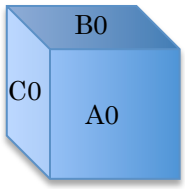
ACM ICPC2014 国内予選　問題 F サイコロ職人

【考察】

- 注文は 6 つの整数が与えられるが、サイコロ面との対応関係に制約がないので、注文の 6 つの整数をサイコロ面に割り当てる割り当て方は、最大 6! = 720 通りある。そこで、サイコロ面に注文の整数が割り当てられたサイコロとしてこの 720 通りのサイコロを、探索の目標として設定する。
- 上記の目標の中には同じ物が含まれている可能性があるので、これをソートして、隣同士を比較して同じ物を取り除く。
- 求める操作列は、辞書式順で最も前にある操作列なので、辞書式順で深さ優先探索を行う。
- 各探索ノードでは、その時点でのサイコロから、目標のどれかのサイコロを作成できる可能性の有無をチェックし、可能性が無ければバックトラックし、可能性があれば、操作として E, N, S, W の順（辞書式順）で再帰的に深さ優先探索を続行する。このとき探索の深さ(level)が q 以下の時は、選択した操作を操作列を格納する配列の level 番目に記憶しておく[1]。
- 辞書式順に深さ優先探索を進めているので、最初に見つかった操作列が辞書式順で最も前の操作列となる。
- サイコロの面に右図のように名前を付ける。A0, B0, C0 の裏面を各々A1, B1, C1 とする。また、A0 または A1 のことを A と表し、B0 または B1 のことを B と表し、C0 または C1 のことを C と表す。



- 目標のサイコロの各面の数値と現在のサイコロの同じ面の数値との差を各々a0, a1, b0, b1, c0, c1[2]とし、a = a0 + a1, b = b0 + b1, c = c0 + c1 とする。現在 A がサイコロの下面だとすると、サイコロを転がした時に次に下面になるのは B または C であり、A が連続して下面になることはない。同様に、現在 B がサイコロの下面ならば、B が連続して下面になることは無く、現在 C がサイコロの下面ならば、次に C が連続して下面になることはない。このことを考慮すると、現在 A が下面の場合

---

[1] 出力は p 番目の操作から q 番目の操作までである。

[2] a0, a1, b0, b1, c0, c1 の内一つでも負ならば目標には達しないのでバックトラックする。

- それ以降にAが下面になる前にはBまたはCが下面になっている必要があるのでb + c >= a でなければならない。
- 最初にBが下面に来てもよく、それ以降にBが下面になる前にはAまたはCが下面になっている必要があるので a + c >= b – 1 でなければならない。
- 同様に、a + b >= c – 1 でなければならない。

以上より、現在Aが下面の場合、目標に到達する為の必要条件として

(b + c >= a) & (a + c >= b – 1) & (a + b >= c – 1) を得る。

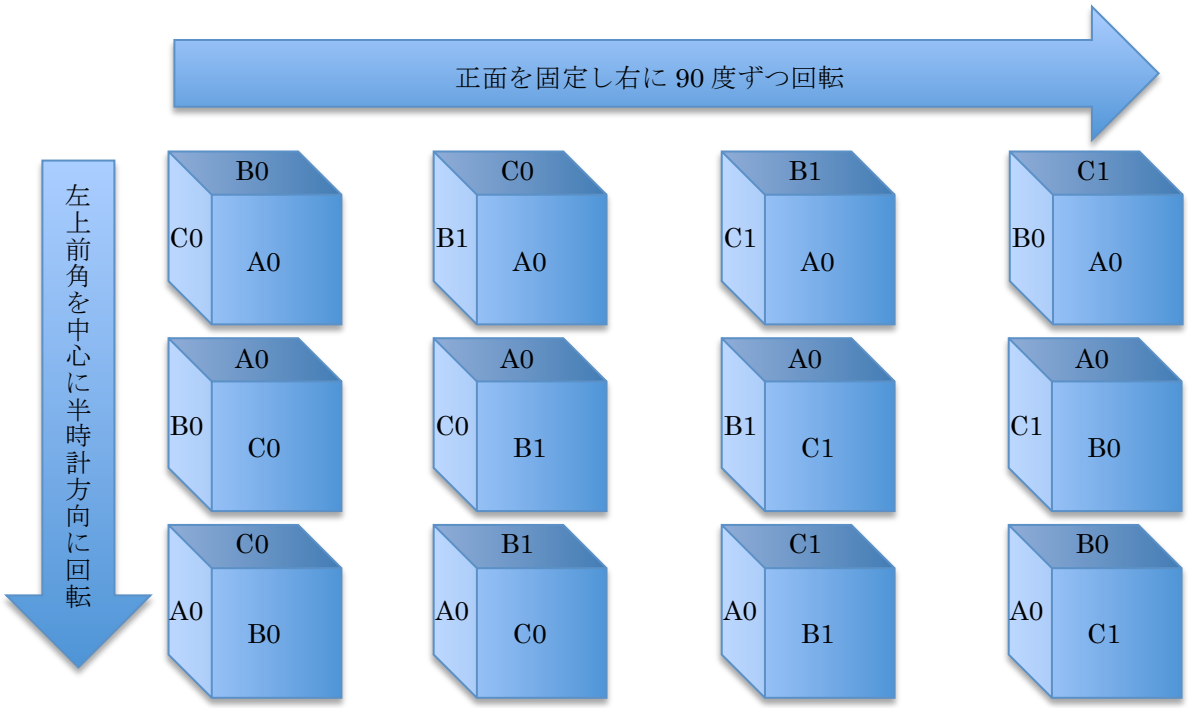
同様に、現在Bが下面の場合、目標に到達する為の必要条件として

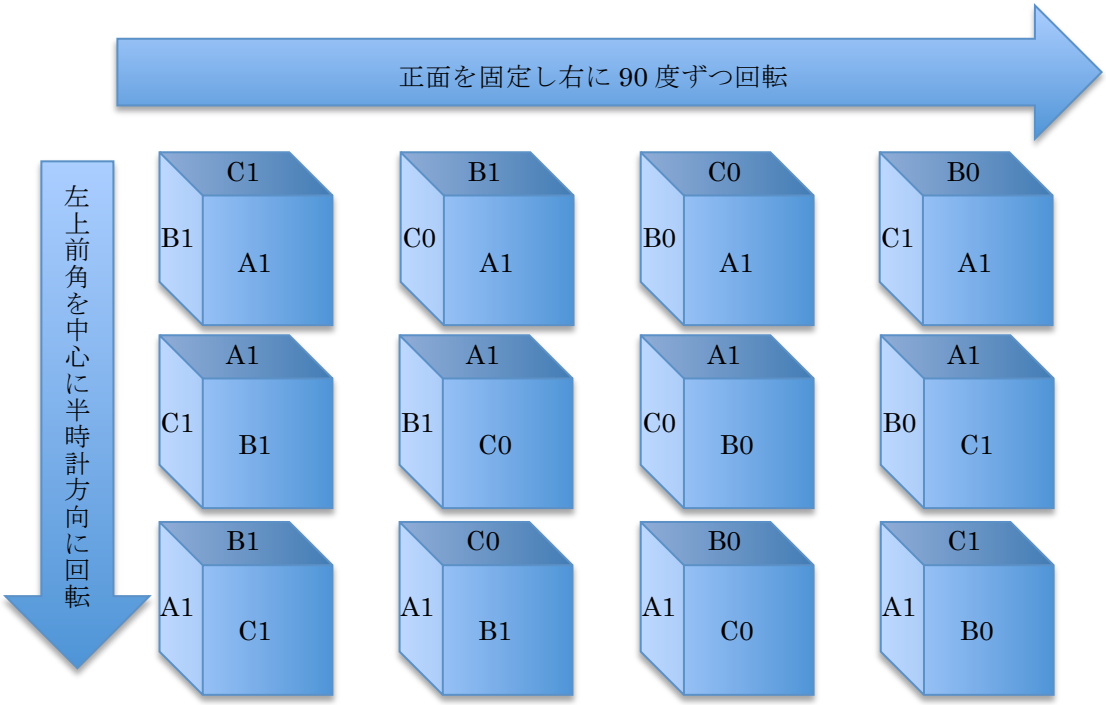
(c + a >= b) & (b + a >= c – 1) & (b + c >= a – 1) を得る。

また、現在Cが下面の場合、目標に到達する為の必要条件として、

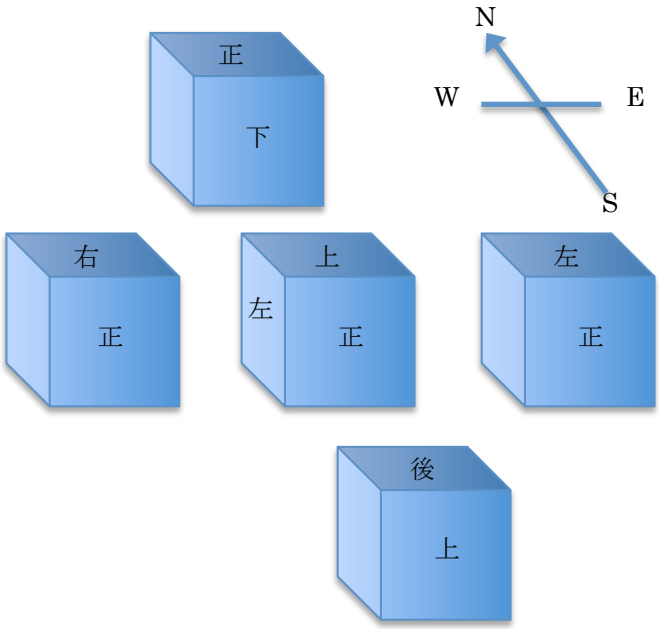
(a + b >= c) & (c + b >= a – 1) & (c + a >= b – 1) を得る。

- サイコロの状態は、サイコロの各面(A0, A1, B0, B1, C0, C1)の数値と、正面（南向きとする）と上面にどの面がきているかで定まる。サイコロの回転操作を容易に実現する為に、サイコロの正面と上面から左面を与える2次元配列left_table[正面][上面]を以下の方法で事前に構築しておく。





- サイコロの回転操作は次のようになる。



【プログラム例】

```
// ACM-ICPC2014 Domestic Problem F サイコロ職人
//   http://icpc.iisf.or.jp/past-icpc/domestic2014/#section_F
//   Filename =        pf4.c
//   Compile:          cc pf4.c
//   Execution:        ./a.out < F0 > F0.out
//   Check:diff F0.ans F0.out

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXT 5000 // t1 ～ t6 の最大値

// 回転方向(方向名の辞書式順に値を付与
#define E 0                     // East
#define N 1                     // North
#define S 2                     // South
#define W 3                     // West
char dir_name[ ] = {'E', 'N', 'S', 'W'};        // 回転方向の名前

// サイコロの面に左手系で名前(A0, B0, C0)を付与
// 値はふつうののサイコロの目の値ー1
#define A0 0
#define B0 1
#define C0 2
#define A1 (5 - A0)   // A0 の裏面
#define B1 (5 - B0)   // B0 の裏面
#define C1 (5 - C0)   // C0 の裏面

// 操作中のサイコロの状態を表す構造体
typedef struct {
   int t[6];            // サイコロ面 A0 ～ C1 の数値
   int top;             // 上面
   int front;           // 正面(南向き)
} state_t;

int t[6];               // t1 ～ t6 を入れる配列
int   p, q;             // p, q の値
int target[720][6];   // 6!通りの目標を格納する配列
int nt;                 // 異なる目標の総数
char opt[MAXT*6+1];     // 操作列を入れる配列

int left_table[6][6];           // left[front][top]=左側のサイコロ面

// サイコロの状態から各面を求めるマクロ
#define LEFT(state) (left_table[state.front][state.top])        // 左面
#define BOTTOM(state) (5 - state.top)                           // 下面
#define BACK(state) (5 - state.front)                           // 後面
#define RIGHT(state) (5 - LEFT(state))                          // 右面

// サイコロの状態 st から目標 target への到達可能性をチェック
```

```
// 戻り値 0: 到達不可能、2: 目標に到達した、 1: 将来到達する可能性有り
int is_valid(state_t st)
{
  int *t;              // check 中の目標値を格納する配列
  int diff[6];         // 現在のサイコロの状態と目標の差を入れる配列
  int i,j;
  int a, b, c;         // 各々{A0,A1}, {B0,B1}, {C0,C1} が下になるべき回数
  int flag;            // -1: 目標到達不可、0: 目標に到達、1: 目標到達可能性有

  for(j=0; j<nt; j++){
    t = &target[j][0];
    flag = 0;
    for(i=0; i<6; i++){
      diff[i] = t[i] - st.t[i];              // 目標値と現状との差
      if(diff[i] < 0){                       // 目標値を超えてしまったので
        flag = -1; break;                    // この目標へは到達できない
      }
      if(diff[i] > 0) flag = 1;              // 目標値の方が大きい->まだ到達可能性有
    }
    if(flag == 0) return 2;                  // diff[i]が全て 0 なので、目標に到達
    if(flag == -1) continue;                 // 現目標へは到達不可。次の目標を check
    a = diff[A0]+diff[A1];                   // A0, A1 が下になるべき回数
    b = diff[B0]+diff[B1];                   // B0, B1 が下になるべき回数
    c = diff[C0]+diff[C1];                   // C0, C1 が下になるべき回数
    switch(st.top){              // 上面（下面）により場合分け
    case A0:
    case A1:                     // 下面が A0 or A1 の場合の必要条件をチェック
      if(b+c < a) continue;      // 目標に到達不能。次の目標を check
      if(c+a < b-1) continue;    // 目標に到達不能。次の目標を check
      if(a+b < c-1) continue;    // 目標に到達不能。次の目標を check
      break;                     // 目標に到達する可能性有り
    case B0:
    case B1:                     // 下面が B0 or B1 の場合の必要条件をチェック
      if(c+a < b) continue;      // 目標に到達不能。次の目標を check
      if(b+c < a-1) continue;    // 目標に到達不能。次の目標を check
      if(a+b < c-1) continue;    // 目標に到達不能。次の目標を check
      break;                     // 目標に到達する可能性有り
    case C0:
    case C1:                     // 下面が C0 or C1 の場合の必要条件をチェック
      if(a+b < c) continue;      // 目標に到達不能。次の目標を check
      if(b+c < a-1) continue;    // 目標に到達不能。次の目標を check
      if(c+a < b-1) continue;    // 目標に到達不能。次の目標を check
      break;                     // 目標に到達する可能性有り
    }
    return 1;                    // 目標に到達する可能性有り
  }
  return 0;                      // どの目標にも到達できない
}

// 指定された方向(dir)へサイコロを回転させる. サイコロの正面は南向き
state_t move(state_t current, int dir)
{
```

```
  state_t new = current;        // 回転後のサイコロの状態を入れる変数
  switch(dir){                  // 回転方向により場合分け
  case E:                       // East
    new.top = LEFT(current);            // 左->上、正面(前)は変化せず
    break;
  case N:                       // North
    new.top = current.front;            // 前->上
    new.front = BOTTOM(current);        // 下->前
    break;
  case S:                       // South
    new.top = BACK(current);            // 後->上
    new.front = current.top;            // 上->前
    break;
  case W:                       // West
    new.top = RIGHT(current);           // 右->上、正面(前)は変化せず
    break;
  }
  new.t[BOTTOM(new)]++;                 // 新しく下に来た面の数値を増やす
  return new;                           // 回転後のサイコロの状態を返す
}

// 辞書式順序で深さ優先探索を行いどれかの目標へ到達する操作列を求める
//    level =  何回目の操作か、  moves =  操作列を記憶する配列
//    st =  サイコロの状態
//    戻り値：   0 = 1 レベル上に戻って次の可能性を探る
//                1 =  どれかの目標に到達
int dfs(int level, char *moves, state_t st)
{
  int valid;
  state_t next;
  int dir;
  int ret;
  valid = is_valid(st);         // どれかの目標への到達可能性を check
  if(valid == 0){               // どの目標へも到達不可能
    return 0;                   // 1 レベル上へ戻って次の可能性を探る
  }
  if(valid == 2) return 1;      // どれかの目標に到達した
  for(dir=E; dir<=W; dir++){    // 各方向に対して辞書式順に
    next = move(st, dir);       // 回転操作を行う
    if(level < q) moves[level] = dir_name[dir];       // レベル q までは操作を記録
    ret = dfs(level+1, moves, next);    // 再帰的に深さ優先探索
    if(ret == 1) return 1;              // どれかの目標に到達したので戻る
                        // この方向への回転では目標に到達しないので次の方向を試す
  }
  return 0;             // 目標に達しなかったので 1 レベル上に戻って次の可能性を探る
}

// サイコロの正面と上の面から左面を求める配列を構築する
void create_left_table(void)
{
  int i,j;
  int t[3];
```

```
  int tt;
  t[0]=A0; t[1]=B0; t[2]=C0;                  // 正面=A0, 上面=B0, 左面=C0
  for(j=0; j<4; j++){                         // 正面を固定し、右回転を４回繰り返す
    for(i=0; i<3; i++){                       // 左上前角を中心に左回転を３回繰り返す
      left_table[t[(i+0)%3]][t[(i+1)%3]] = t[(i+2)%3];
    }
    tt = t[1]; t[1] = t[2]; t[2] = 5 - tt;    // 正面を固定し右回転
  }
  t[0]=A1; t[1]=C1; t[2]=B1;                  // 正面=A1, 上面=B1, 左面=C1
  for(j=0; j<4; j++){                         // 正面を固定し、右回転を４回繰り返す
    for(i=0; i<3; i++){                       // 左上前角を中心に左回転を３回繰り返す
      left_table[t[(i+0)%3]][t[(i+1)%3]] = t[(i+2)%3];
    }
    tt = t[1]; t[1] = t[2]; t[2] = 5 - tt;    // 正面を固定し右回転
  }
}

// 6!通りの目標を再帰的に作成
//   level =  処理レベル（何番目の数値を処理中か）
//   used[ ] =  使用済みを表す配列
//   ct[ ] =  現在作成中の目標
void create_target_sub(int level, int *used, int *ct)
{
  int i;
  if(level == 6){                 // 最後の数値まで処理したので
    for(i=0; i<6; i++)
      target[nt][i] = ct[i];      // その目標を target[][]にコピー
    nt++;
    return;
  }
  for(i=0; i<6; i++){
    if(used[i]) continue;         // 使用済みの数値はスキップ
    used[i] = 1;                  // その数値を使用済みとし
    ct[level] = t[i];             // 目標にセット
    create_target_sub(level+1, used, ct);  // 残り位置にも再帰的にセット
    used[i] = 0;                  // 上に戻って別の組合せを作成する前に、
                                  // その数値を未使用に戻す
  }
}

// 二つのサイコロ面の配列の数値比較を行う関数
int compare(const void *t1, const void *t2)
{
  int *p1 = (int *)t1;            // 整数配列に型変換
  int *p2 = (int *)t2;            // 整数配列に型変換
  int i;
  for(i=0; i<6; i++){  // ベクトルとして大小を比較
    if(p1[i] < p2[i]) return -1;          // t1[ ] の方が小さい
    else if(p1[i] > p2[i]) return 1;      // t1[ ] の方が大きい
  }
  return 0;            // 同じ値
}
```

```
// 指定の数値はサイコロのどの面でも良いので
// 指定された数値から 6! 通りの目標値を作成し
// その中から異なる物を取り出し、その個数を nt にセットする。
void create_target(void)
{
  int i;
  int flag[6];                          // flag[i]: i 番目の数値の使用状況
  int ct[6];
  int j;
  nt = 0;
  for(i=0; i<6; i++)
    flag[i] = 0;                        // 最初、全ての数値は未使用状態
  create_target_sub(0, flag, ct);       // 6!通りの目標を作成
  qsort(&target[0][0], 720, sizeof(int)*6, compare);  // 6!個の目標をソート
  nt = 0;                               // 目標の総数を 0 に初期化
  for(i=1; i < 720; i++){               // 6!通りの目標に対して
    // 最後にセットした目標値と同じならスキップ
    if(compare(&target[nt][0], &target[i][0])==0) continue;
    // 最後にセットした目標と異なる新たな目標なので
    nt++;                               // 目標値総数を 1 増やし、
    for(j=0; j<6; j++)                  // その目標を前詰めでセット
      target[nt][j] = target[i][j];
  }
  nt++;                 // 最初に、先頭の目標を加えているので、総数を補正
}

int main()
{
  int i, ret;
  state_t st;

  while(1){
    for(i=0; i<6; i++)              // t1 ～ t6 を t[0] ～ t[5] に格納
      scanf("%d", &t[i]);
    // 全て 0 なら終了
    if(t[0]==0 && t[1]==0 && t[2]==0 && t[3]==0 && t[4]==0 && t[5]==0)
      break;
    scanf("%d %d", &p, &q);     // p, q の読み込み
    create_target();            // 6!通りの内、異なる目標を生成
    create_left_table();        // 正面,上面から左面を求める配列を作成
    for(i=0; i<6; i++)          // サイコロの全ての面の初期値は 0
      st.t[i] = 0;
    // 正面と上面の初期値を決める。値は全て 0 なので実現可能であれば何でも良い
    st.front = A0;              // ここでは、A0 を正面、
    st.top = B0;                // B0 を上面とする。
    opt[q] = 0;                 // 操作列を長さ 0 に初期化

    ret = dfs(0, opt, st);      // 辞書式順深さ優先探索で目標への操作列を求める

    if(ret == 1){               // どれかの目標に達したので
      opt[q] = 0;
```

```
            printf("%s¥n", &opt[p-1]);          // p 番目から q 番目までの操作列を出力
        }
        else printf("impossible¥n");           // どの目標にも到達不能
    }
}
```