[j])) bc[i-j+50000] = true;
29
30     int ans = 3 * M;
31     for(i=-2*M;i<=2*M;i++) for(j=-2*M;j<=2*M;j++) if(!ab[i+50000] && !ac[j+50000] &&
32         !bc[j-i+50000]){
33         int L = min(0, min(i, j));
34         int R = max(A, max(B + i, C + j));
35         ans = min(ans, R - L);
36     }
37
38     cout << ans << endl;
39
40     return 0;
41 }
```

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変わってきます。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。)

```

1 typedef long long ll;
2 #define _abs(x) ((x)>0?(x):-x)
3
4 bool between(ll x, ll y1, ll y2){
5     if(y1 > y2) swap(y1, y2);
6     y1++; y2--;
7     if(!(y1 <= y2)) return false;
8
9     while(x > 0){
10        if(x % 3 == 1){
11            if(y2 - y1 > 3) return true;
12            for(ll y=y1;y<=y2;y++) if(y % 3 == 1) return true;
13        }
14        x /= 3;
15        y1 /= 3;
16        y2 /= 3;
17    }
18
19    return false;
20 }
21
22 ll get_extra(ll x1, ll y1, ll x2, ll y2){
23     ll ans = 0;
24     int i;
25     ll three = 1;
26
27     REP(i,35){
28         if(x1 / three == x2 / three && between(x1 / three, y1 / three, y2 / three)){
29             ll tmp = min(min(x1 % three, x2 % three) + 1, three - max(x1 % three,
30                 x2 % three));
31             ans = max(ans, tmp);
32         }
33         three *= 3;
34     }
35     return ans;
36 }
37
38 ll get_dist(ll x1, ll y1, ll x2, ll y2){
39     return _abs(x1 - x2) + _abs(y1 - y2) + 2 * max(get_extra(x1, y1, x2, y2), get_extra(
40         y1, x1, y2, x2));

```

Hello. I'm the Admin rng_58.

This time, I gathered the problems which can be solved either easily or arduously depending on which course you take.

By all means, please compare the various solutions.

A. Kth Term

Since there are only 32 kinds of input, for example it can be solved as follows.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void){
6     int K;
7     cin >> K;
8
9     if(K == 1) cout << 1 << endl;
10    if(K == 2) cout << 1 << endl;
11    :
12    if(K == 32) cout << 51 << endl
13
14    return 0;
15 }
```

By using arrays, it can be solved more simply.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(void){
6     int K;
7     cin >> K;
8
9     int a[] = {1, 1, 1, 2, 1, 2, 1, 5, 2, 2, 1, 5, 1, 2, 1, 14, 1, 5, 1, 5, 2, 2,
10              1, 15, 2, 2, 5, 4, 1, 4, 1, 51};
11    cout << a[K-1] << endl;
12
13    return 0;
14 }
```

B. Bishop

From the diagrams on the sample, it can be seen that the bishop can reach to the half the squares of all. Also, if there are odd number of squares overall, it is rounded up.

In general, for any positive integers x, y , the quotient x/y rounded up can be implemented as $(x + y - 1)/y$ (where $/$ denotes integer division). Paying attention to overflow, it can be written as follows for example.

```
1 #include <iostream>
2
3 using namespace std;
4 typedef long long ll;
5
6 int main(void){
7     ll H,W;
8     cin >> H >> W;
9
10    cout << (H * W + 1) / 2 << endl;
11
12    return 0;
13 }
```

Unfortunately, it will be WA. For example, consider the case of $(H, W) = (1, 10^9)$. The solution above outputs 500000000, but actually the bishop cannot move at all, so the answer is 1. Such exception is called corner cases.

```
1 #include <iostream>
2
3 using namespace std;
4 typedef long long ll;
5
6 int main(void){
7     ll H,W;
8     cin >> H >> W;
9
10    if(H == 1 || W == 1){
11        cout << 1 << endl;
12    } else {
13        cout << (H * W + 1) / 2 << endl;
14    }
15
16    return 0;
17 }
```

The key to avoid the corner cases are:

- to prove the solution properly (in this problem, try writing the proof why it can reach half the squares)
- to pay attention the lower bounds of the constraints

and so on.

C. Sqrt Inequality

By implementing naturally, it will be like as follows:

```
1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     ll a,b,c;
9     cin >> a >> b >> c;
10
11     if(sqrt(a) + sqrt(b) < sqrt(c)){
12         cout << "Yes" << endl;
13     } else {
14         cout << "No" << endl;
15     }
16
17     return 0;
18 }
```

Unfortunately, it will be WA in such case like $(a, b, c) = (249999999, 250000000, 999999998)$.

Let's experiment with the following code.

```
1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     printf("%.30f\n", sqrt(249999999));
9     printf("%.30f\n", sqrt(250000000));
10    printf("%.30f\n", sqrt(249999999) + sqrt(250000000));
11    printf("%.30f\n", sqrt(999999998));
12    return 0;
13 }
```

The execution result is as follows.

```
1 15811.388269219120047637261450290680
2 15811.388300841896125348284840583801
3 31622.776570061017991974949836730957
4 31622.776570061017991974949836730957
```

Although $\sqrt{249999999} + \sqrt{250000000} < \sqrt{999999998}$ actually, but the two output values are the same. This is because real numbers are treated approximately inside the program. Like this, when treating real numbers you have to pay attention to errors.

Also, even when $\sqrt{a} + \sqrt{b} = \sqrt{c}$ it may be judged as $\sqrt{a} + \sqrt{b} < \sqrt{c}$ because of errors.

Paying attention to those points, if you use long double type, which is more precise, you can get AC.

```

1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     long double a,b,c;
9     cin >> a >> b >> c;
10
11     long double eps = 1.0E-14;
12
13     if(sqrt(a) + sqrt(b) + eps < sqrt(c)){
14         cout << "Yes" << endl;
15     } else {
16         cout << "No" << endl;
17     }
18
19     return 0;
20 }

```

Here, if you set the value $\varepsilon = 10^{-14}$ to larger or smaller numbers, you will get WA. It can be explained why $\varepsilon = 10^{-14}$ is just right, but the easier and safer way is to use only integers.

$$\sqrt{a} + \sqrt{b} < \sqrt{c} \tag{5}$$

$$\Leftrightarrow (\sqrt{a} + \sqrt{b})^2 < c \tag{6}$$

$$\Leftrightarrow 2\sqrt{ab} < c - a - b \tag{7}$$

$$\Leftrightarrow c - a - b > 0 \wedge 4ab < (c - a - b)^2 \tag{8}$$

```

1 #include <iostream>
2 #include <cmath>
3
4 using namespace std;
5 typedef long long ll;
6
7 int main(void){
8     ll a,b,c;
9
10    cin >> a >> b >> c;
11
12    ll d = c - a - b;
13    if(d > 0 && d * d > 4 * a * b){
14        cout << "Yes" << endl;
15    } else {
16        cout << "No" << endl;
17    }
18
19    return 0;
20 }

```

D. String Equivalence

Brute-force problems that requires to use for loops, or more complex bitmask or next_permutation, are very common. This time you have to generate even more complex structure. In such case DFS is useful.

文字列 $s = s_1s_2 \dots s_N$ が標準形であることの条件を言い換えると、The conditions where a string $s = s_1s_2 \dots s_N$ is in normal form can be rephrased to:

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

Here, +1 denotes the next character lexicographically.

This can be implemented as follows for example.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int N;
7
8 void dfs(string s, char mx){
9     if(s.length() == N){
10         printf("%s\n", s.c_str());
11     } else {
12         for(char c='a';c<=mx;c++){
13             dfs(s + c, ((c == mx) ? (char)(mx + 1) : mx));
14         }
15     }
16 }
17
18 int main(void){
19     cin >> N;
20     dfs("", 'a');
21     return 0;
22 }
```

E. Three Substrings

Intuitively, the following greedy may be easy to come up with.

- Consider the order of appearances of a, b, c in s . For example, assume that the left appends are in the order of $a \rightarrow b \rightarrow c$. (Perform an exhaustive search.)
- It can be assumed that s is at the left end of s .
- Put b to as left as possible so as not to contradict to the information of a .
- Put b to as left as possible so as not to contradict to the information of a .

Actually there is a counterexample of this.

For example, if you apply the "as-left-as-possible strategy" to three strings $??c? \rightarrow ac?a \rightarrow ?b?a$, you will obtain

- $??c?$ (Place $??c?$)
- $acca$ (Place $ac?a$ overlapped)
- $accab?a$ (Place $?b?a$ as left as possible)

but the optimal solution is

- $??c?$ (Place $??c?$)
- $?ac?a$ (Place $ac?a$ to the different place intentionally)
- $?acbaa$ (Place $?b?a$ as left as possible).

Like this, intuitive unproved greedy is dangerous. Important things are such as

- to prove greedy before using, or
- to consider some other ways such as exhaustive search or DP.

This time, it can be seen that it can be solved safer by performing exhaustive search on the position of b, c relative to a . Note that, let the position of left end of a be 0, then the positions of left ends of the other strings is in the range of -4000 to 4000 .

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 #define REP(i,n) for((i)=0;(i)<(int)(n);(i)++)
7
8 #define M 2000
9
10 bool ab[100000],ac[100000],bc[100000];
11
12 bool match(char c1, char c2){
13     return (c1 == '?' || c2 == '?' || c1 == c2);
14 }
15
16 int main(void){
17     int i,j;
18
19     string a,b,c;
20     cin >> a >> b >> c;
21
22     int A = a.length();
23     int B = b.length();
24     int C = c.length();
25
26     REP(i,A) REP(j,B) if(!match(a[i], b[j])) ab[i-j+50000] = true;
27     REP(i,A) REP(j,C) if(!match(a[i], c[j])) ac[i-j+50000] = true;
28     REP(i,B) REP(j,C) if(!match(b[i], c[j])) bc[i-j+50000] = true;
29
30     int ans = 3 * M;
31     for(i=-2*M;i<=2*M;i++) for(j=-2*M;j<=2*M;j++) if(!ab[i+50000] && !ac[j+50000] &&
32         !bc[j-i+50000]){
33         int L = min(0, min(i, j));
34         int R = max(A, max(B + i, C + j));
35         ans = min(ans, R - L);
36     }
37
38     cout << ans << endl;
39
40     return 0;
41 }
```

F. Fractal Shortest Path

This is a problem of finding the distance in the grid of size 3^{30} . First, divide it into nine blocks of size 3^{29} and name them as follows:

A	B	C
D	-	E
F	G	H

Let us split into case depending on which block the two points belong.

- When both A, it can be seen that the shortest path completes in A, so it is enough to solve the problem of one level lower.
- When A and E, we can assume that the shortest path passes the right-bottom square of A and left-top square of E, so it is enough to solve the problem in A and E.
- when B and H, by symmetry it is the same to the problem in A and E. (Like this, trying to reduce the case splitting leads to decreasing the miss and faster solving.)
- :

When A and B, it is little more complex, but the problem can be solved by splitting like this.

However, depending on the choice of course, the difficulty varies. Many courses can be considered, but I think one like the following is relatively easy.

As a result of case splitting on paper, the following property can be seen:

T

he black squares in the fractal form square-shaped connected components. Here, focusing on only one connected component, consider the distance between the two points when all the other black squares are painted white. Actually, it appears that the maximum distance between the two points which are obtained by such way is the answer (by splitting into case as above).

For each t , consider:

- Is there a connected component of black squares of size 3^t such that obstructs the path between the two points?
- If exists, how much does it have to be detoured because of it?

then it can be calculated as follows. (When calling `get_dist`, convert the coordinates into 0-based ones.)

```

1 typedef long long ll;
2 #define _abs(x) ((x)>0?(x):-x)
3
4 bool between(ll x, ll y1, ll y2){
5     if(y1 > y2) swap(y1, y2);
6     y1++; y2--;
7     if(!(y1 <= y2)) return false;
8
9     while(x > 0){
10        if(x % 3 == 1){
11            if(y2 - y1 > 3) return true;
12            for(ll y=y1;y<=y2;y++) if(y % 3 == 1) return true;
13        }
14        x /= 3;
15        y1 /= 3;
16        y2 /= 3;
17    }
18
19    return false;
20 }
21
22 ll get_extra(ll x1, ll y1, ll x2, ll y2){
23     ll ans = 0;
24     int i;
25     ll three = 1;
26
27     REP(i,35){
28         if(x1 / three == x2 / three && between(x1 / three, y1 / three, y2 / three)){
29             ll tmp = min(min(x1 % three, x2 % three) + 1, three - max(x1 % three,
30                 x2 % three));
31             ans = max(ans, tmp);
32         }
33         three *= 3;
34     }
35     return ans;
36 }
37
38 ll get_dist(ll x1, ll y1, ll x2, ll y2){
39     return _abs(x1 - x2) + _abs(y1 - y2) + 2 * max(get_extra(x1, y1, x2, y2), get_extra(
40         y1, x1, y2, x2));

```
