M-SOLUTIONS プロコンオープン 2020 解説

E869120, square1001

2020 年 7 月 25 日

For International Readers: English editorial will be published in a few days.

こんにちは、本コンテストの writer を務めました、高校 3 年生の E869120、square1001 です。

この度は、コンテストに参加していただきありがとうございました。

今回は、今後 ABC や AGC で上位に入るために必要な解法テクニックや実装テクニックを入れた、教育的な問題を集めました。

様々な解法を解説に載せていますので、是非見比べてみてください。

A: Kyu in AtCoder

AtCoder の最高レーティングに応じて、級位・段位がつけられていることを、みなさんは知っていましたか？ プロフィールのページを見てみると、「12 級」や「初段」などが書かれていると思います。この問題は、400 以上 1999 以下のレーティング X を入力して、これが何級にあたるのかを出力するプログラムを書く問題です。

単純な解法

整数 X を入力した後は、問題文に書かれている通りに、次のように条件分岐すれば良いです。C++ や Python などのプログラミング言語では、条件分岐は if 文を用いて書くことができます。

- $400 \leq X$ かつ $X \leq 599$ のとき — 「8」と出力する
- $600 \leq X$ かつ $X \leq 799$ のとき — 「7」と出力する
- $800 \leq X$ かつ $X \leq 999$ のとき — 「6」と出力する
- $1000 \leq X$ かつ $X \leq 1199$ のとき — 「5」と出力する
- $1200 \leq X$ かつ $X \leq 1399$ のとき — 「4」と出力する
- $1400 \leq X$ かつ $X \leq 1599$ のとき — 「3」と出力する
- $1600 \leq X$ かつ $X \leq 1799$ のとき — 「2」と出力する
- $1800 \leq X$ かつ $X \leq 1999$ のとき — 「1」と出力する

（解説は次ページへ続きます）

たとえば、以下のような実装ができます。（C++ での実装例です）

```cpp
#include <iostream>
using namespace std;
int main() {
    int X; cin >> X;
    if (400 <= X && X <= 599) cout << "8" << endl;
    if (600 <= X && X <= 799) cout << "7" << endl;
    :
    if (1800 <= X && X <= 1999) cout << "1" << endl;
    return 0;
}
```

もっと "楽" な解法

実は、上のような複雑な条件分岐をしなくても、解く方法があるのです。レーティング 200 ごとに、AtCoder の級位が変わっていることに着目してみましょう。400 – 599、600 – 799、800 – 999、… と、レーティング 200 上がるにつれ級の値が 1 ずつ下がっていることがわかると思います。

このように考えると、(級位) = 10 - 「X を 200 で割った商」で計算できることが分かるでしょう。例えば、$X = 961$ の場合、$10 - 4 = 6$ 級、と計算することができます。

実装例 (C++)

```cpp
#include <iostream>
using namespace std;

int main() {
    int X; cin >> X;
    cout << 10 - X / 200 << endl;
    return 0;
}
```

実装例 (Python)

```python
print(10 - int(input()) // 200)
```

B. Magic 2

この問題には、大きく分けて以下の 2 つの解法があります。

- 全探索ベースの解法
- 貪欲法（greedy）ベースの解法

全探索ベースの解法

以下のように、操作に番号を付けることを考えます。

- 操作 1： 赤のカードに書かれた整数を 2 倍する。
- 操作 2： 緑のカードに書かれた整数を 2 倍する。
- 操作 3： 青のカードに書かれた整数を 2 倍する。

そこで、操作 1 を p 回、操作 2 を q 回、操作 3 を r 回行うとき、以下の通りになります。

- 赤のカードに最終的に書かれた整数： $A \times 2^p$
- 緑のカードに最終的に書かれた整数： $B \times 2^q$
- 青のカードに最終的に書かれた整数： $C \times 2^r$

このように、カードに書かれた整数は、各操作の回数 p, q, r の値に依存します。つまり、

$$A \times 2^p \ < \ B \times 2^q \ < \ C \times 2^r \ (p + q + r \leq K)$$

を満たす非負整数 (p, q, r) の組がある場合のみ、答えは Yes となることがわかります。$K \leq 7$ なので、三重ループを用いて全探索を行うと解くことができます。以下 C++ での実装例です。

```cpp
#include <iostream>
using namespace std;

int main() {
    int A, B, C, K; bool flag = false;
    cin >> A >> B >> C >> K;

    for (int i = 0; i <= K; i++) {
        for (int j = 0; j <= K; j++) {
            for (int k = 0; k <= K; k++) {
                int x = A * (1 << i), y = B * (1 << j), z = C * (1 << k);
                if (i + j + k <= K && x < y && y < z) flag = true;
            }
```

```
        }
    }

    if (flag == true) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

貪欲法ベースの解法

どんな方法を使えば、最短手数で魔術を成功できるか考えてみましょう。自明な考察として、

- 赤のカードには一切操作を行わない

ことが最適であることが分かります。なぜなら、赤のカードに操作を行った場合、他のカードに対して操作すべき回数が増えるからです。同様に考えて、緑のカードに対して行う操作も、(赤のカード) < (緑のカード) となるために必要な最小回数で構いません。

このように、その場その場で最善だと思われる手を選び、答えを得る手法を「貪欲法」といいます。以下 C++ での実装例です。

```cpp
#include <iostream>
using namespace std;

int main() {
    int A, B, C, K, cnt = 0;
    cin >> A >> B >> C >> K;
    while (A >= B) { cnt += 1; B *= 2; }
    while (B >= C) { cnt += 1; C *= 2; }

    if (cnt <= K) cout << "Yes" << endl;
    else cout << "No" << endl;
    return 0;
}
```

C. Marks

i 学期の評点を L_i とします。まず、$i-1$ 学期の評点 L_{i-1} と i 学期の評点 L_i を比べてみましょう。

$i-1$ 学期の評点 L_{i-1}	$A_{i-K} \times A_{i-K+1} \times A_{i-K+2} \times ... \times A_{i-1}$
i 学期の評点 L_i	$A_{i-K+1} \times A_{i-K+2} \times A_{i-K+3} \times ... \times A_i$

そこで、

$$\frac{L_i}{L_{i-1}} > 1$$

が満たされる時のみ i 学期の評点の方が $i-1$ 学期の評点より高いです。つまり、

$$\frac{A_{i-K+1} \times A_{i-K+2} \times A_{i-K+3} \times ... \times A_i}{A_{i-K} \times A_{i-K+1} \times A_{i-K+2} \times ... \times A_{i-1}} > 1$$

$$(\Leftrightarrow) \frac{A_i}{A_{i-K}} > 1$$

$$(\Leftrightarrow) A_i > A_{i-K}$$

が満たされるときのみ、i 学期の評点が $i-1$ 学期の評点に比べて高いことがわかります。この方法を用いると、実装は if 文を用いるだけで簡単です。以下 C++ での実装例です。

```cpp
#include <iostream>
using namespace std;

long long N, K, A[1 << 18];

int main() {
    cin >> N >> K;
    for (int i = 1; i <= N; i++) cin >> A[i];
    for (int i = K + 1; i <= N; i++) {
        if (A[i - K] < A[i]) cout << "Yes" << endl;
        else cout << "No" << endl;
    }
    return 0;
}
```

（解説は次ページへ続きます）

別解：累積和を用いた解法

$N = 200000$ の場合、評点が最大で $10^{1800000}$ と非常に大きくなってしまうので、long long 型などの整数型では表せません。そこで、対数（log）を用いることを考えます。

$$C_i = \log A_1 + \log A_2 + \log A_3 + ... + \log A_i$$

とし、i 学期の評点を L_i、$M_i = \log L_i$ とするとき、

$$M_i = C_i - C_{i-K}$$

となります。そして、$M_{i-1} < M_i$ であれば $i-1$ 学期の評点より i 学期の評点の方が高いことがわかります。なお、C_i の値については、$i = 1, 2, 3, ..., N$ の順に $C_i = C_{i-1} + \log A_i$ とすると、$O(N)$ 回の演算で求められます。

誤差が心配かもしれませんが、与えられる制約の下で i 学期の評点と $i-1$ 学期の評点が違う場合は、評点の差が 1.0000000001 倍以上あること、つまり $|M_i - M_{i-1}| \geq 4.5 \times 10^{-10}$ が証明できます。（最も大きい場合で、$1000000000 \div 999999999$ 倍です。）

したがって、$M_{i-1} = M_i$ の場合に間違って Yes と判定しないように適切に誤差に対処すれば AC できます。例えば、$M_{i-1} + 10^{-10} < M_i$ を満たす時のみ Yes と判定すると上手くいきます。以下 C++ での実装例です。

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

long long N, K, A[1 << 18];
double B[1 << 18], M[1 << 18];

int main() {
    cin >> N >> K;
    for (int i = 1; i <= N; i++) cin >> A[i];
    for (int i = 1; i <= N; i++) B[i] = B[i - 1] + log10(A[i]);
    for (int i = K; i <= N; i++) M[i] = B[i] - B[i - K];

    for (int i = K + 1; i <= N; i++) {
        if (M[i - 1] + 1e-10 < M[i]) cout << "Yes" << endl;
        else cout << "No" << endl;
    }
    return 0;
}
```

D. Road to Millionare

この問題は、大きく分けて以下の 2 つの解法があります。

- 貪欲法（greedy）を用いた解法
- 動的計画法（DP）を用いた解法

貪欲法を用いた解法

まず、どのような売買の仕方をした場合でも、以下の例の通り、**「すべての株は買ってから 1 日で売る」**とみなすことができます。

具体例

1 日目に 10 株買い、3 日目に 10 株売り、4 日目に 8 株買い、5 日目に 3 株売り、6 日目に 5 株売りたい場合は、以下と同じになります。

- 1 日目に 10 株買い、2 日目に 10 株売る。
- 2 日目に 10 株買い、3 日目に 10 株売る。
- 3 日目に 0 株買い、4 日目に 0 株売る。
- 4 日目に 8 株買い、5 日目に 8 株売る。
- 5 日目に 5 株買い、6 日目に 5 株売る。

どのような売買方法が最適か？

どのような株も、1 日で売ってよいことがわかりました。

そこで、i 日目に買い $i+1$ 日目に売る株の数 T は、以下の通りにすることが最適となります。

$A_i \geq A_{i+1}$ の場合

そのような状況において、株を買っても損するだけです。

具体的には、i 日目に T 株買った場合、$T \times (A_i - A_{i+1})$ 円損します。

よって、i 日目には 1 株も買わないのが最適です。

$A_i < A_{i+1}$ の場合

そのような状況において、株を買えば買うだけ得します。

具体的には、i 日目に T 株買った場合、$T \times (A_{i+1} - A_i)$ 円得します。

よって、i 日目には全財産をつぎ込み、できるだけ多くの株を買うのが最適です。

（解説は次ページへ続きます）

サンプルコード

以下のようにシミュレーション的に実装できます。（C++ での実装例です。）

```cpp
#include <iostream>
using namespace std;
long long N, A[87], CurrentMoney = 1000;
int main() {
    cin >> N;
    for (int i = 1; i <= N; i++) cin >> A[i];
    for (int i = 1; i <= N - 1; i++) {
        long long Stocks = 0;
        if (A[i] < A[i + 1]) Stocks = CurrentMoney / A[i];
        CurrentMoney += (A[i + 1] - A[i]) * Stocks;
    }
    cout << CurrentMoney << endl;
}
```

DP を用いた解法

紙面の都合上、詳しい証明は省略しますが、以下のように株を売買するのが最適です。

- 株を買う時は、必ず全財産をつぎ込む。
- 株を売る時は、必ず今持っている株をすべて売る。

つまり、中途半端に株を買ったり売ったりするような方法は、最適ではないということです。
そこで、以下のような DP テーブル（配列）を持つことを考えます。

- $dp[x]$： x 日目に、株の売却が終了した時点の所持金の最大値。

そこで、各 i について $dp[i]$ を求めることを考えますが、「$i-1$ 日目から何もしない場合」「j 日目に全財産を使って株を買い、i 日目に株を全部売る場合」のみ考えれば良いので、各 i について考えるべき遷移は高々 N 通りになります。合計計算量は $O(N^2)$ となり、余裕を持って間に合います。

DP を用いた解法のサンプルコード

- https://atcoder.jp/contests/m-solutions2020/submissions/15082710

注意

本問題の制約下では、答えが必ず $2^{40} \times 1000 < 1.1 \times 10^{15}$ 以下になることが証明できます。なぜそうなるか、考えてみましょう。

E. M's Solution

プログラミングで問題を解くときに、

- そもそもどうやったら解けるのか？
- どうやったら計算時間を改良できるのか？

を考えるときは、いろいろな切り口で考え始めて、考察を深めながら解法を思いつくでしょう。

この問題では、3 つの全く異なる出発点から考え始めても、3 つすべてで別々の解法にたどり着きます。本解説では、これをすべて紹介してみたいと思います。

解法 1：使う道の候補を絞ってみる

いきなり 2 次元バージョンの問題を解くのは難しいです。ですから、まずは 1 次元の場合に落として考えてみましょう。ここで、次のような問題を考えます。

> N 個の家が数直線上の座標 $x_1, x_2, x_3, ..., x_N$ にある。あなたは店を K $(K \leq N)$ 個建てて、すべての家についての「最も近い店までの距離」の合計を最小にするとき、これはいくつになるか？

この問題では、最適解は 「すべての店を $x_1, x_2, ..., x_N$ のどれかに立てること」 になります。特殊な例として、$K = 1$ の場合は、$x_1, x_2, ..., x_N$ を小さい順に並び替えた時の真ん中の値になることは有名です (知らない人は、ABC 102-C 「Linear Approximation」 を解いてみましょう)

ですから、店を立てる位置の候補は最大で N 個しかありませんので、高々 2^N 通りの店の立て方を全探索すると解けます。距離の合計は $O(NK)$ で求められるので、計算時間 $O(2^N \times NK)$ で解けます。

元の問題に戻ってみても、同じようなアイデアを使えます。実は、建設する鉄道路線の候補は、

- 南北方向に N 個（直線 $x = X_1, x = X_2, ..., x = X_N$）
- 東西方向に N 個（直線 $y = Y_1, y = Y_2, ..., y = Y_N$）

の合計 $2N$ 個となります。

これで、鉄道路線の建設方法は高々 4^N 通りに絞れることが分かりました。距離の合計は $O(NK)$ で計算できるので、この問題は $O(4^N \times N^2)$ で解くことができました。$N = 10$ 程度なら高速に計算できますが、$N = 15$ だと実行に 30 分程度の時間がかかってしまいます。

（解説は次ページへ続きます）

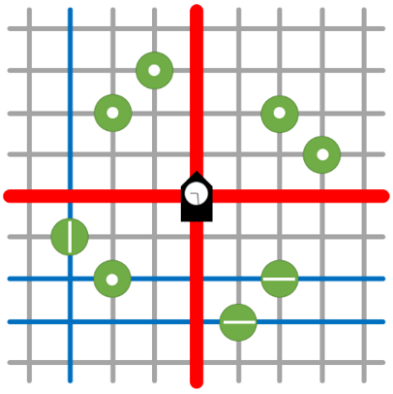
★さらに建設方法の候補を絞る

例えば、直線 $x = X_1$ と直線 $y = Y_1$ を両方選んでも無駄です。なぜなら、直線 $x = X_1$ 上に鉄道路線が建設されたら、集落 1 の住民の歩行距離は 0 になりますから、それ以降「集落 1 は消えたも同然」と考えることができるからです。（注：$Y_1 = Y_2$ の場合、$x = X_1$ と $y = Y_1$ を両方選ぶのは無駄ですが、$x = X_1$ と $y = Y_2$ を両方選ぶのは無駄ではありません）

したがって、各集落 i について、鉄道路線の建て方は次の 3 通りになります。

- 直線 $x = X_i$ 上に鉄道路線を建設する（このような鉄道路線を「｜」の路線とする）
- 直線 $y = Y_i$ 上に鉄道路線を建設する（このような鉄道路線を「−」の路線とする）
- 直線 $x = X_i$ 上、直線 $y = Y_i$ 上のいずれにも鉄道路線を建設しない

これで、鉄道路線の建て方の候補を 3^N 通りに絞ることができました。それぞれの場合について、歩行距離の合計を計算するのに $O(N^2)$ かけると、全体の計算時間が $O(3^N \times N^2)$ になってしまうので、$N = 15$ だと例えば C++ で適切に実装しても $4 - 10$ 秒ほど実行にかかり、TLE します。



★計算量を削減する

適切に前処理を行うことで、歩行距離合計の計算が $O(N)$ でできるようになります。例えば、

- 「｜」の路線しか建てない 2^N 通りのパターンについて、各集落における歩行距離
- 「−」の路線しか建てない 2^N 通りのパターンについて、各集落における歩行距離

を前計算することを考えます。（前計算には $O(2^N \times N^2)$ の計算量が必要ですが、小さいです）

各集落における歩行距離は、「｜」の路線のみ考えた場合の歩行距離、「−」の路線のみ考えた場合の歩行距離のうち小さい方になるので、歩行距離の合計は $O(N)$ で計算できます。

これでこの問題は $O(3^N \times N)$ で解けました！ $N = 15$ のケースで、適切に実装すれば C++ だと 0.3 秒程度、Python でも 1.7 秒程度で答えを求めることができます。

★サンプルコード (C++)

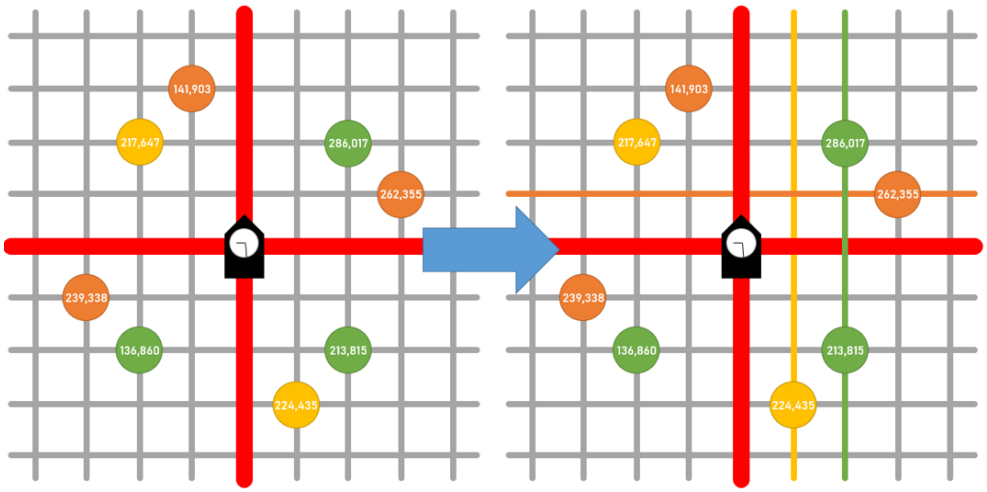
- https://atcoder.jp/contests/m-solutions2020/submissions/15122924

解法 2：各集落がどの鉄道路線を使うか決め打ちする

恐らく、こちらの解法を選ぶ人の方が多いでしょう。

南北大通り、東西大通りの他に K 本の道路を付け加え、それぞれ番号 $1, 2, ..., K+2$ を付けることを考えます。（便宜上、番号 $K+1$ を南北大通り、$K+2$ を東西大通りとします）

ここで、各集落がどの番号の道路を ”使う” のか割り当ててみましょう。この割り当て方は N^{K+2} 通りありますが、割り当てさえ決まれば鉄道路線 $1, 2, ..., K$ それぞれについて最適な位置が求まります。下図は、同じ色に同じ鉄道路線を割り当て、新たに付け加える 3 本の路線の位置を求める例です。例えばオレンジの鉄道路線の位置を決めるときは、オレンジの集落のことしか考えなくてよく、黄色・緑についても同じです。



ところで、集落の選び方 S は 2^N 通りあります。（S は選ぶ集落の集合です）

このそれぞれについて、S に含まれる集落が 1 つの鉄道路線を使う時の歩行距離の最小値 $cost[S]$ を前もって計算しておくことを考えます。それぞれの S について $cost[S]$ は $O(N^2)$ で求められるので、前処理にかかる計算時間は $O(2^N \times N^2)$ となり、十分高速です。

しかし、N^{K+2} 通りある割り当て方を全探索することはできません。ここで、動的計画法 (DP) を用いて高速化することを考えます。ここでは、最初に番号 $K+1, K+2$ の道路を使う集落を割り当て、残った集落を番号 $1, 2, ..., K$ の順に割り当てることを考えます。

$dp[S][i]$ ＝「番号 i まで割り当て終わって、現在すでに割り当てられた集落が S のときの、現時点での歩行距離の合計の最小値」 とします。All を 「すべての集落を割り当てた状態」 とすると、$dp[All][0], dp[All][1], \cdots, dp[All][N]$ がこの問題の $K = 0, 1, ..., N$ での答えになります。

ここで $dp[S][i]$ の値は、番号 i を割り当てる集落の選び方 T すべてに対しての $dp[S-T][i-1]+cost[T]$ の最小値で計算できます。S に含まれる集落の個数を $|S|$ として、T は $2^{|S|}$ 通りあります。このすべてを効率的に列挙すれば、計算時間 $O(3^N \times N)$ で DP を終わらせることができます！

S や T などの 「集落の選び方」 は、2 進数で 「0 から $2^N - 1$ までの整数」 として表すことができます。例えば、8 つの集落のうち $4, 6, 7$ 番目が選ばれる場合は $2^3 + 2^5 + 2^6 = 104$、のような感じです。これを使うと、この DP は 2 次元配列で実装できます。このような、2 進数を使って “選び方” で DP するテクニックを 「ビット DP」 といいます。

なお、2 進数 x で表せる選び方の「部分集合」i は、以下のような繰り返し処理で列挙できます。

```
for (int i = x; i >= 0; --i) {
    i &= x; // i を (i AND x) に変える
    // ここに処理を書く
}
```

このアルゴリズムを適切に実装すると、$N = 15$ のケースで C++ で 0.3 秒、Python でも 1 秒程度で実行を終わらせることができます。

★サンプルコード (C++)

- https://atcoder.jp/contests/m-solutions2020/submissions/15122985

解法 3 :「焼きなまし法」を使う

焼きなまし法は、2020/6/28 に行われた 「Introduction to Heuristics Contest」 など、できるだけ良い答えを制限時間内に見つける ”マラソン型課題” でよく使われます。でも、実はこの問題でも使うことができ、上手に実装すれば全てのテストケースで最適解が出せてしまいます！焼きなまし法の説明は、Introduction to Heuristics Contest 解説 の 9〜11 ページを見てください。

この問題では、道路が 「直線 $x = X_1, X_2, ..., X_N$」 または 「直線 $y = Y_1, Y_2, ..., Y_N$」 の $2N$ 本の候補のうちから選ぶので、最大 30 本と、焼きなまし法で最適解を得るのに十分な小ささです。

焼きなまし法を 1 回だけ実行した場合、最適解を得るのに失敗することもあります。しかし、ランダムに $5 - 10$ 回実行を繰り返し、その中の最善の解を出力することで、全ケースで AC できます。

★サンプルコード (C++)

- https://atcoder.jp/contests/m-solutions2020/submissions/15132853

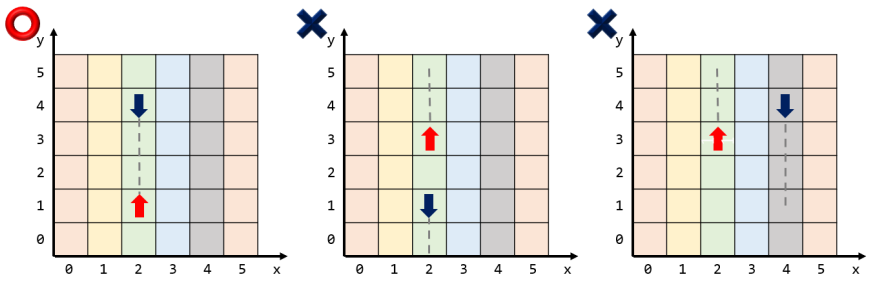
F. Air Safety

まず、「2 つの飛行機 A と B が衝突する条件」について考えてみましょう。

Case 1. 逆向きで動いている場合

ここでは、片方が U、もう片方が D の向きに移動している場合に絞って考えます。
（他の場合も、図を 90 度回転することで同じことがいえます。）

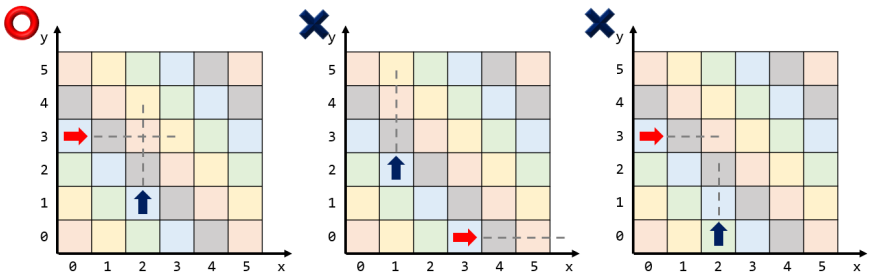
- x 座標が同じで、U の向きに動く飛行機の方が y 座標小さい場合に限り、衝突します。
- 下図左側のようなパターンです。



Case 2. 90 度違う向きで動いている場合

ここでは、片方が U、もう片方が R の向きに移動している場合に絞って考えます。
（他の場合も、図を適切な角度で回転することで同じことが言えます）

- $x_i + y_i$ の値が同じで、U の向きに動く飛行機の方が y 座標小さい場合に限り、衝突します。
- 下図左側のようなパターンです。



Case 3. 同じ向きで動いている場合

どの 2 つの飛行機も同じ速度であるため、衝突することは絶対にありません。
したがって、考える必要性がありません。

次に、衝突する最も早い時刻を求めることについて考えてみましょう。

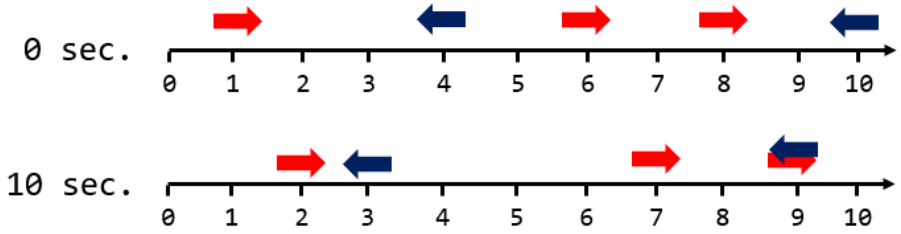
最小衝突時刻を求める - 逆向きの場合

ここでは、片方が U、もう片方が D の向きに移動している場合に絞って考えます。まずは、x 座標が 0 であるようなものの最小衝突時刻を求めます。

$x_i = 0$ であるような、U または D の向きに移動する飛行機すべてを y 座標小さい順に並べるとき、(y 座標, 向き) $= (R_1, S_1), (R_2, S_2), ..., (R_A, S_A)$ となったとします。そのとき、以下の値が最小衝突時刻になります。

- $S_i =$ 'U'、$S_{i+1} =$ 'D' となるような i における、$(R_{i+1} - R_i) \times 5$ の最小値。

例えば、$(R, S) =$ (1,'U'), (4,'D'), (6,'U'), (8,'U'), (10,'D') の場合は、以下の図のように、時刻 $2 \times 5 = 10$ にはじめて衝突します。



このように、$O(A \log A)$ で最小衝突時刻を求めることができました。x 座標が 0 でない場合についても、最初に x 座標ごとの飛行機の番号を vector 型などで持っておくと、合計して $O(N \log N)$ で計算できます。

最小衝突時刻を求める - 90 度違う向きに動いている場合

ここでは、片方が U、もう片方が R の向きに移動している場合に絞って考えます。基本的には、前述した逆向きの場合と似たような考え方が使えます。

さて、$0 \leq B \leq 400000$ それぞれについて、以下の値を定義することを考えます。

> $x_i + y_i = B$ であるような、U または R の向きに移動する飛行機すべてを y 座標小さい順に並べるとき、(y 座標, 向き) $= (R_1, S_1), (R_2, S_2), ..., (R_A, S_A)$ となったとする。そのとき、$S_i =$ 'U'、$S_{i+1} =$ 'R' となるような i における、$(R_{i+1} - R_i) \times 10$ の最小値を $f(B)$ とする。

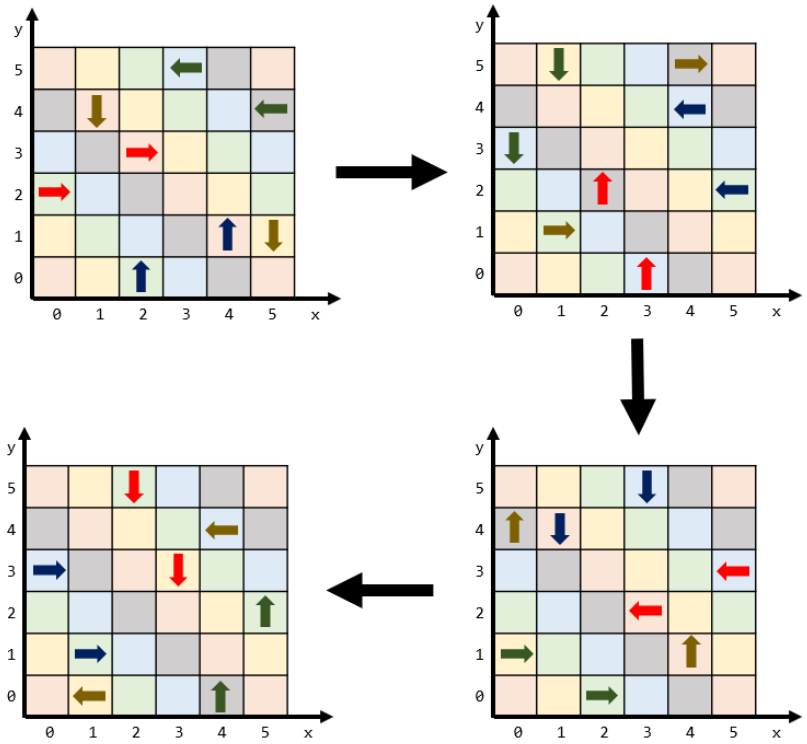
そのとき、$f(0), f(1), ..., f(400000)$ の最小値が、最小衝突時刻となります。各 $f(B)$ は $O(A \log A)$ で求められるので、合計して $O(N \log N)$ で最小衝突時刻が計算できます。

回転によって実装量を減らす

全ての組について衝突時刻の判定をするためには、以下の 6 通りについて計算する必要があります。

1. U の飛行機と D の飛行機の衝突時刻の最小値
2. L の飛行機と R の飛行機の衝突時刻の最小値
3. U の飛行機と R の飛行機の衝突時刻の最小値
4. U の飛行機と L の飛行機の衝突時刻の最小値
5. D の飛行機と R の飛行機の衝突時刻の最小値
6. D の飛行機と L の飛行機の衝突時刻の最小値

しかし、6 つの計算全て実装したら、コード長が 150 行を超え、プログラムをバグらせる可能性が高くなってしまいます。そこで、図を 90 度回転させることを 4 回繰り返すと、1. と 3. の計算しか実装する必要がなくなります。例えば、D の飛行機と R の飛行機の衝突時刻の最小値は、90 度回転させた後に 3.（U の飛行機と R の飛行機の場合）の計算を行うと、求められます。



サンプルコード

C++ の実装例です。およそ 70 行程度で実装できます。

- https://atcoder.jp/contests/m-solutions2020/submissions/15082216