AtCoder Beginner Contest #004

解説資料

2014年　2月 16日

AtCoder株式会社　青木謙尚

目次

1. 競技プログラミングを始める前に
2. A~D問題
    - ところどころに入出力の方法があります

競技プログラミングを始める前に

競技プログラミングを始める前に

AtCoder

（競技）プログラミングを始めたいけど、

そもそもプログラムって何をするの？

競技プログラミングを始める前に

AtCoder

- （競技）プログラミングを始めたいけど、そもそもプログラムって何をするの？

- 大雑把に言うと

1. 情報を受け取って

情報 → プログラム

競技プログラミングを始める前に

AtCoder

- （競技）プログラミングを始めたいけど、そもそもプログラムって何をするの？

- 大雑把に言うと

1. 情報を受け取って

   情報 → プログラム

1. 何らかの処理を施して情報を加工する

   情報 → プログラム → 情報

競技プログラミングを始める前に

- （競技）プログラミングを始めたいけど、そもそもプログラムって何をするの？

- 大雑把に言うと

1. 情報を受け取って

   情報 → プログラム

1. 何らかの処理を施して情報を加工する

   情報 → プログラム → 情報

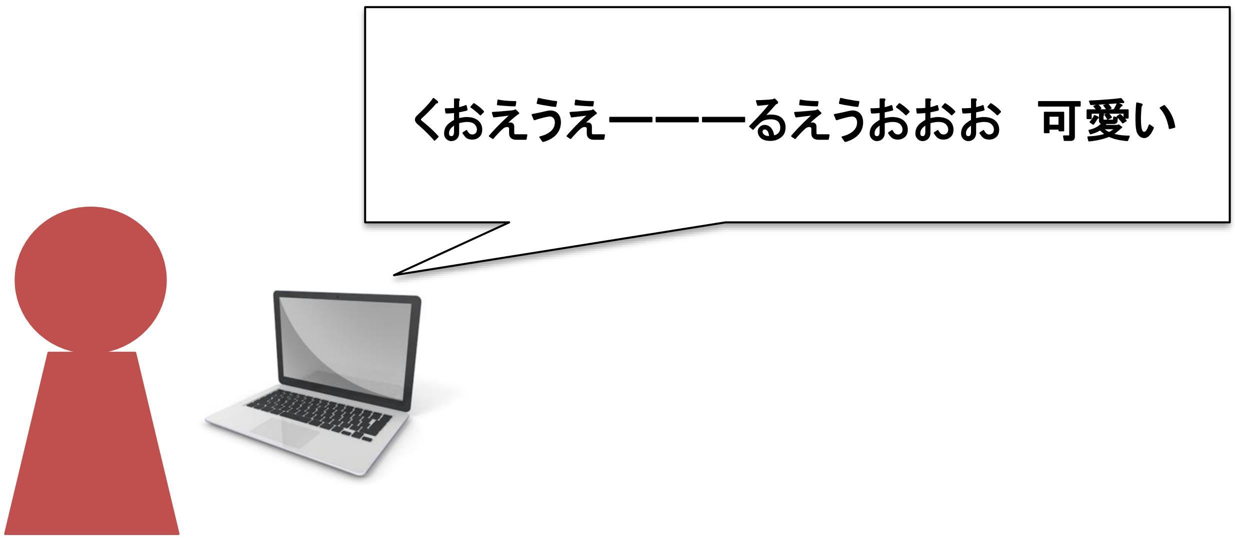
1. そしてどこかへ渡す

   情報 → プログラム → 情報 →

競技プログラミングを始める前に

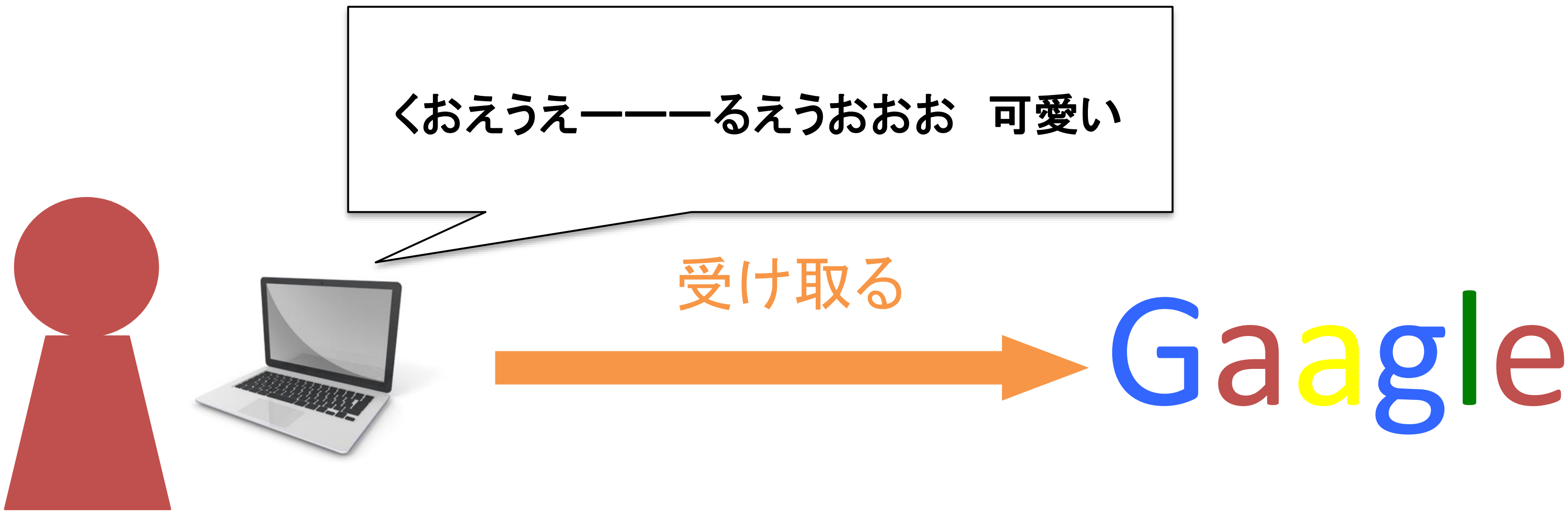
AtCoder

- 具体例
  - 検索



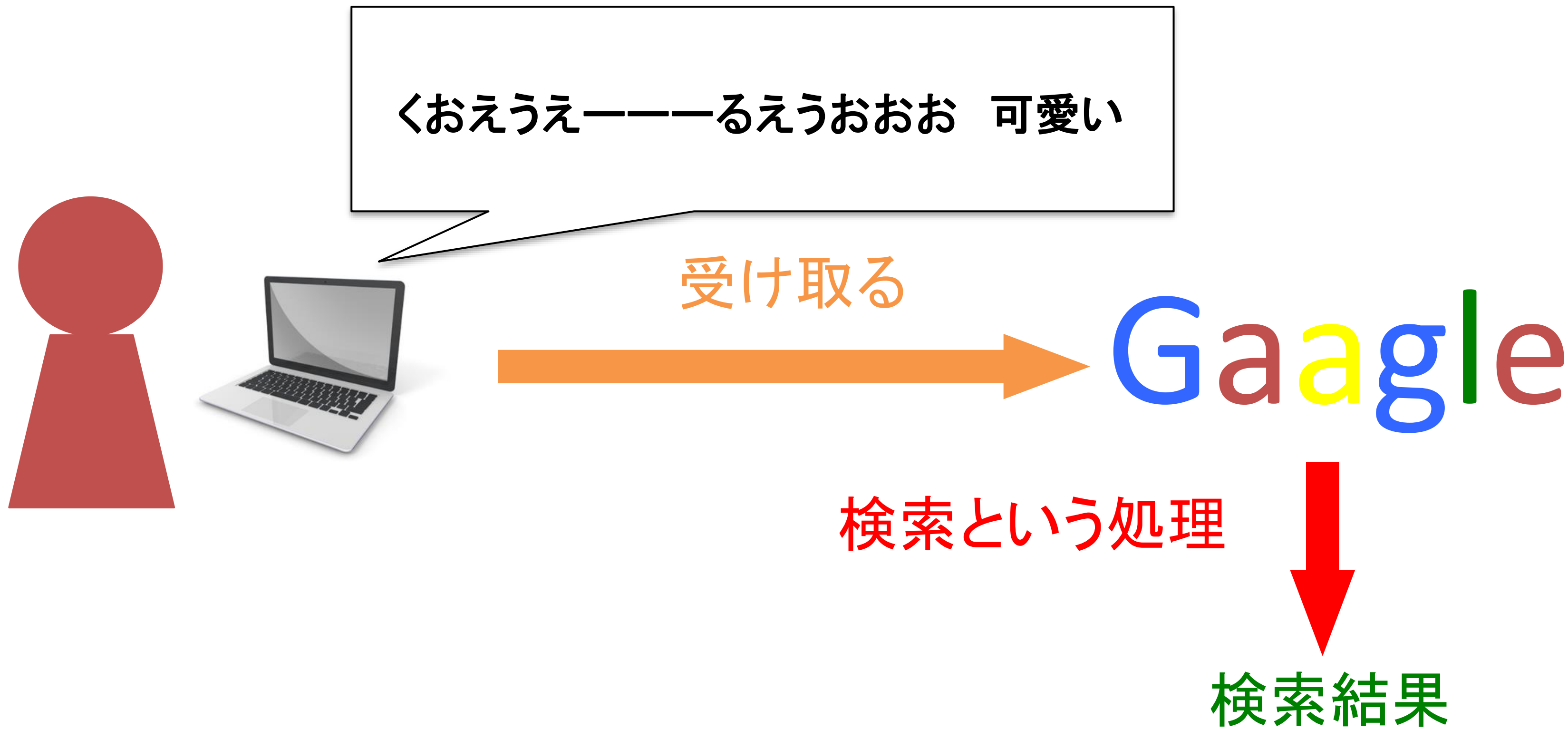
競技プログラミングを始める前に

- 具体例
  - 検索



競技プログラミングを始める前に

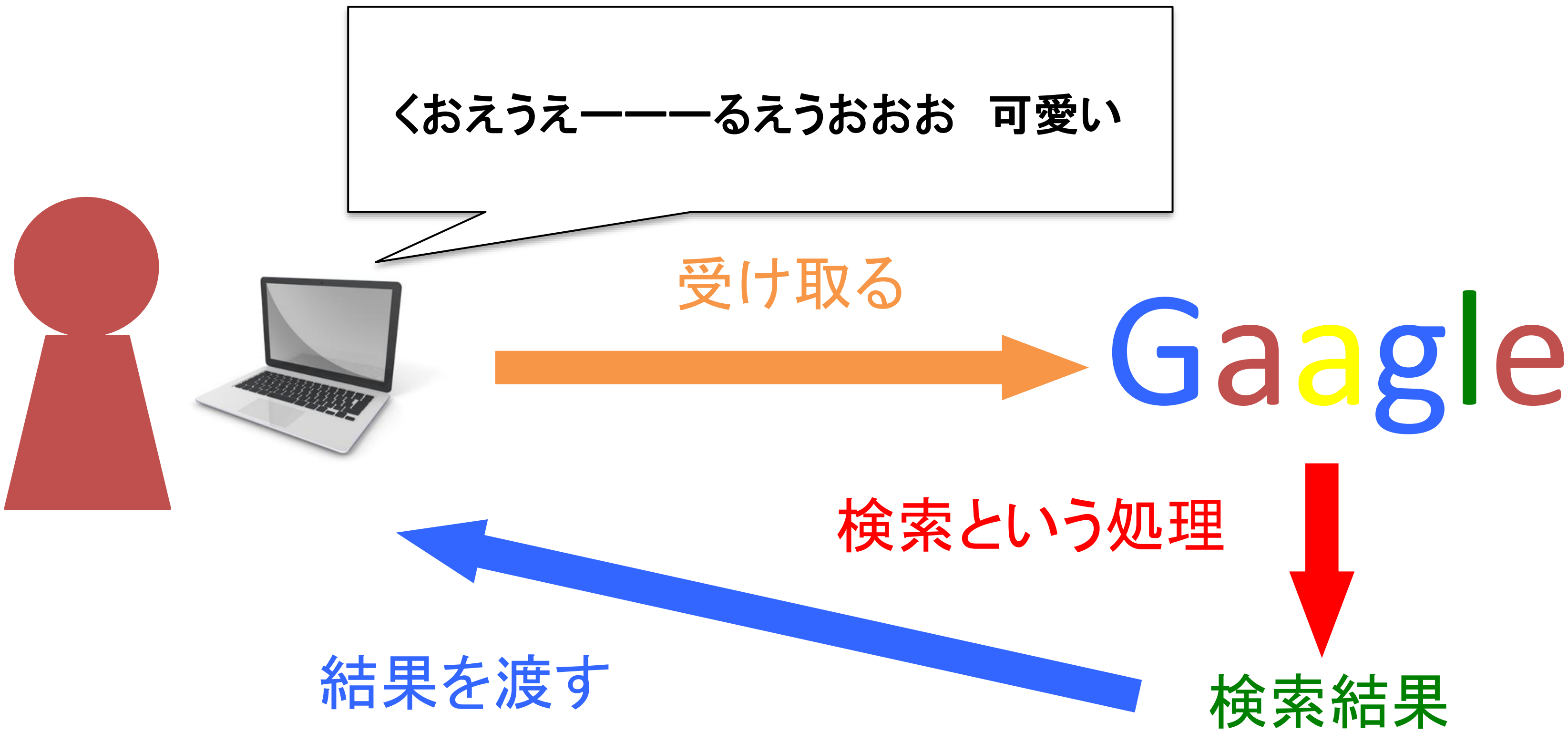
- 具体例
  - 検索



競技プログラミングを始める前に

AtCoder

- 具体例
  - 検索



競技プログラミングを始める前に

AtCoder

- 入出力ができなければ、
  - **“くおえうえーーーるえうおおお　可愛い”**
  - を受け取れないし、
  - 検索結果も渡せない

競技プログラミングを始める前に

AtCoder

- 入出力ができなければ、
    - **“くおえうえーーーるえうおおお　可愛い”**
    - を受け取れないし、
    - 検索結果も渡せない
- アルゴリズムより先にすることがある！
- 本当に本当に始めたばかりの人へ
    - まずは入出力から始めませんか？

A問題

1. 問題概要
2. 入力
3. 処理（アルゴリズム）
4. 出力

A問題　問題概要

AtCoder

1. 整数Nが与えられる。

1. 2*Nを出力せよ。

A問題　問題概要

1. 整数Nが与えられる。
   - 入力される値であるNを保存する！
2. 2*Nを出力せよ。
   - 保存したNに2をかけて出力する！

A問題　入力

- 入力の取り方は標準入出力でググってください
  - とはいえ、少しだけサンプルを載せます
  - コードの色は
    - 受け取り
    - 処理
    - 出力
  - を表したものではないので、注意して下さい
    - AtCoderへ提出したときの色です

A問題　入力

AtCoder

- C

```
#include<stdio.h>
int main(){
    int N;
    scanf("%d", &N);
    return 0;
}
```

A問題　入力

- C++

```
#include<iostream>
int main(){
    int N;
    std::cin >> N;
    return 0;
}
```

A問題　入力

AtCoder

- Java

```java
import java.util.Scanner;
public class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int N = sc.nextInt();
    }
}
```

A問題　入力

AtCoder

- C#

```
using System;
class Program{
    static void Main(string[] args){
        int N = int.Parse(Console.ReadLine());
    }
}
```

A問題　入力

- その他
  - http://practice.contest.atcoder.jp/ を参照してください
  - たいていの言語の例があります

A問題　補足（変数と型について）

- さきほどから入力を受け取ると何度も書いてますが、受け取るためには保存するための入れ物が必要です。

- プログラムの世界でも入れ物に形があります。
  - これを型といいます。

A問題　補足（変数と型について）

AtCoder

- 型の例
  - 整数型 int
    - 0, 1, 2 … 100 … などの整数
  - 文字型 char
    - a, b, c … A, B … などの1文字
- 各型で表現できる最大、最小の値は言語や処理系で変化します

A問題　補足（変数と型について）

- 変数について
  -
```
int variable = 4;
```
とすると、

4

variable

variableという入れ物に4が入ります

  - さらに
```
variable = 1;
```
とする

と

1

variable

variableに1が上書きされます

A問題　処理（アルゴリズム）

受け取ったNに

- 2をかける

A問題　処理（アルゴリズム）

受け取ったNに

- 2をかける

コードは以下だけ

```
N = N * 2;
```

A問題　出力

処理の結果を出力します

- C

```
printf(“%d¥n”, N);
```

- C++

```
cout << N << endl;
```

A問題　出力

- Java

```
System.out.println(N);
```

- C#

```
Console.WriteLine(N);
```

B問題

1. 問題概要
2. 入力
3. 処理（アルゴリズム）
4. 出力

B問題　問題概要

1. 4x4の盤面が与えられる。

2. 盤面を180度回転させて出力せよ。

B問題　問題概要

AtCoder

1. 4x4の盤面が与えられる。
   - 盤面を保存する
2. 盤面を180度回転させて出力せよ。
   - 盤面を180度回転させる
   - 出力する

AtCoder

絶望ポイントその１

- 盤面の状態ってどう保存すればいいんだ！？

絶望ポイントその２

- 180度の回転ってどうやればいいんだ！？

AtCoder

絶望ポイントその１

- 盤面の状態ってどう保存すればいいんだ！？

➢ 配列を使いましょう

絶望ポイントその２

- 180度の回転ってどうやればいいんだ！？

➢ コピー用の配列を作る

➢ 元の配列を逆からコピーする

B問題　入力

まずは盤面の保存から

- 配列って何？

まずは盤面の保存から

- 配列って何？

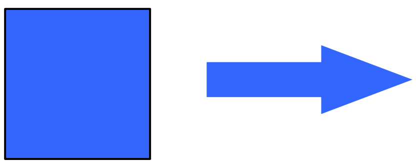
➢ **連続した箱**です

- 世界で最も偉大な箱かも

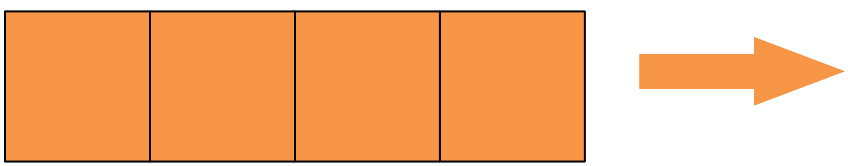
B問題　入力

まずは盤面の保存から

- 配列って何？



1つだけだと変数

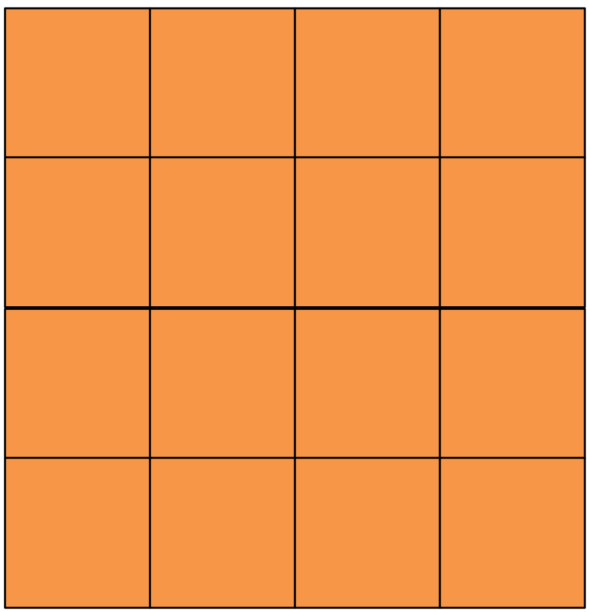


連続してたら配列

B問題　入力

まずは盤面の保存から

- どこが偉大なの？



縦に並べることができる

➢ これで盤面が作れる！

B問題　入力

AtCoder

まずは盤面の保存から

- 配列の概念はわかったけど、何をすればいいの？

B問題　入力

まずは盤面の保存から

- 配列の概念はわかったけど、何をすればいいの？

➢ **宣言**して下さい

2次元配列の宣言

- C/C++

```
char board[4][4];
```

- Java

```
char[][] board = new char[4][4];
```

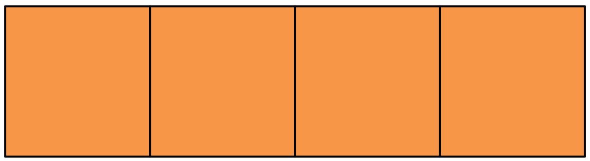
- C#

```
char[,] board = new[4, 4];
```

B問題　補足（配列のアクセス方法）

AtCoder

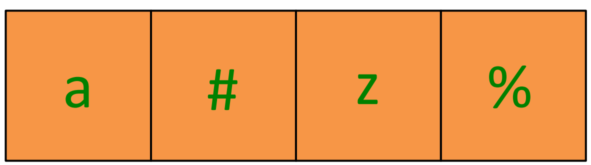
1



```
char array [4];
```

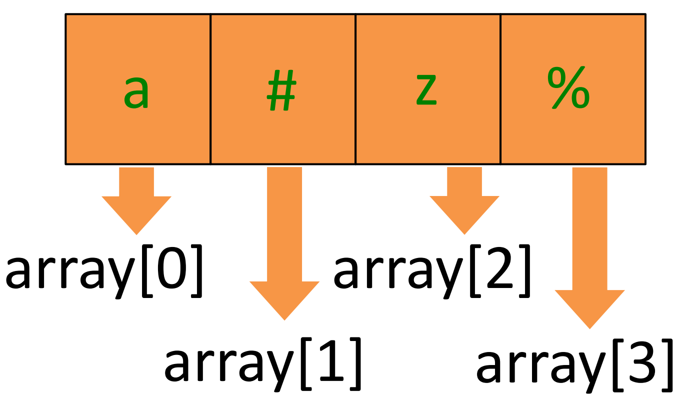
array という配列を宣言する

2



すでに文字が格納されているとする

3



array[0]のようにしてアクセスできる

0から始まることに注意！

B問題　補足（配列のアクセス方法２）

board


```
char board[2][2];
```

boardという2次元配列を宣言する

board

a	#
z	%

同様に、何らかの文字が入っているとする

board[0][0]	a	#	board[0][1]
board[1][0]	z	%	board[1][1]

board[0][0]には ’a’
board[0][1]には ’#’
board[1][0]には ’z’
board[1][1]には ’%’

180度の回転ってどうするの？

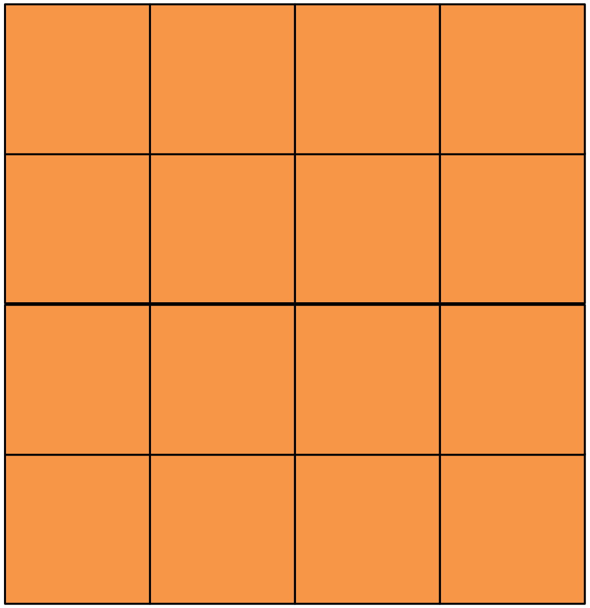
- コピー用の配列を用意します。
  - `char copy[4][4];` を宣言しておきます。
- 元の配列を逆からコピー用の配列に移します。
  - for文を使います。

B問題　処理（アルゴリズム）

AtCoder

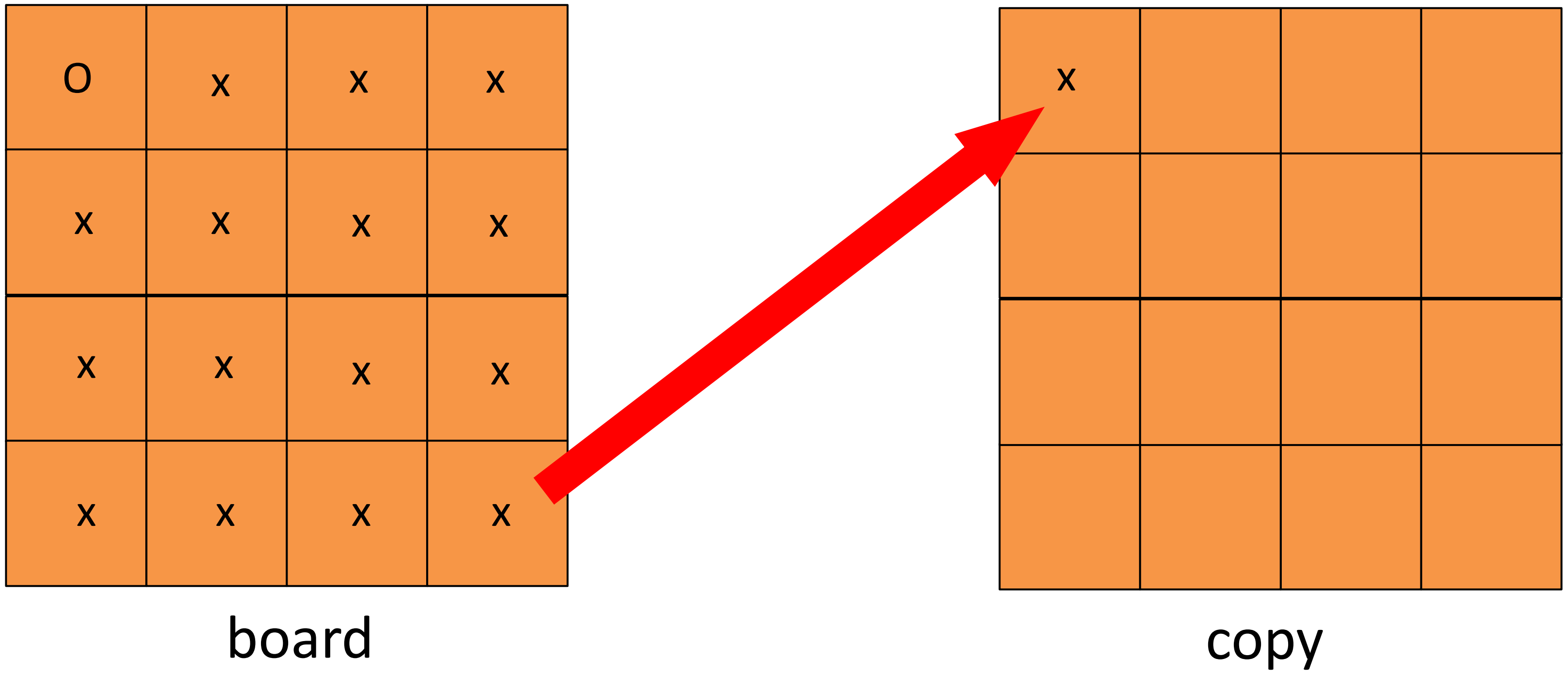
コーディングに入る前に、いまからすることを図示します。

1. コピー用の配列を宣言します。

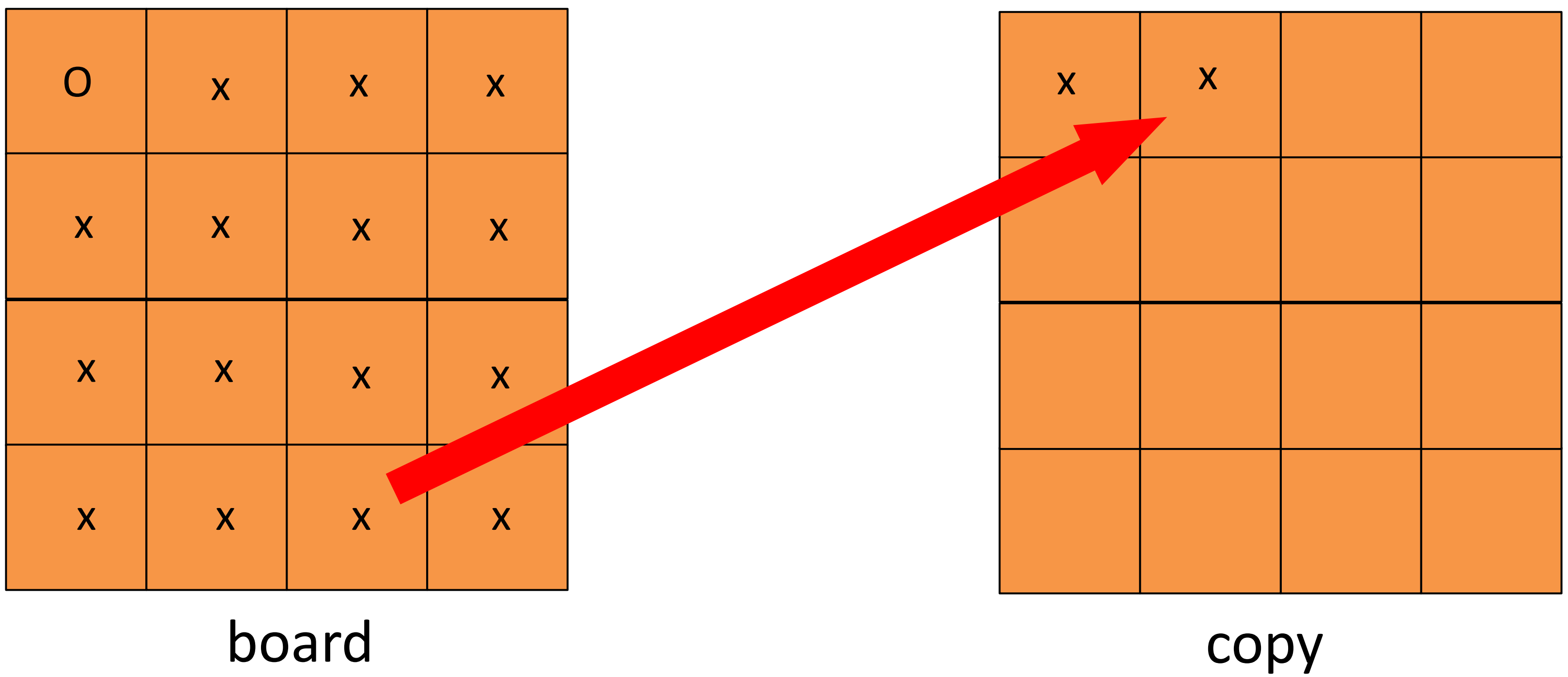


```
char copy[4][4];
```

2.元の配列を逆からコピー用の配列に移します。

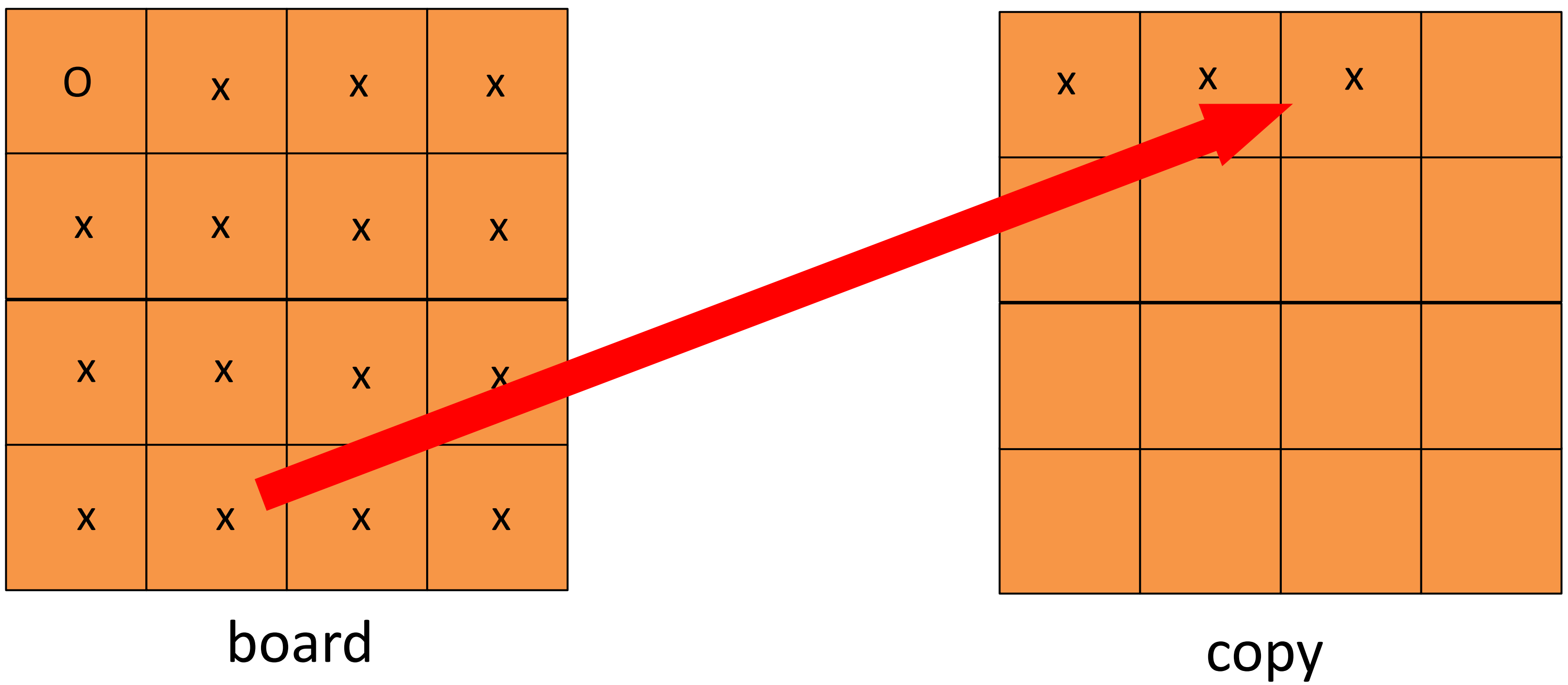


2.元の配列を逆からコピー用の配列に移します。



B問題　処理（アルゴリズム）

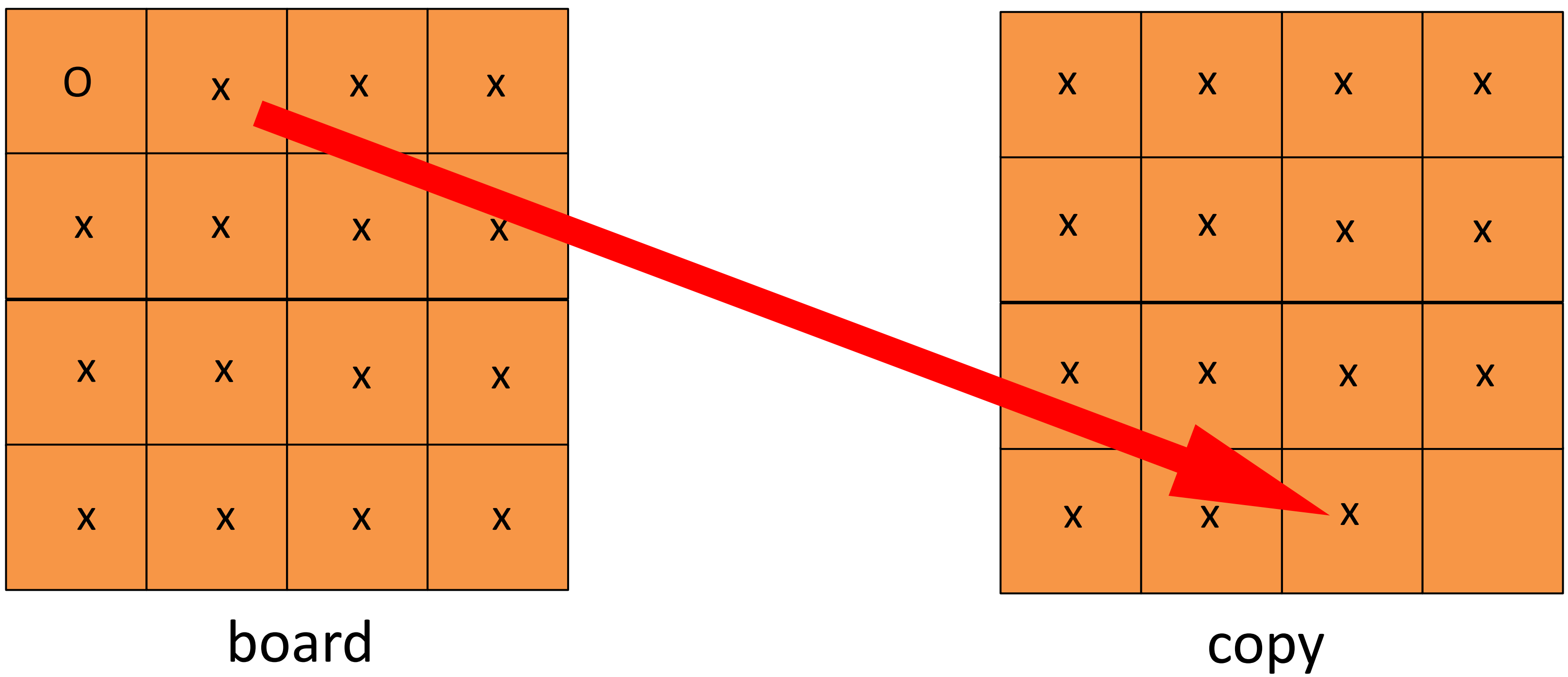
2.元の配列を逆からコピー用の配列に移します。



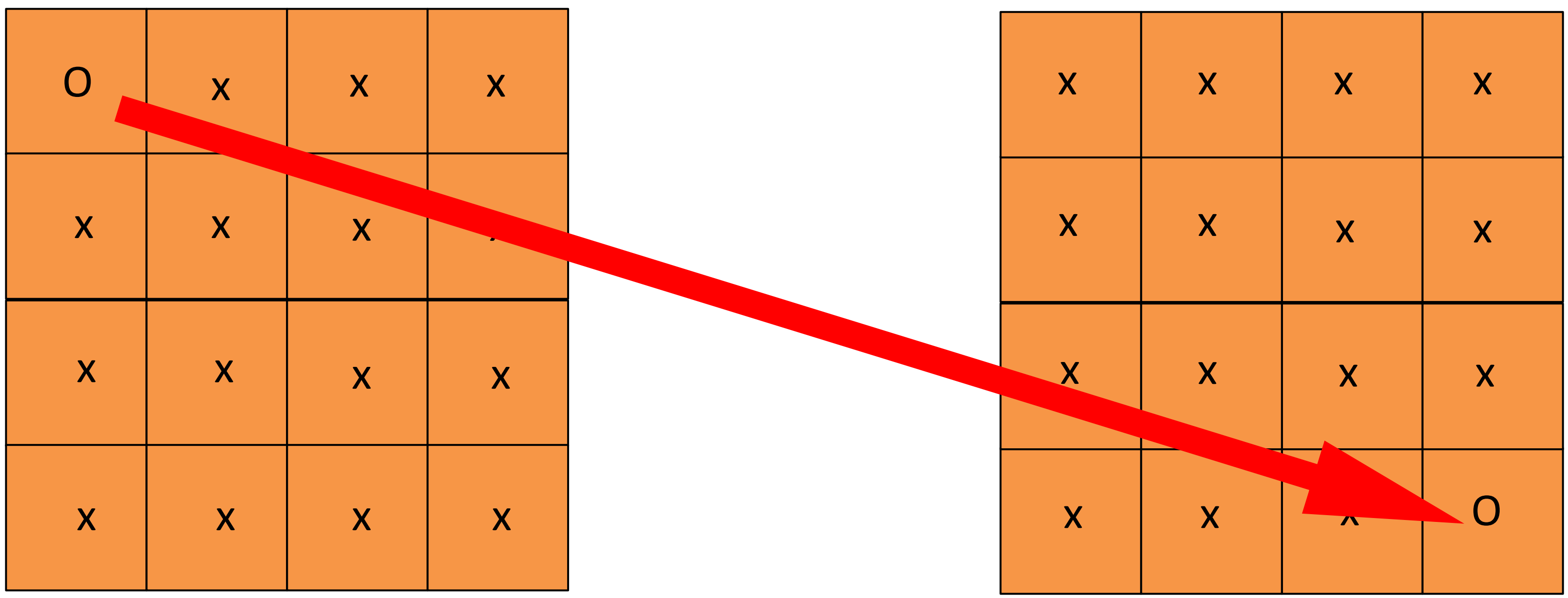
B問題　処理（アルゴリズム）

途中を省略して…

2.元の配列を逆からコピー用の配列に移します。



2.元の配列を逆からコピー用の配列に移します。



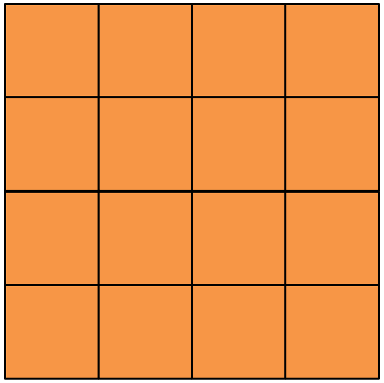
board　　　　　　　　copy

B問題　処理（アルゴリズム）

AtCoder

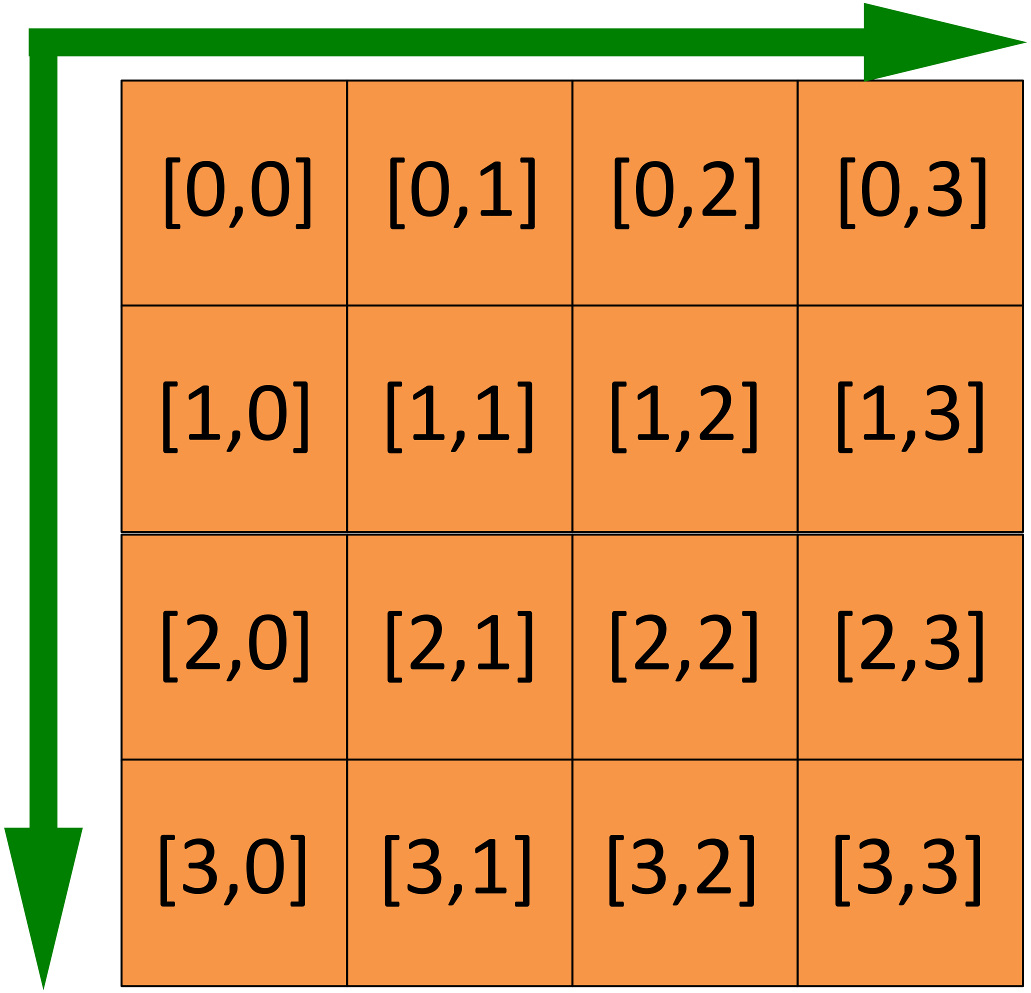
- 移せばよいことはわかったけど、実際にどうするの？
  - for文を使います。
  - for文とは繰り返し行われる処理を記述するものです。

- 復習
  - 2次元配列の添字について



AtCoder

- 2次元配列の添字について



- 注意点
  - 数学のxy座標とは異なる
  - マスは4x4だが、添字は[3,3]まで

B問題　処理（アルゴリズム）

AtCoder

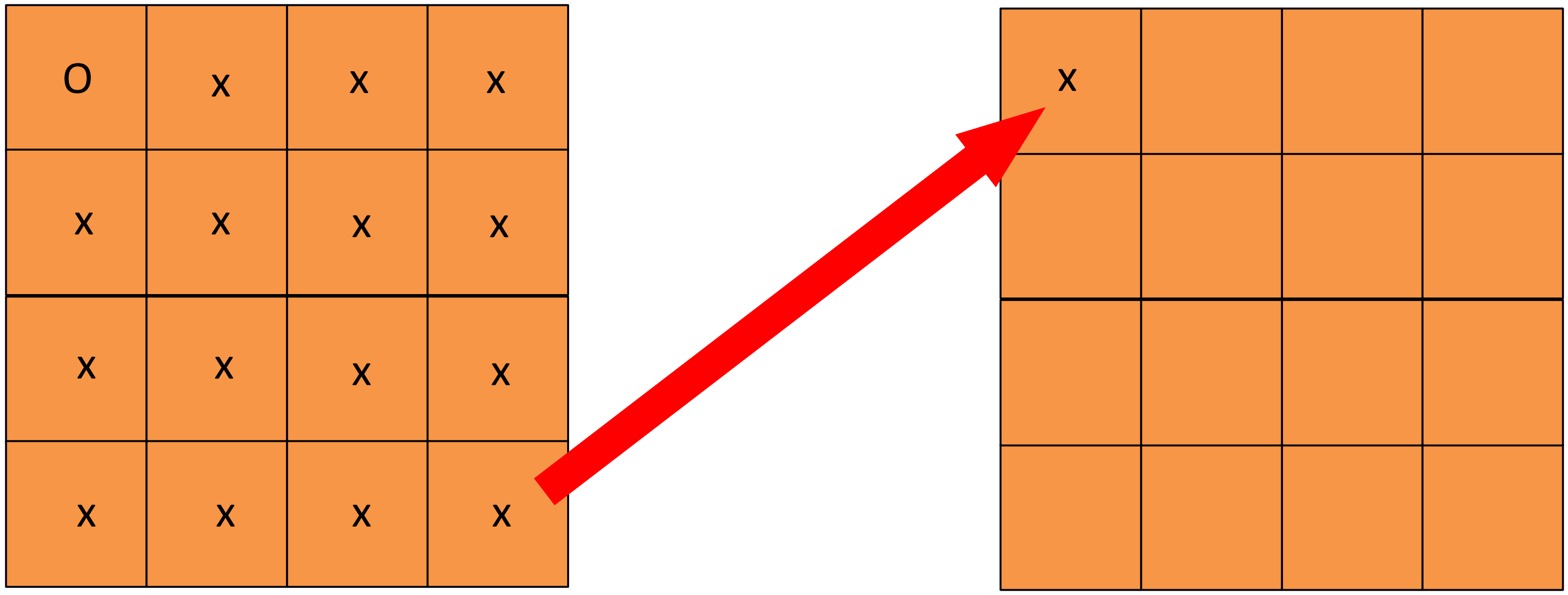
- for文のコード

```
for(int i = 0; i < 4; i++){
    for(int j = 0; j < 4; j++){
        copy[i][j] = board[3-i][3-j];
    }
}
```

AtCoder

- for文のコード(i = 0, j = 0のとき)

```
copy[0][0] = board[3-0][3-0];
```



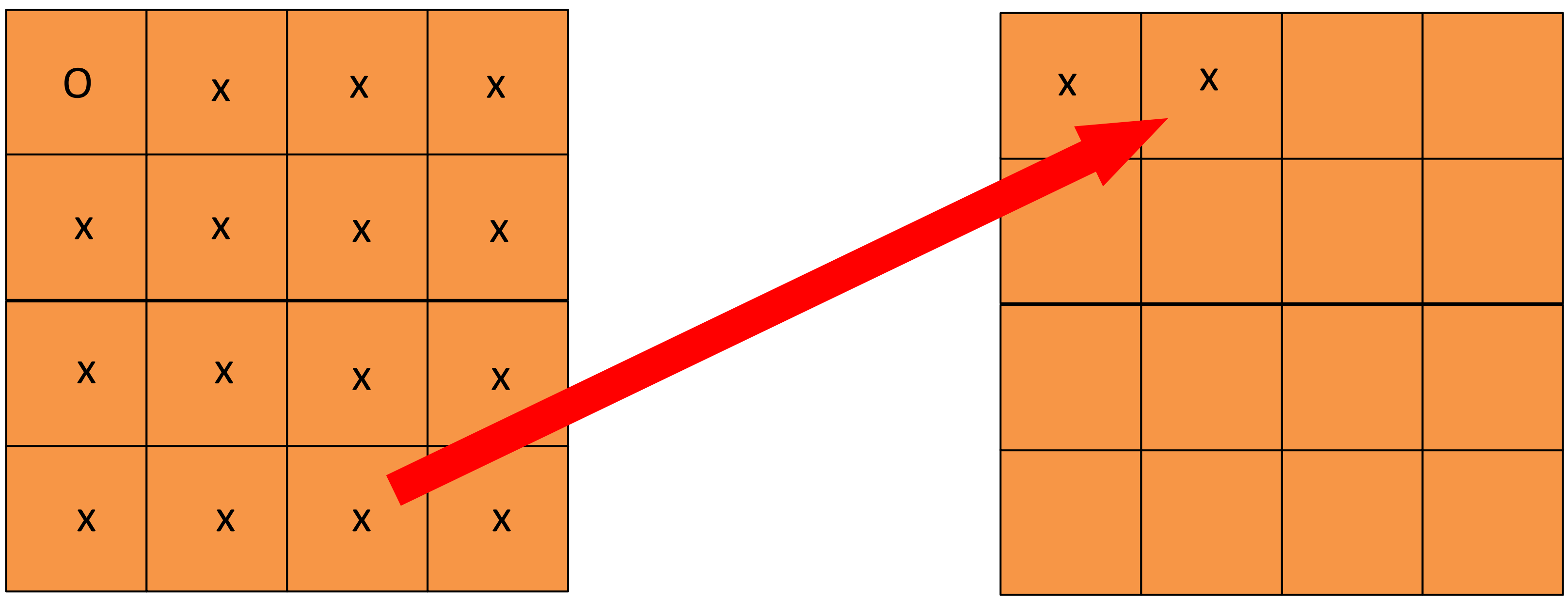
board

copy

B問題　処理（アルゴリズム）

- for文のコード(i = 0, j = 1のとき)

```
copy[0][1] = board[3-0][3-1];
```

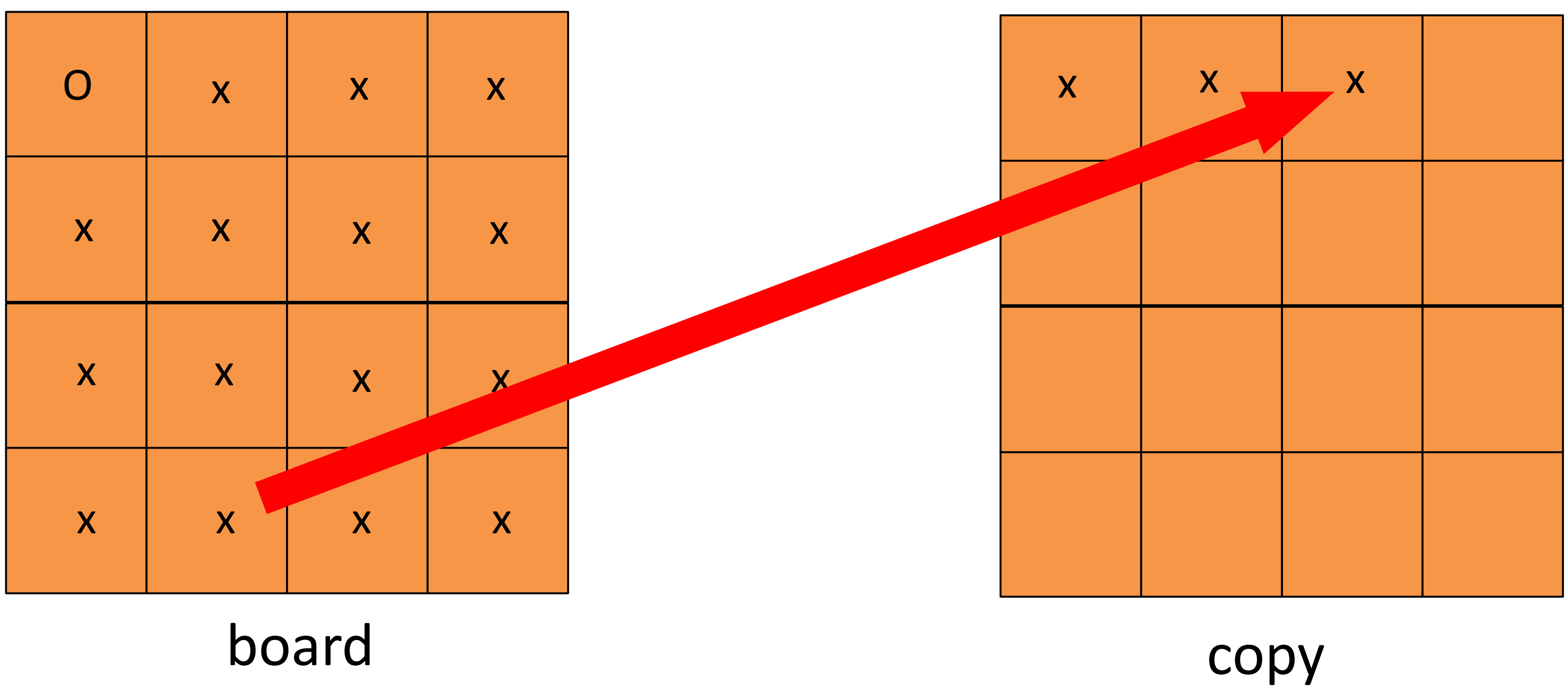


board　　　　copy

B問題　処理（アルゴリズム）

- for文のコード(i = 0, j = 2のとき)

```
copy[0][2] = board[3-0][3-2];
```



B問題　処理（アルゴリズム）

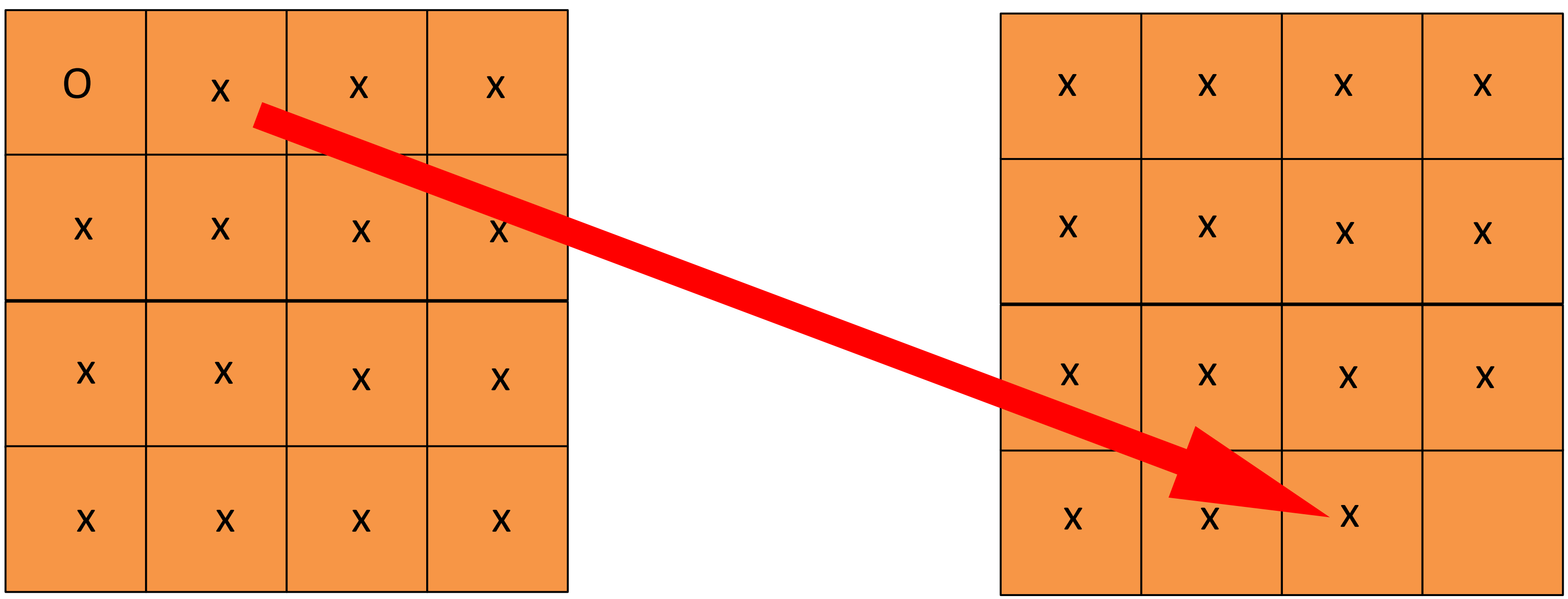
途中を省略して…

B問題　処理（アルゴリズム）

AtCoder

- for文のコード(i = 3, j = 2のとき)

```
copy[3][2] = board[3-3][3-2];
```

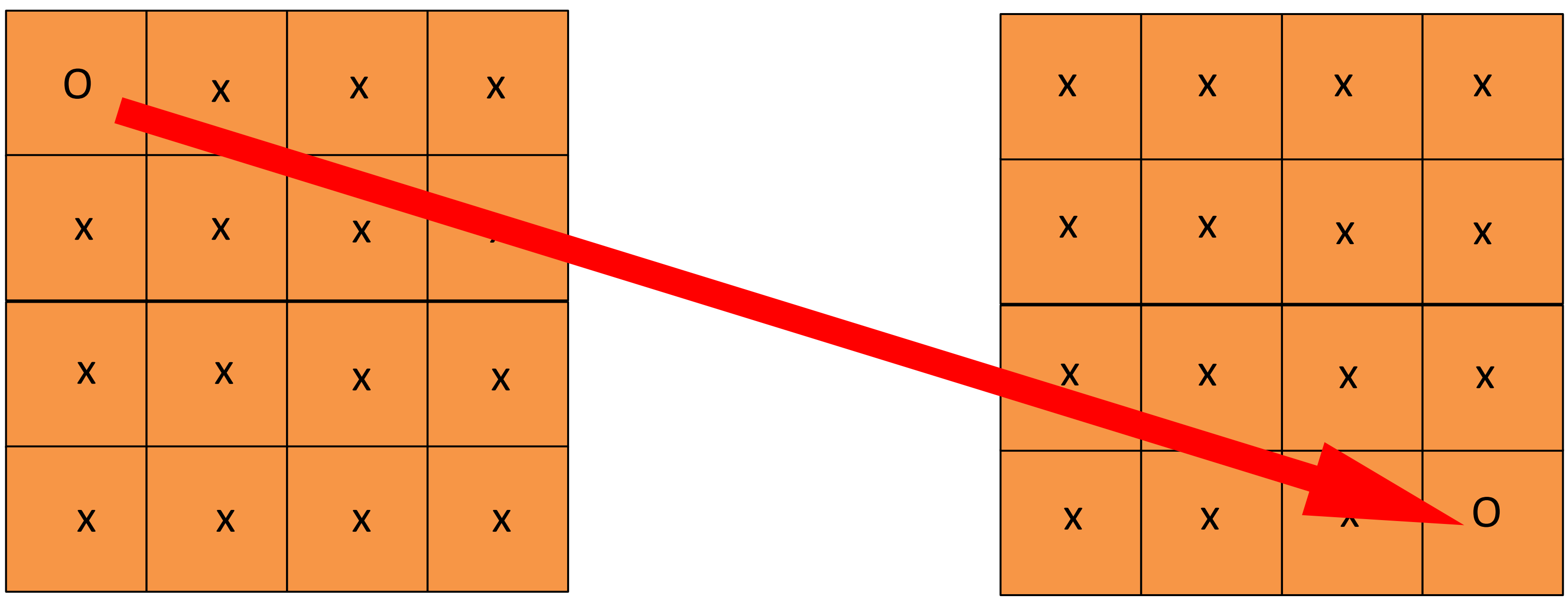


board　　　　copy

AtCoder

- for文のコード(i = 3, j = 3のとき)

```
copy[3][3] = board[3-3][3-3];
```



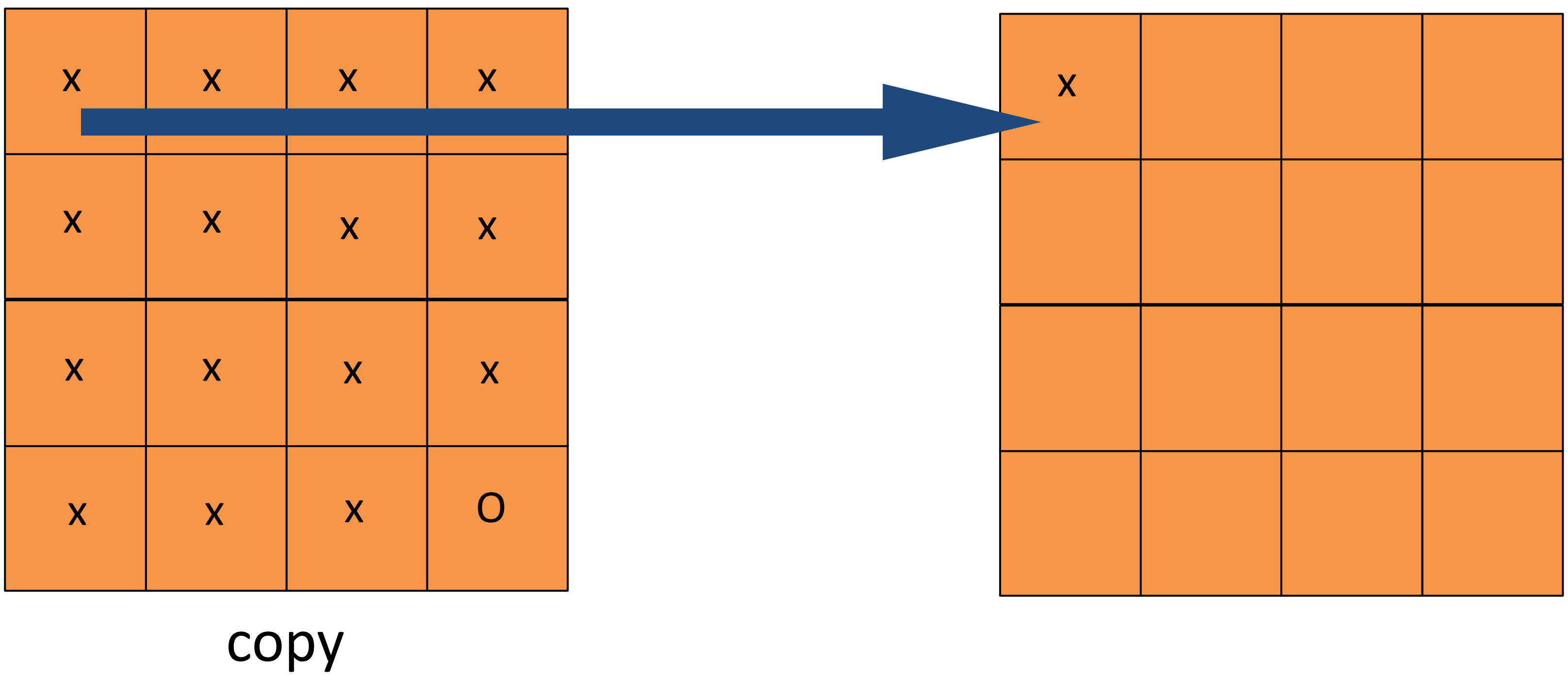
board　　　　copy

B問題　出力

AtCoder

3.180度回転させたものを順に出力するだけ

- これもforを使って書いて下さい。

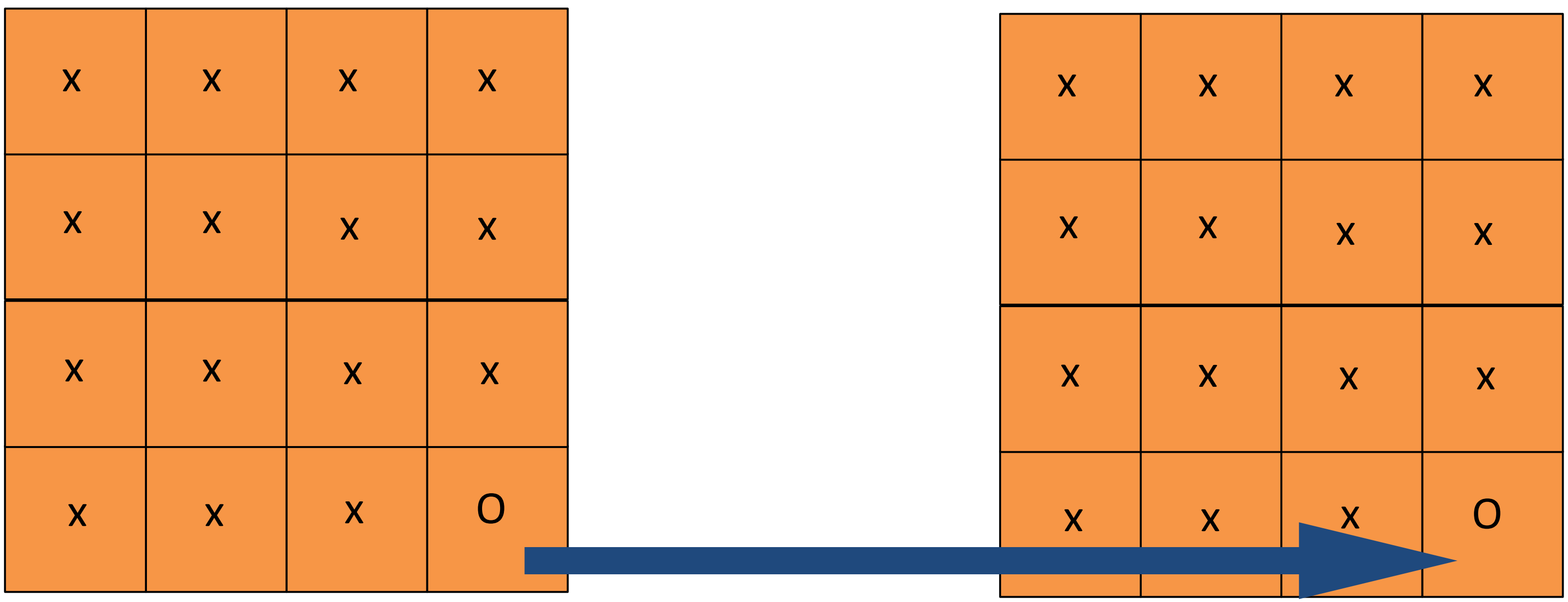


B問題　処理（アルゴリズム）

途中を省略して…

B問題　出力

3.180度回転させたものを順に出力するだけ



copy

C問題

1. 問題概要
2. 処理（アルゴリズム）

C問題　問題概要

1. 1から6までの数字が割り振られた、6枚のカードが左から順に整列している。
2. 整数N(1≦N≦10^9)が与えられる。
3. i=0からi=N-1までのN回、カードを入れ替える。
   - 左から{(i%5)+1}番目のカードと、左から{(i%5)+2}番目のカードを入れ替える。

C問題　補足（剰余について）

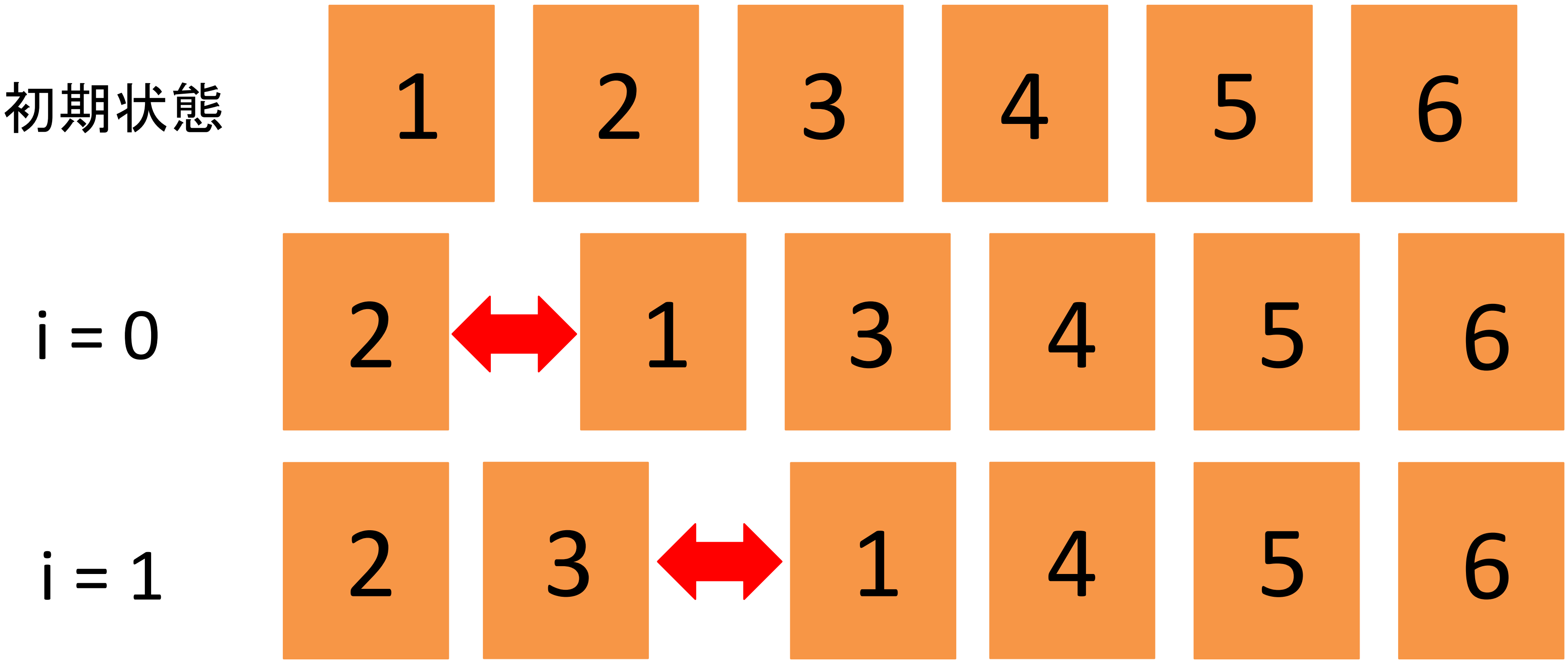
- 剰余演算子 % について
  - 要は「余り」のこと
  - 7%3 は「7を3で割った余り」を意味する
    - つまり1
  - ほとんどのプログラミング言語では%が剰余演算子として実装されている

C問題　処理（アルゴリズム）

1. この問題のポイントは、Nの値に大きな値が含まれること(最大でN=10^9まで)
2. つまり、普通に交換していくだけでは時間的な制約上、満点を得るのは厳しい(2s以内)
3. ここに気づくことができるかが、満点を得るためのポイント
   - できるだけ交換回数を減らしたい！

C問題　処理（アルゴリズム）

- とはいえ、まずは実験してみる



C問題　処理（アルゴリズム）

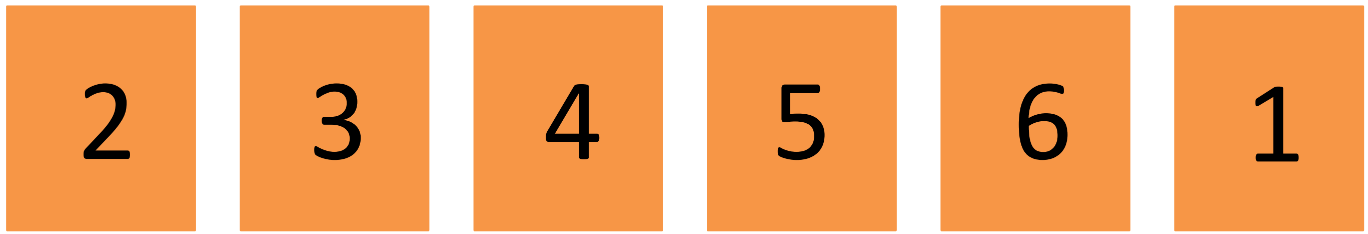
- まずは実験してみる

i = 2	2	3	4	⇔	1	5	6
i = 3	2	3	4	5	⇔	1	6
i = 4	2	3	4	5	6	⇔	1

C問題　処理（アルゴリズム）

- あれ？1が右端に移動して2~6までは整列している・・・？

i = 4



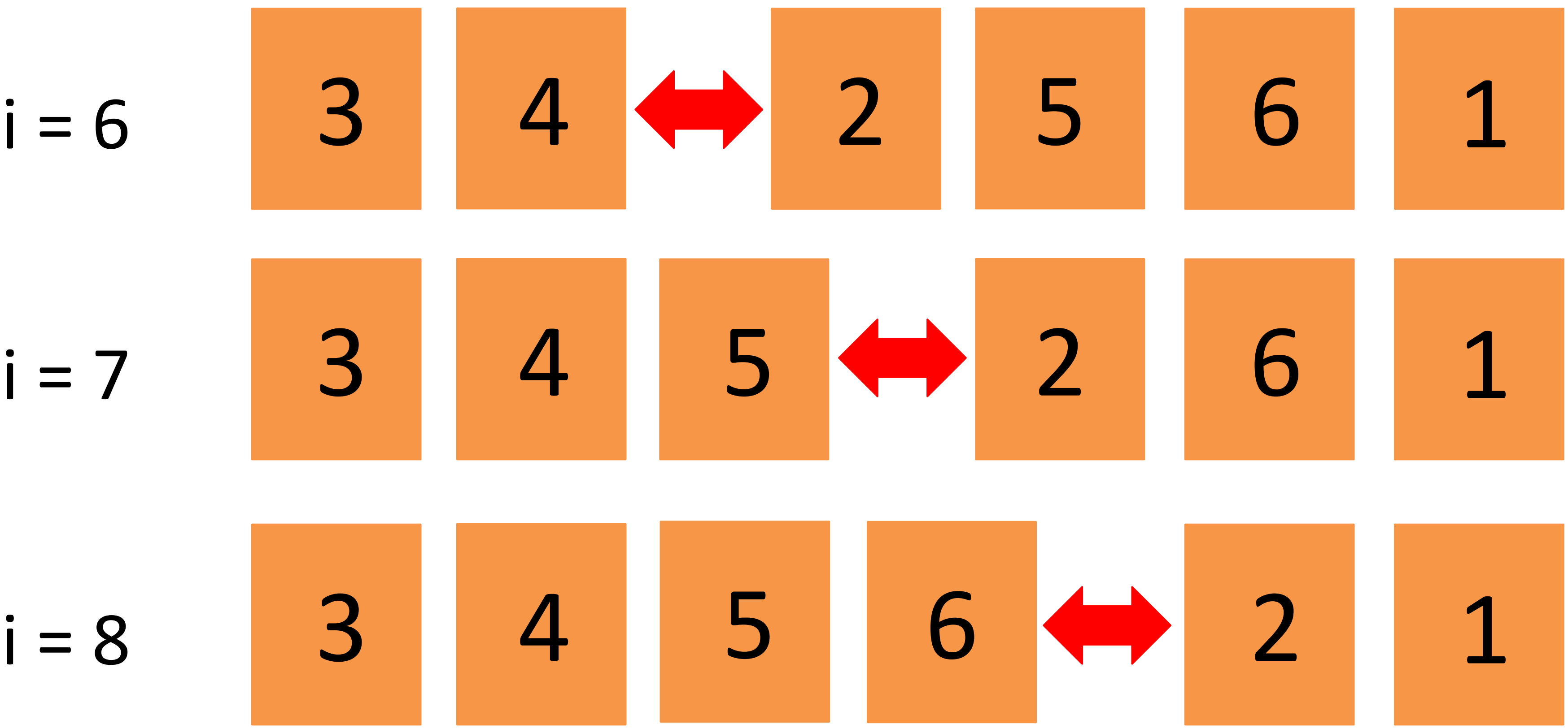
- さらに実験してみる

i = 5

C問題　処理（アルゴリズム）

AtCoder

- さらに実験してみる

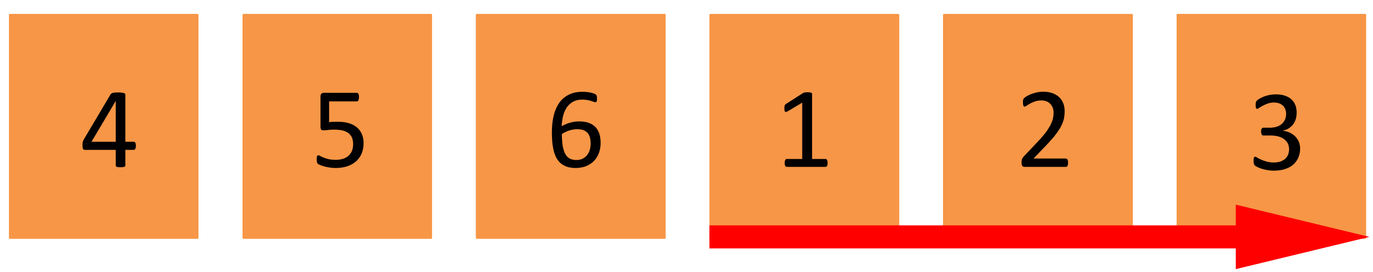


C問題　処理（アルゴリズム）

i = 9

- 先頭から3~6が整列している。で、後ろに1と2が並んでいる。
- ということは、もっと進めてみると以下のようになるのでは？

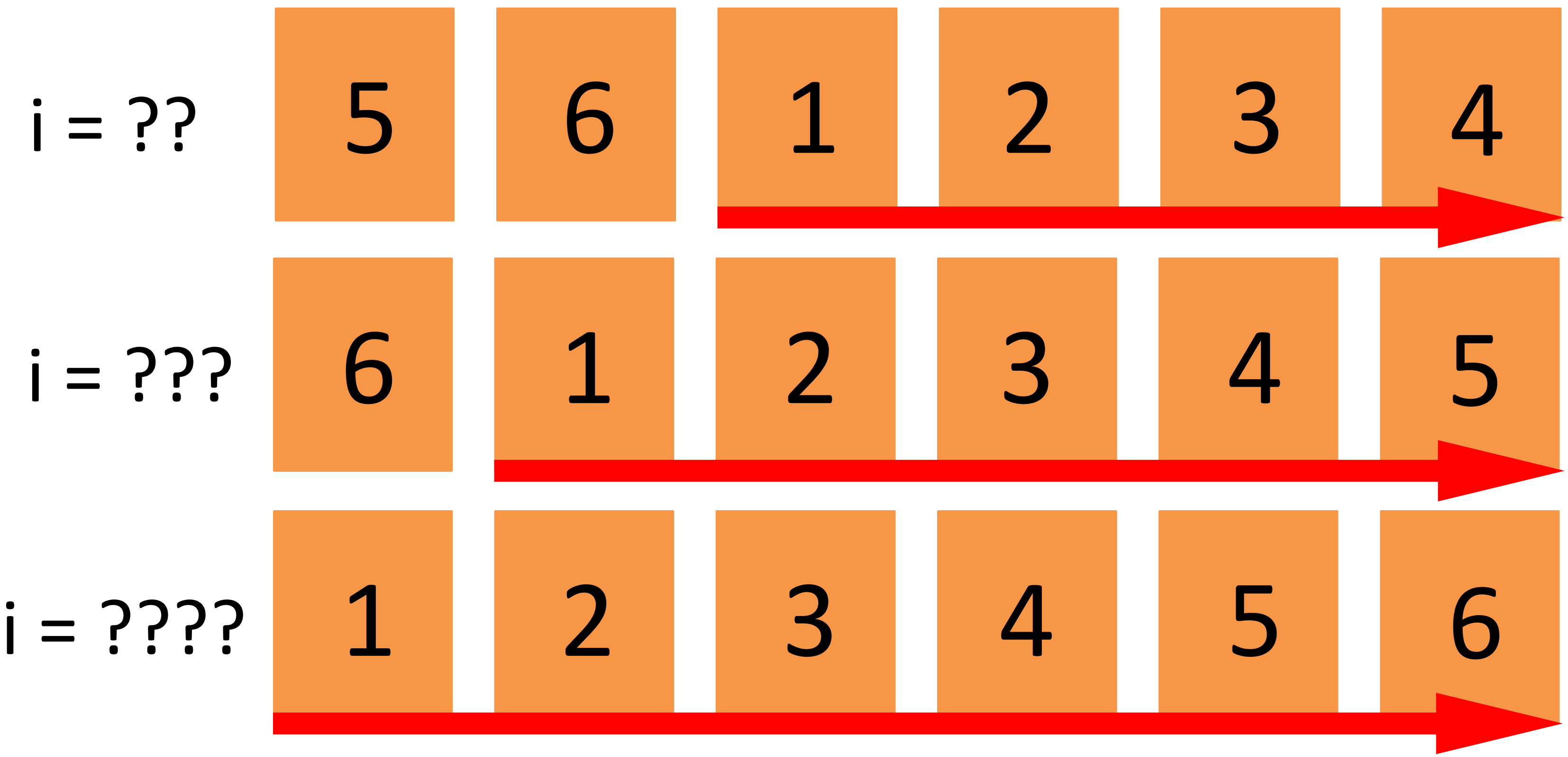
i = ?



C問題　処理（アルゴリズム）

AtCoder

- さらにさらに進めると、きっとこういった遷移が行われるはず



C問題　処理（アルゴリズム）

AtCoder

- iがいくつのときかはわからないけれど、初期状態に帰ってくる？
  - ➢ あくまでも現段階では予想である

i = 0	1	2	3	4	5	6
i = ????	1	2	3	4	5	6

C問題　処理（アルゴリズム）

- 確かめてみる！
  - 方法は、カードの遷移を出力する

i = 0　213456
i = 1　231456
i = 2　234156
:
i = 29　123456 ← 発見！
i = 30　213456

C問題　処理（アルゴリズム）

- 確かめてみる！
  - 方法は、カードの遷移を出力する

交換回数はiが
0から29までの30回

i = 0　213456
i = 1　231456
i = 2　234156
:
i = 29　123456
i = 30　213456

カードが6枚あって、入れ替え可能な場所が5通りあるから
6 x 5 = 30回交換すると初期状態に戻る

C問題　処理（アルゴリズム）

AtCoder

- 実験の結果、30回交換すると初期状態に戻ることがわかった。
  - 31回の交換は、1回の交換と同じ。
  - 32回の交換は、2回の交換と同じ。
  - 100回の交換は？
    - 100 mod 30 => 10 より、10回の交換と同じ
    - 30で割ったときの余り（剰余）がポイント
- Nを30で割ったときの余りで交換すればよい

C問題　処理（アルゴリズム）

AtCoder

- これでNの値が大きくなっても対応できる！
  - たとえば、N = 1000 のとき、実際に1000回交換するのではなく、1000 mod 30 => 10回交換すればよい
  - 30で割ったときの余りは、0~29の30通りあるが、最大でも29回ですむ。

```
N = N % 30;
```

としておけばよい

C問題　処理（アルゴリズム）

- 数値の入れ替えの仕方
- A, Bを入れ替えたい時の処理
  - C = A;
  - A = B;
  - B = C;
- のように、Cを中継してあげれば入れ替えることが出来る。
- C++などは、標準でswap関数が用意されている

AtCoder Beginner Contest #004

解説資料　続き

2014年　2月 16日

AtCoder株式会社　高橋直大

AtCoder

D問題

1. 問題概要
2. 処理（アルゴリズム）

D問題　問題概要

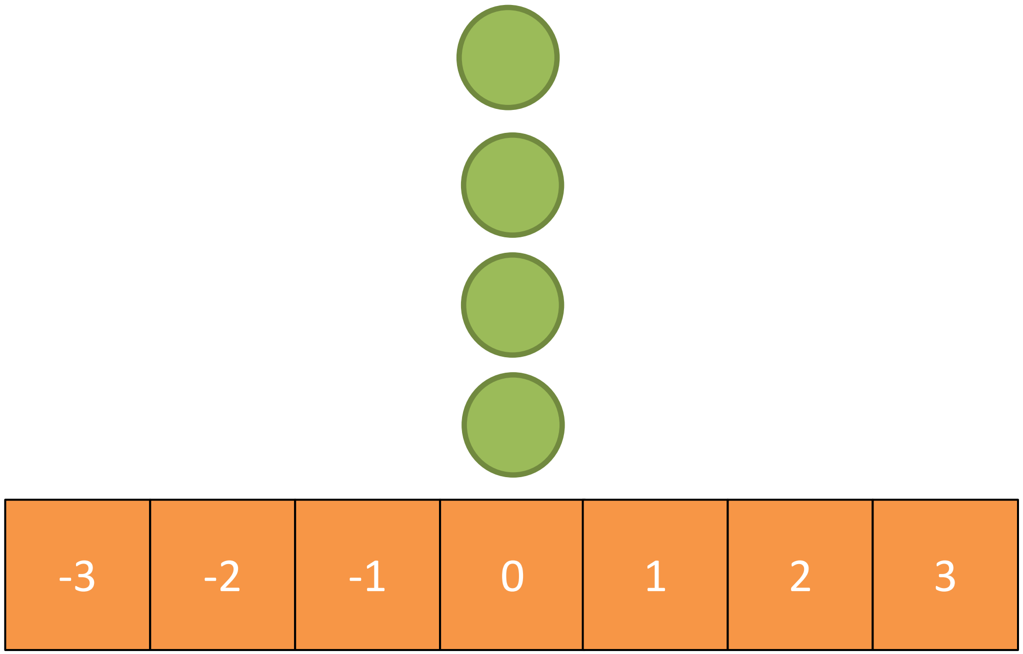
- 番号がついた箱が無限個並んでおり、左から順番に、..,-2,-1,0,1,2,..と番号がついている
- いくつかの箱には、マーブルが入っている。
  - 番号-100の箱には、赤いマーブルがR個
  - 番号0の箱には、緑のマーブルがG個
  - 番号100の箱には、青いマーブルがB個
- これらのマーブルを、１個隣の箱に移動することが出来る。
- 全ての箱に、マーブルが２個以上入っていない状態にする。

D問題　問題概要

- 部分点１
  - R, G, B ≦ 5
- 部分点２
  - R, G, B ≦ 40
- 満点
  - R, G, B ≦ 300

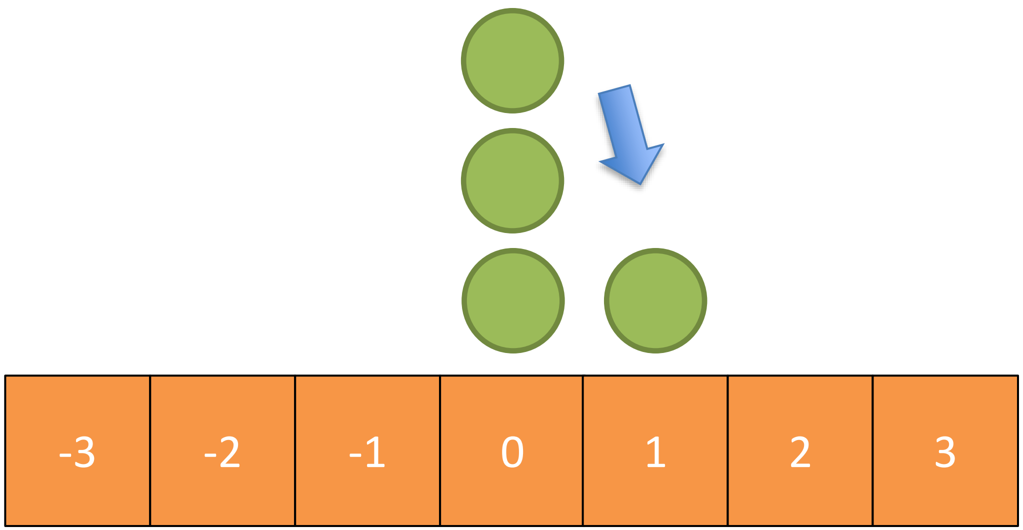
D問題　処理（アルゴリズム）

- 例えば、Gが4の時



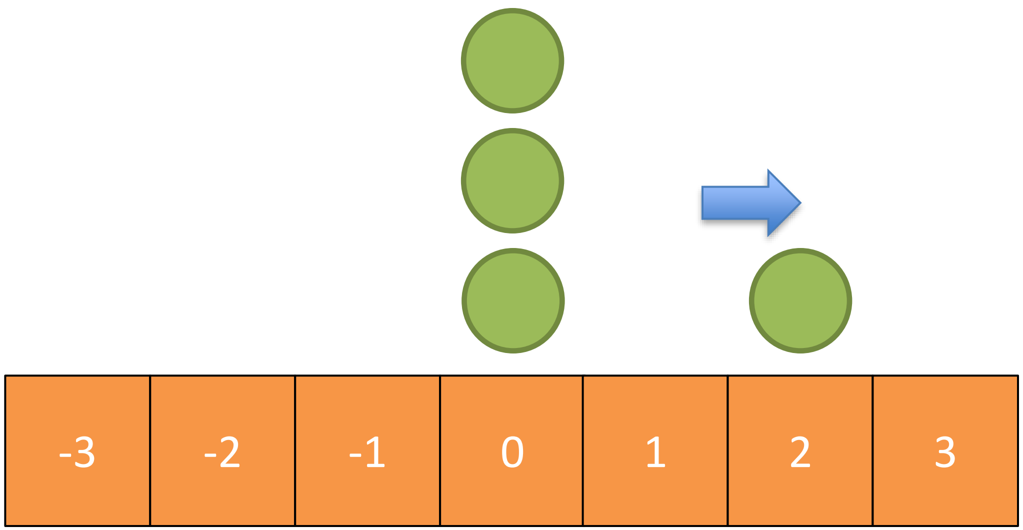
D問題　処理（アルゴリズム）

- 例えば、Gが4の時
- マーブルを右に



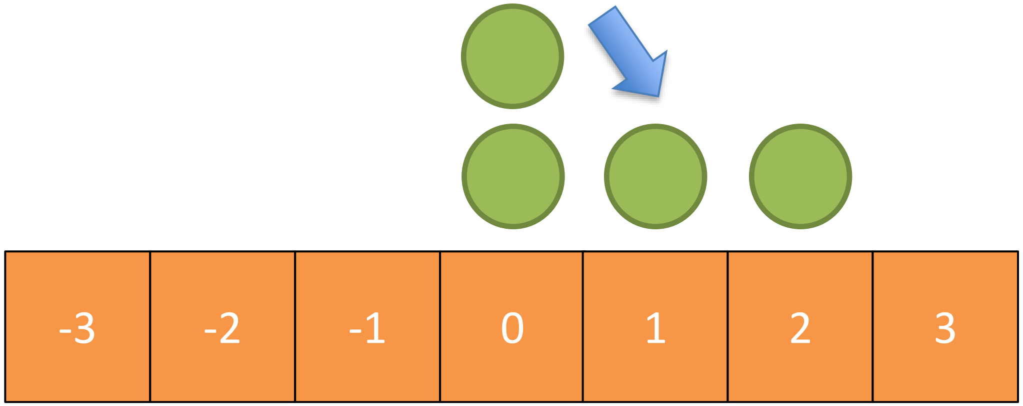
D問題　処理（アルゴリズム）

- 例えば、Gが4の時
- マーブルを右に２回移動して



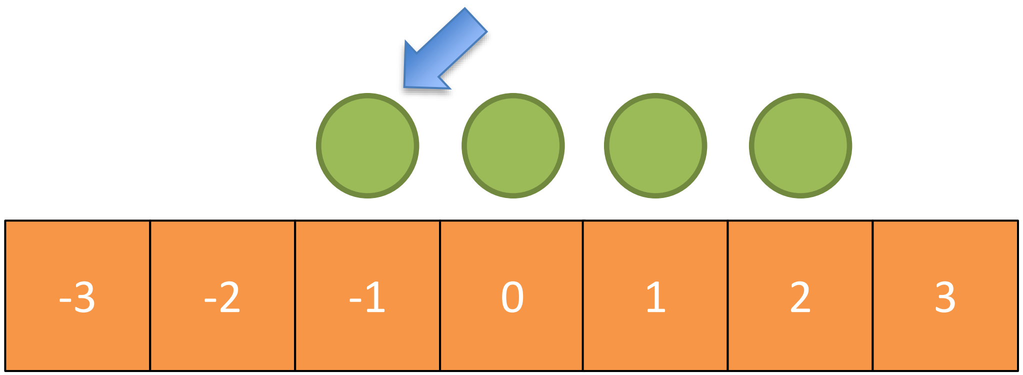
D問題　処理（アルゴリズム）

- 例えば、Gが4の時
- マーブルを右に２回移動して、さらにもう１個右に



D問題　処理（アルゴリズム）

- 例えば、Gが4の時
- マーブルを右に２回移動して、さらにもう１個右に
- 最後に左に移動しておしまい

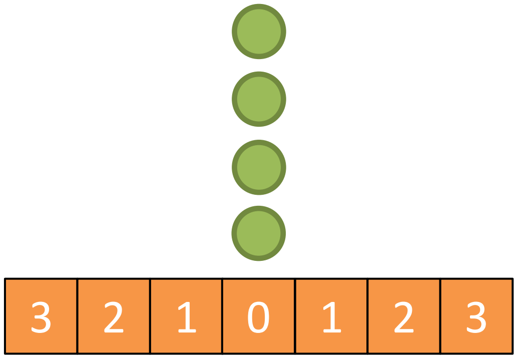


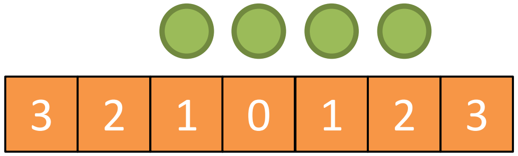
D問題　処理（アルゴリズム）

- 部分点1は、このように、R,G,Bに対して、左右にマーブルを振り分けてあげれば良い。
  - 深さ優先探索や、幅優先探索を使ってもOK。
- 部分点2も、右、左、右、左、と振り分けてあげてしまえば、R,G,Bのマーブルが40個以下、かつ、箱の数が100個離れているので、重なることもなく、解くことが出来る。
- 満点解法は、もう少し考察が必要。

D問題　処理（アルゴリズム）

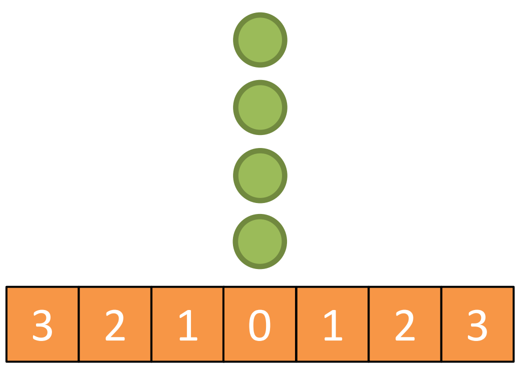
- どのマーブルを右に、どのマーブルを左に・・・と考えるのは非常に面倒！
- 最後の状態だけ考えて、そこから、その状態にするのに必要な手数を考えたい。

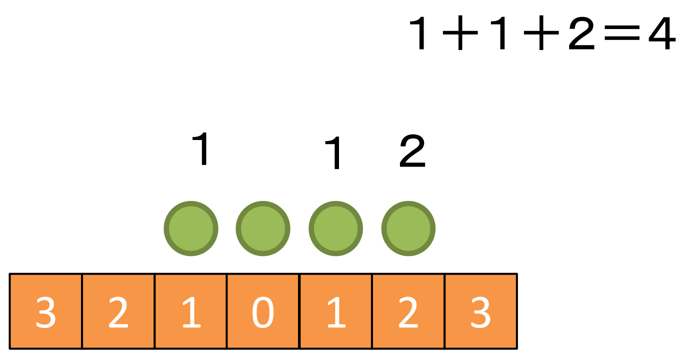




D問題　処理（アルゴリズム）

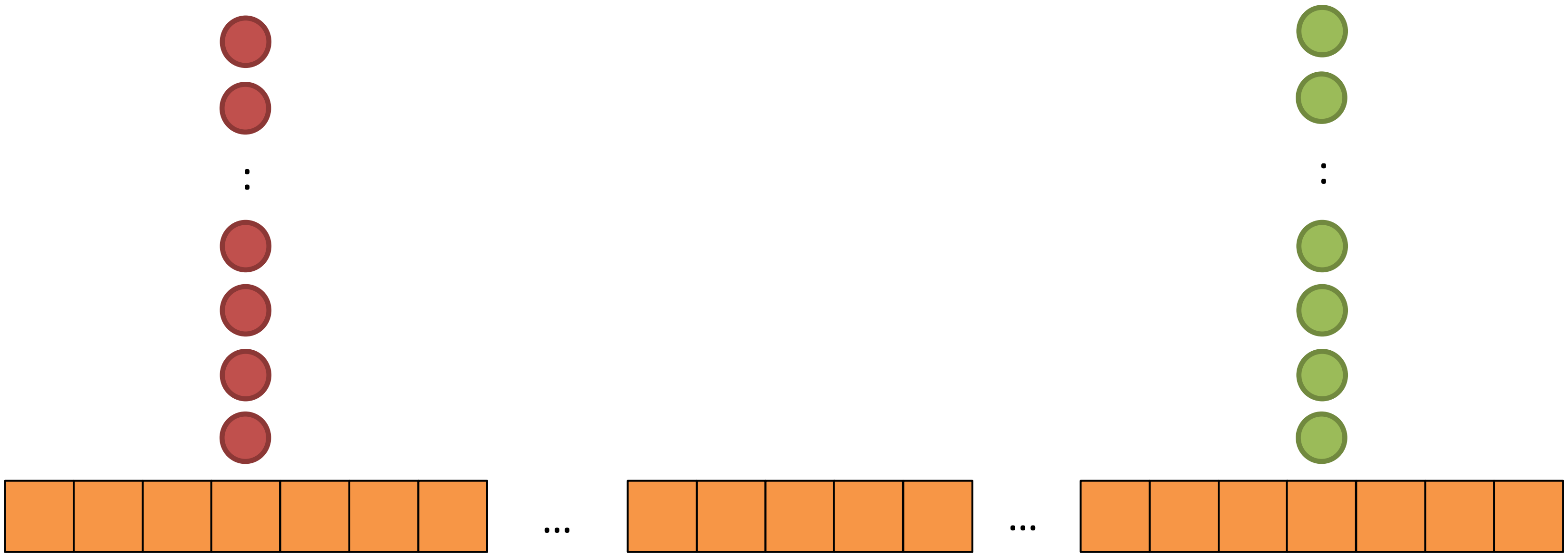
- どのマーブルを右に、どのマーブルを左に・・・と考えるのは非常に面倒！
- 最後の状態だけ考えて、そこから、その状態にするのに必要な手数を考えたい。
- 単純に、移動距離を足し算してあげればＯＫ！





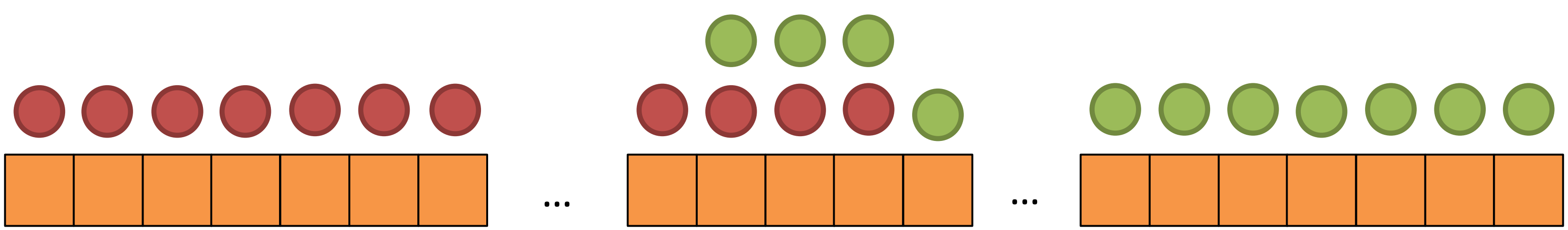
D問題　処理（アルゴリズム）

- R,G,Bが大きくなると・・・



D問題　処理（アルゴリズム）

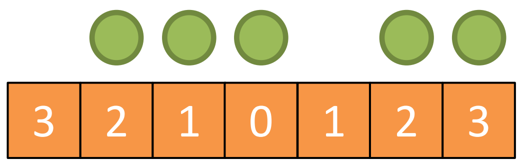
- R,G,Bが大きくなると・・・
- 同じ広げ方をすると、被ってしまうことがある！
  - 単純に右、左、右、左、と考えるだけではダメ。



D問題　処理（アルゴリズム）

AtCoder

- 解法1　広げ方の全探索！
  - 図のように、途中で間が空くことはありえない。



- よって、赤・緑・青のマーブルの入っている、一番左の箱だけ決めれば、最終状態は確定する。
  - これだけだと、それぞれ可能性のある場所が1000近くあるため、微妙に間に合わない。
    - 高速な実装をすればこれでもギリギリ間に合う。

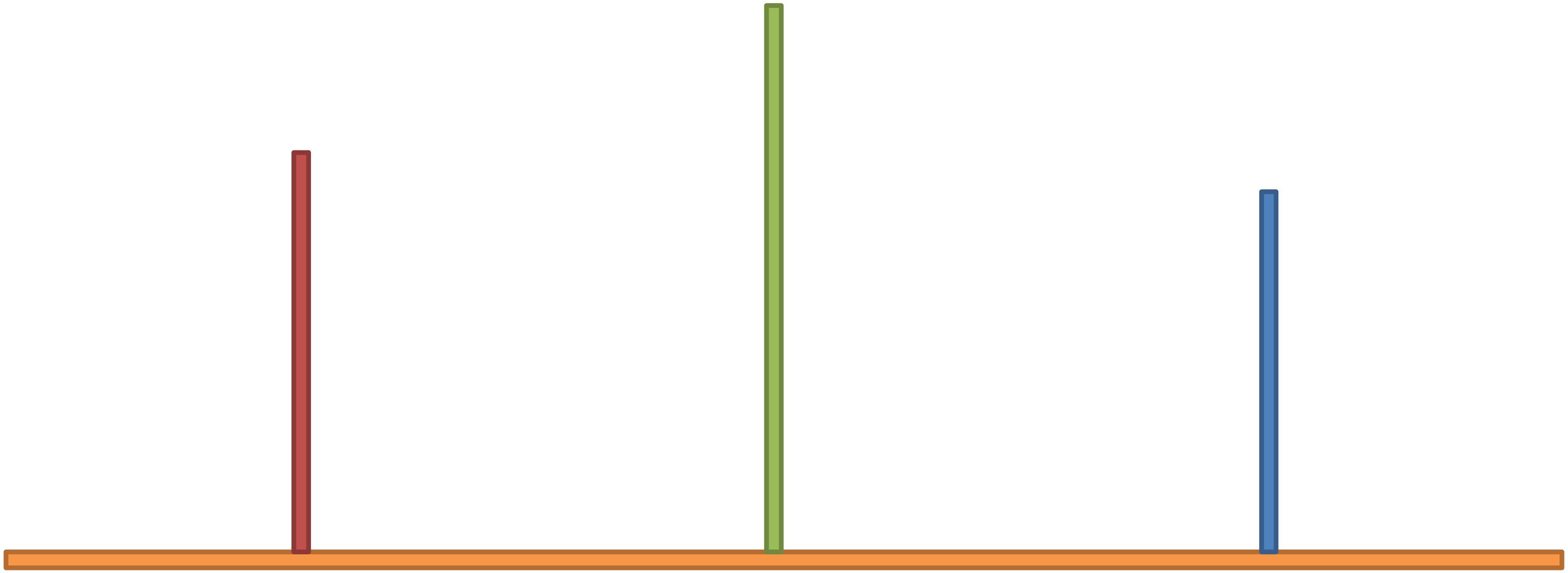
D問題　処理（アルゴリズム）

AtCoder

- 赤、緑、青に対して、それぞれ全て全探索をすると、計算量が3乗になってしまい間に合わない。
  - 何か工夫が必要
- 緑から先に決めてあげることによって、赤と青を別々に計算することが出来る！

D問題　処理（アルゴリズム）

- 例えば図のような場合



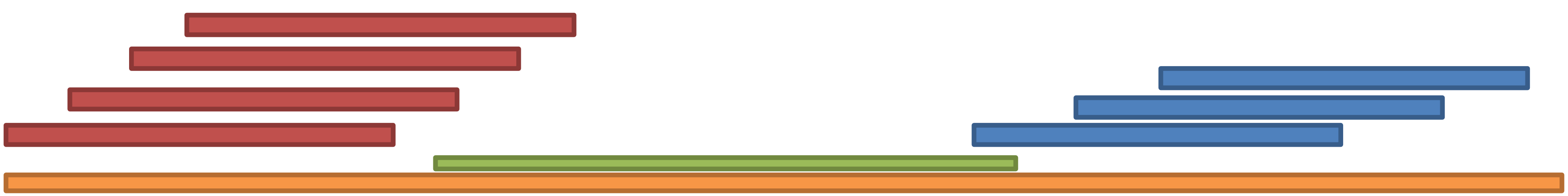
D問題　処理（アルゴリズム）

- 例えば図のような場合
- まず緑の広げ方を決めてあげる

D問題　処理（アルゴリズム）

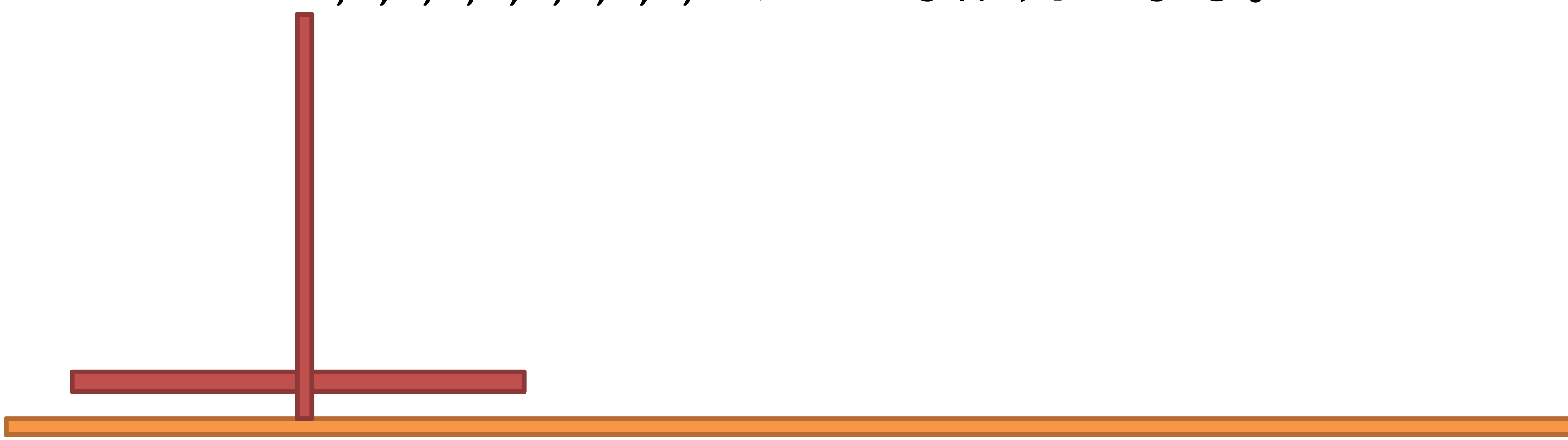
AtCoder

- 例えば図のような場合
- まず緑の広げ方を決めてあげる
- すると、赤の広げ方、青の広げ方は、それぞれの広げ方に干渉しない。
- よって、2乗の計算量で計算することが可能になる！



D問題　処理（アルゴリズム）

- 広げる時のコストは、O(1)の計算量で計算することが可能。
  - それぞれのマーブルの移動距離は、中央で区切ると等差数列になるため、足し算が可能。
    - 3,2,1,0,1,2,3,4,5,6みたいな配列になる。



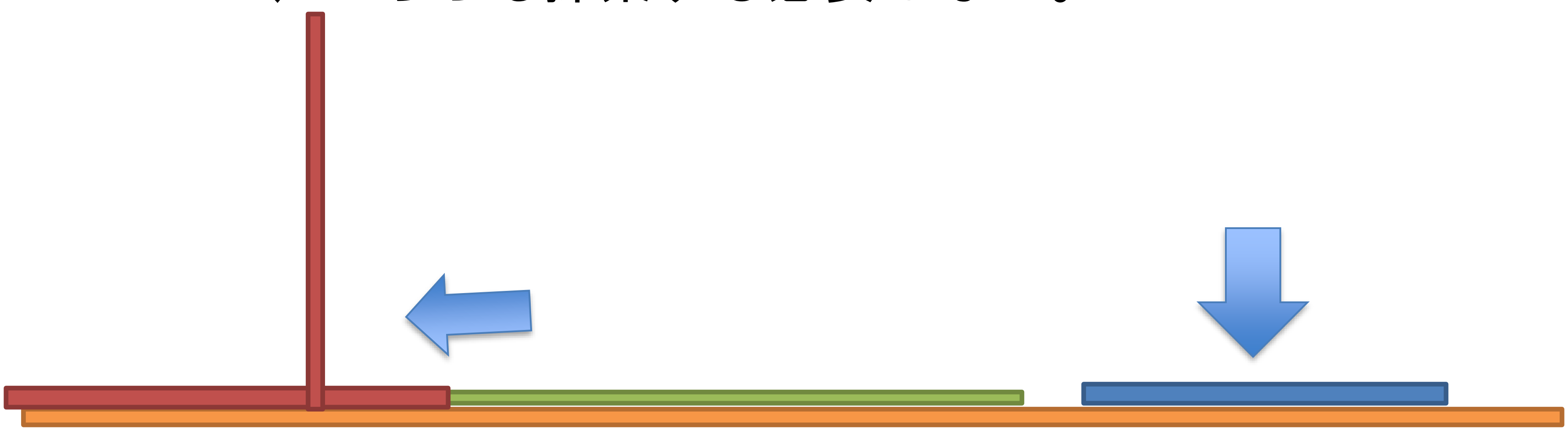
D問題　処理（アルゴリズム）

AtCoder

- そもそも、部分点２と同じ並べ方をまず試してみて、
  それが収まるなら、探索する必要はない。

D問題　処理（アルゴリズム）

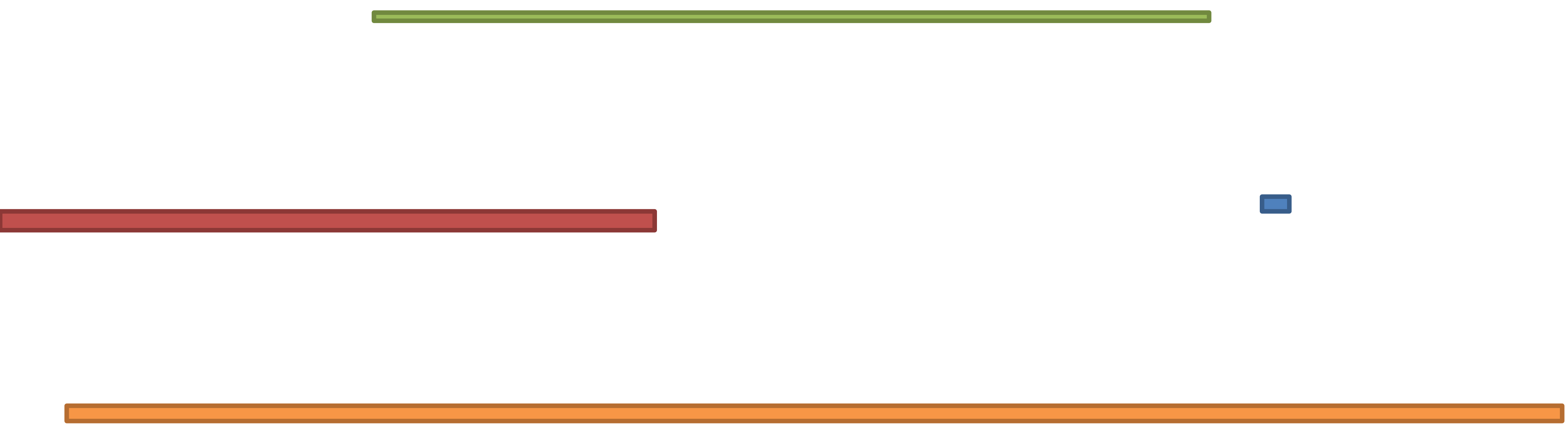
- そもそも、部分点２と同じ並べ方をまず試してみて、それが収まるなら、その解を採用すれば良いので、探索する必要はない。
- 収まらないなら、限界まで中央に寄せてあげれば良いので、こちらも探索する必要はない。



D問題　処理（アルゴリズム）

AtCoder

- 緑が中心だとダメな例！
- 緑・赤が多くて、青が少ない場合。

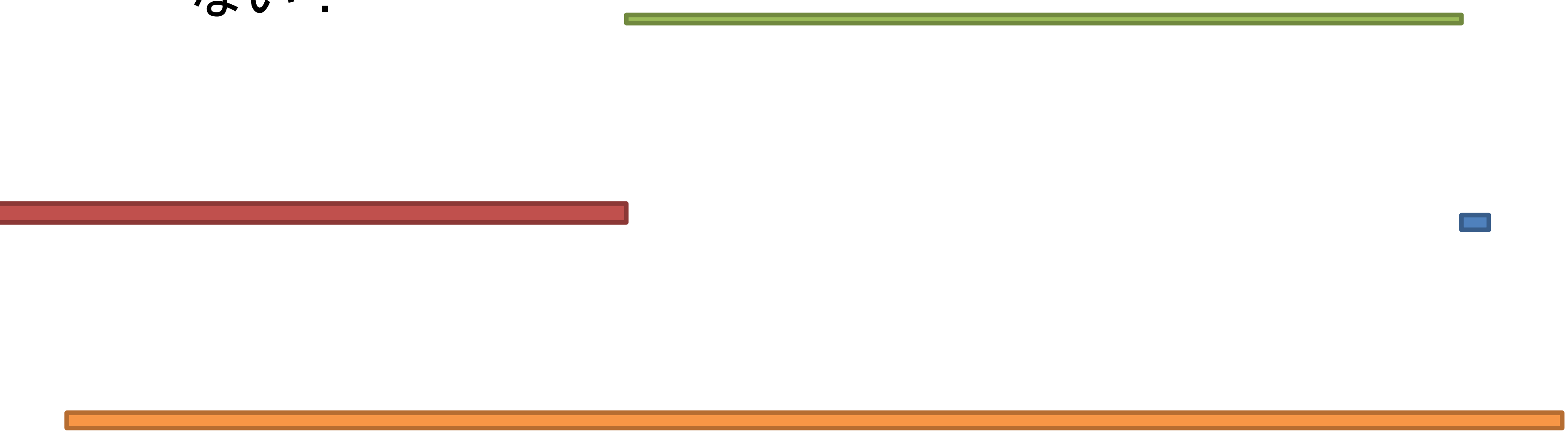


D問題　処理（アルゴリズム）

- 緑が中心だとダメな例！
- 緑・赤が多くて、青が少ない場合。
- 赤を跳ね飛ばすと、赤を大量に移動させないといけなくなってしまう。

D問題　処理（アルゴリズム）

- 緑が中心だとダメな例！
- 緑・赤が多くて、青が少ない場合。
- 赤を跳ね飛ばすと、赤を大量に移動させないといけなくなってしまう。→緑を右に移動してあげないといけない！

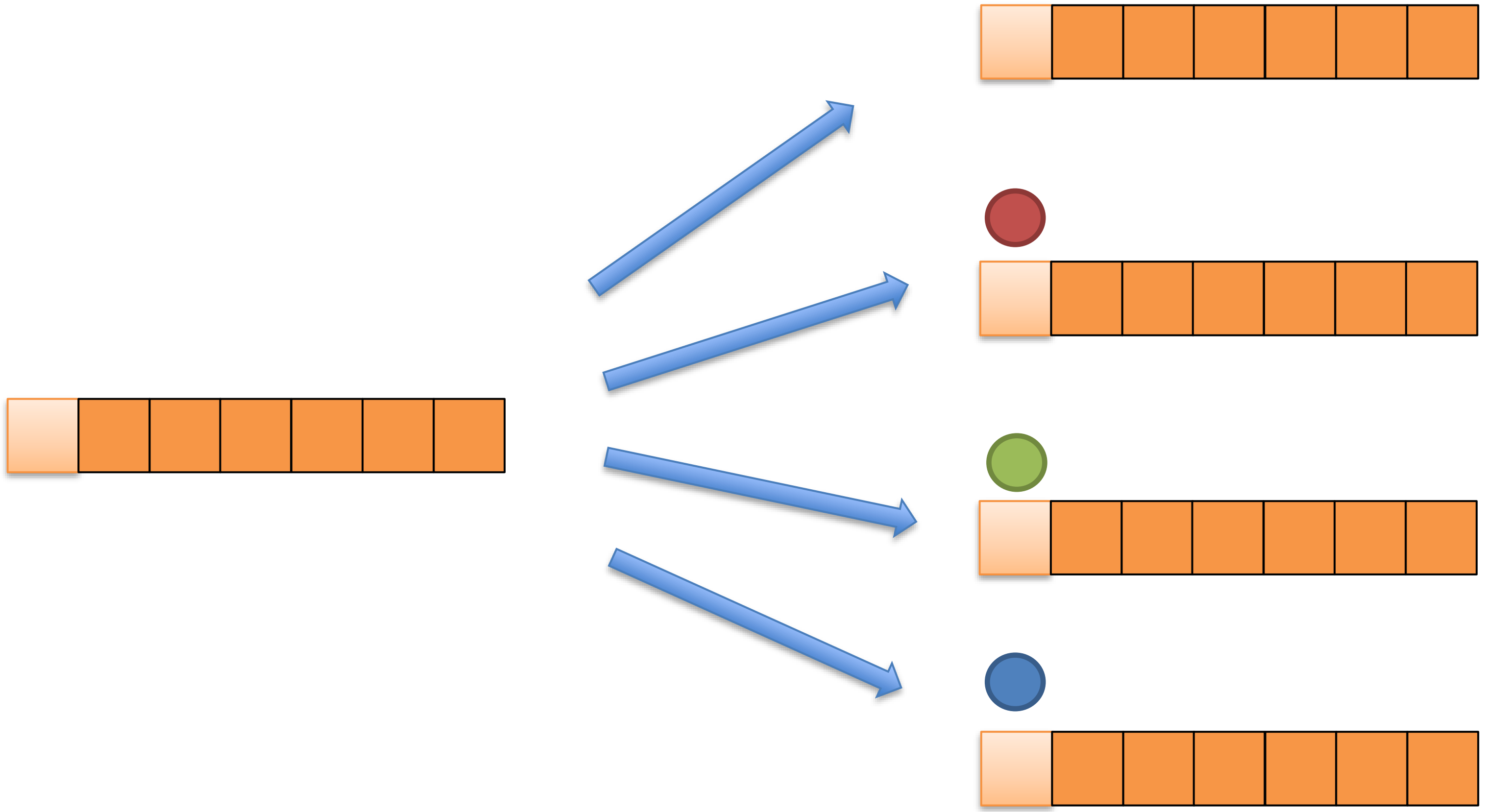


D問題　処理（アルゴリズム）

- 解法２　動的計画法を使おう！
- 左の箱から順番に、「赤を置く」「緑を置く」「青を置く」「置かない」の4通りの全探索が考えられる。
- 範囲は適当に-1000から+1000の箱までやるとして、2000回程度の分岐
- このまま全探索すると、4^2000程度の計算量
  - これを、動的計画法orメモ化再帰をしてあげることにより、解いてあげる。

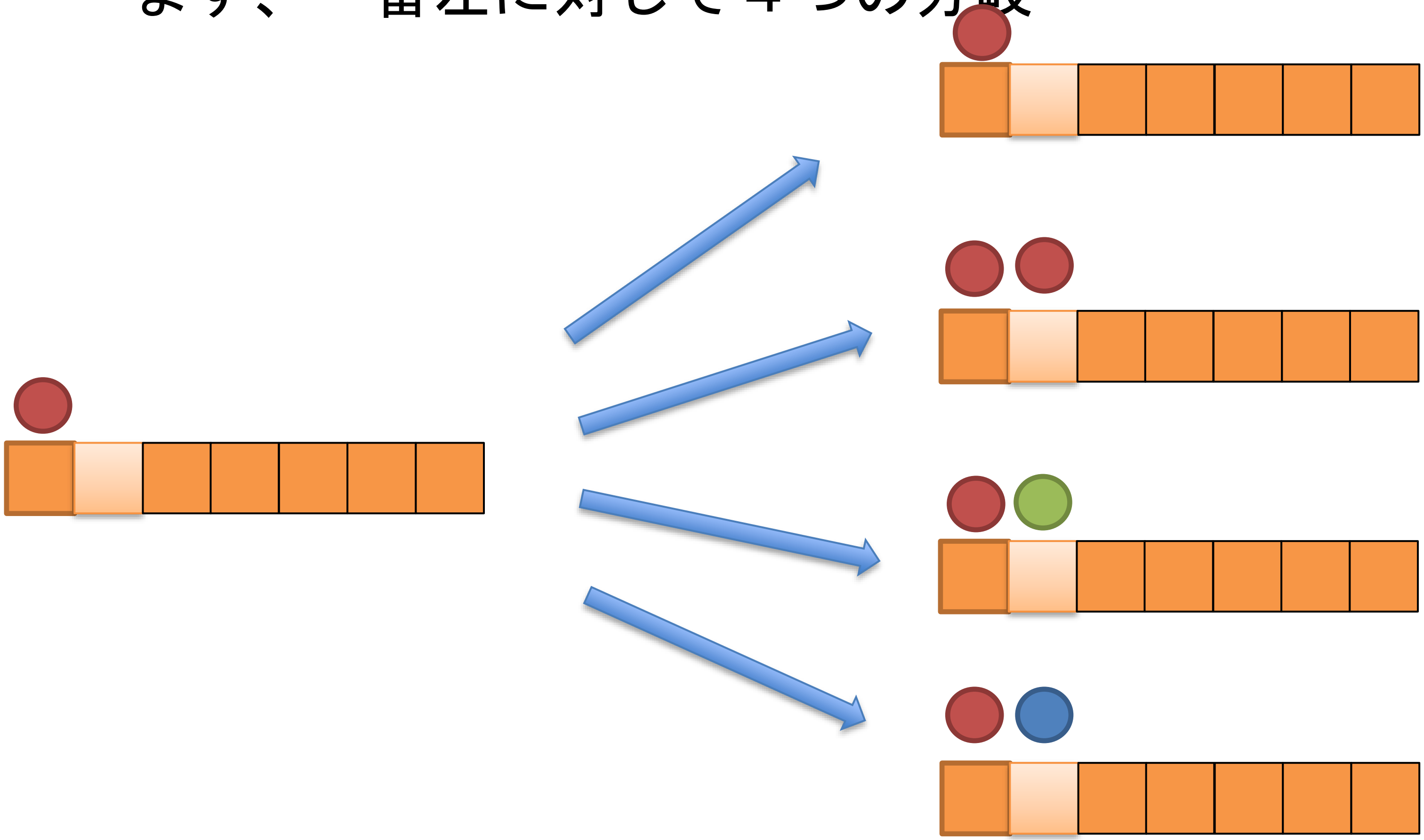
D問題　処理（アルゴリズム）

- まず、一番左に対して４つの分岐



D問題　処理（アルゴリズム）

- まず、一番左に対して４つの分岐



D問題　処理（アルゴリズム）

- 動的計画法・メモ化再帰とは？
- 一度計算したものを、二度計算しなかったり、同じものを纏めて計算してあげることにより、計算量を大幅に削減してあげるテクニック

D問題　処理（アルゴリズム）

- 今回の場合は、左から順番に探索してあげるとして、
  - どの箱を見ているか
  - 赤いマーブルがいくつ残っているか
  - 緑のマーブルがいくつ残っているか
  - 青のマーブルがいくつ残っているか
- しか状態が存在しない。

D問題　処理（アルゴリズム）

AtCoder

- 動的計画法の場合
- 適当な４次元配列dpを用意する
- dp[今見ている場所][赤の残り数][緑の残り数][青の残り数]
  - これに対して、「この状況になるための最小の移動数」を格納してあげるような、計算の省略を行う。

D問題　処理（アルゴリズム）

- 深さ優先探索を利用したメモ化再帰の場合
- int dfs(今見ている場所,赤の残り数,緑の残り数,青の残り数)のような再帰関数を作る。
    - 返り値は、その先で全てのマーブルを配置するために必要な移動数

D問題　処理（アルゴリズム）

- このように動的計画法・メモ化再帰を行うと、それぞれの状態数は、2000 * 300 * 300 * 300程度存在し、それぞれに対して分岐の数が4つ。
  - これでは計算量が大きすぎる！
  - もう少し工夫をしてあげる必要がある。

D問題　処理（アルゴリズム）

- 考察をすることで、計算量を削減しよう！
  - 先ほどは、「赤を置く」「緑を置く」「青を置く」の3通りの置き方を考慮していた。
  - しかし、左から順番に、赤・緑・青と並ぶのが自然であり、赤が置けるなら緑・青を置く必要はないし、緑が置けるなら青を置く必要はない。（今回の問題設定では出来ない！）
  - つまり、分岐の数を4つから2つ（置く、置かない）に減らせる！

D問題　処理（アルゴリズム）

- さらに、置くマーブルの順番を決めてしまえば、それぞれのマーブルの個数でなく、全てのマーブルの残り個数の和だけ覚えておけば、次に置くのはどのマーブルかを求めることが出来る！
    - 残りマーブルの状態数が、300 ^ 3から、300 * 3に削減できる！
    - これなら制限時間内に解くことが可能となる

D問題　処理（アルゴリズム）

- 動的計画法の場合
- 適当な2次元配列dpを用意する
- dp[今見ている場所][マーブルの残り数]
  - これに対して、「この状況になるための最小の移動数」を格納してあげるような、計算の省略を行う。
  - マーブルの残り数に対して、移動量の計算が変わるので注意！

D問題　処理（アルゴリズム）

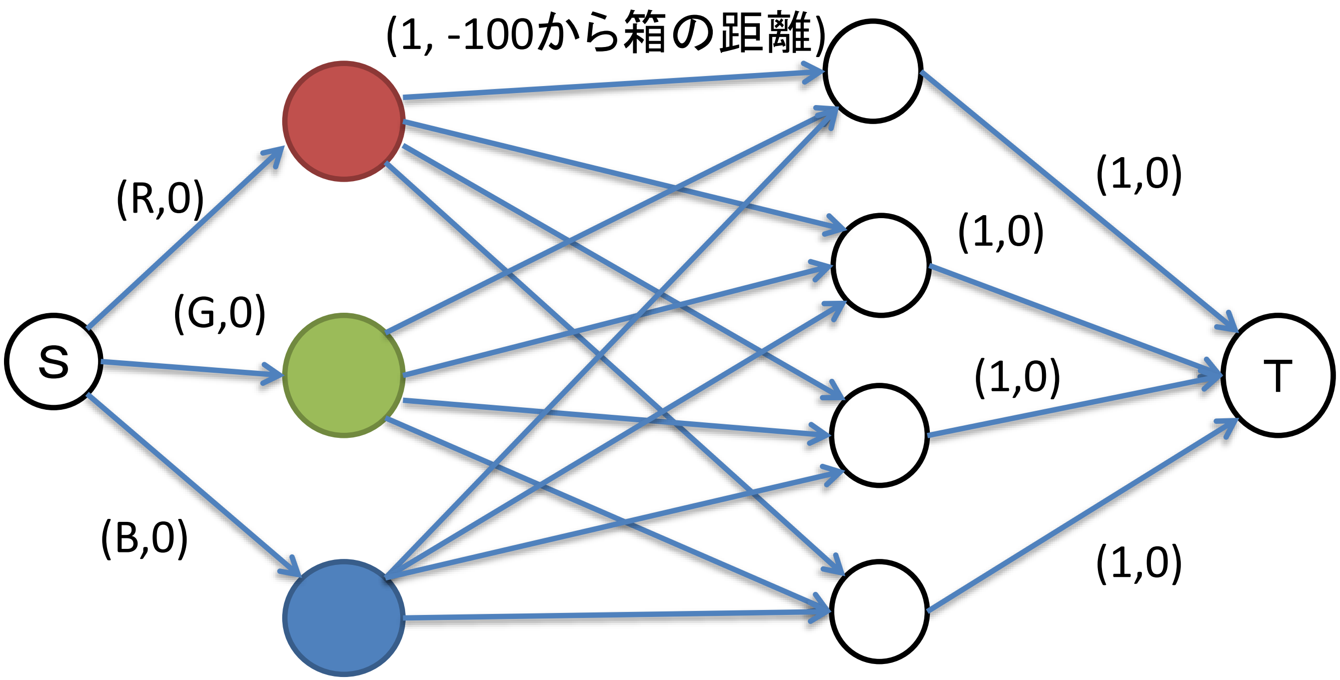
- 深さ優先探索を利用したメモ化再帰の場合
- int dfs(今見ている場所,マーブルの残り数)のような再帰関数を作る。
  - 返り値は、その先で全てのマーブルを配置するために必要な移動数
  - こちらもマーブルの残り数から、R,G,Bどのマーブルを使うか求める必要があるので注意。

D問題　処理（アルゴリズム）

- 解法３　最小費用流を使おう！（想定外でした。）
  - Komakiさんの解法からのアイデアです。
  - http://abc004.contest.atcoder.jp/submissions/132198

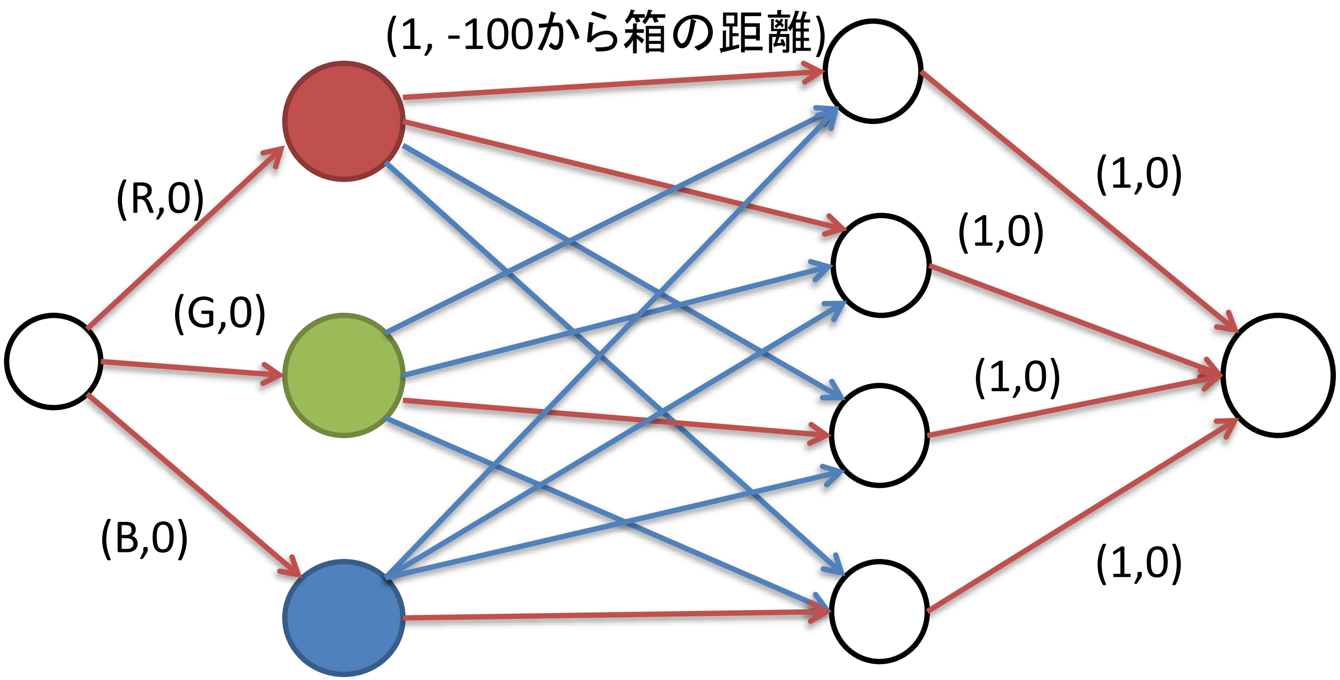
D問題　処理（アルゴリズム）

- こんな感じでグラフを作る
  - 辺には容量と重みを持たせる



D問題　処理（アルゴリズム）

- こんな感じでグラフを作る
  - 辺には容量と重みを持たせる
- このグラフの最小費用流を求める！



D問題　処理（アルゴリズム）

AtCoder

- 最小費用流って？
  - グラフ（丸と矢印で構成されたさっきの図みたいなもの）に関する有名なアルゴリズム
  - 辺に、「容量」「重み」を持つグラフに対して、始点から終点までフローを流す。
    - 各辺には、辺の容量の分だけフローを流すことが可能であり、流すごとに重み分のコストがかかる。
  - 必要なフローを流すために必要なコストの最小値を求めるアルゴリズム。

D問題　処理（アルゴリズム）

- 最小費用流の解き方
  - グラフを作ったら、ダイクストラで頑張る！
  - ダイクストラで1つのフローを流し終えたら、逆の辺を作る。
  - フローが流せなくなるまでダイクストラを繰り返す！
  - 詳しくはググってね！