連載

サイコロパズル

寺田　実（電気通信大学情報通信工学科）
terada@ice.uec.ac.jp

■問題

今回は 2004 年 11 月に愛媛大学で行われたアジア地区大会の問題 F, “Dice Puzzle” をとりあげる．問題は以下より入手可能である：
http://www.ehime-u.ac.jp/ICPC/jp/

通常のコンテストの問題文は，開催地にちなむ故事来歴を含む，冗長ともいえる導入部分から始まるのが常である．一部には，最初の段落は読まずにスキップしてもよいという説さえある．しかるに，この問題には，そのような導入部が一切なく，単刀直入に問題の説明が始まる．多くの問題を見てきた参加者は，これを見るだけで，この問題が通常のものとは異なっていることに気づくかもしれない．

図 -1 のような普通のサイコロを 27 個，図 -2 に示すように 3 × 3 × 3 の立方体に積み上げる．そのときに制限が 1 つあって，接する面と面の目の合計が 7 となるようにしなければならない（図 -3）．

こうしてできた大きな立方体の正面と上面について，それぞれの 9 個の目のうちのいくつかを入力データに基づいて固定する．その条件下で，右側面の 9 個の目の可能な配置をすべて求めるのが問題である．より正確にいうと，右側面の 9 個の目の合計を重複を除いてリストアップするのが目標である．

問題文にある例を見よう．

その例における上面と正面の目の指定を図 -4 に示す．空白の場所は，どの目でも構わないことを示している．これを，入力データとしては以下のように表現する：

```
1 0 0
0 2 0
0 0 0
5 1 2
5 1 2
0 0 0
```

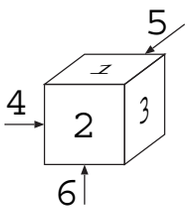

図 -1　サイコロ

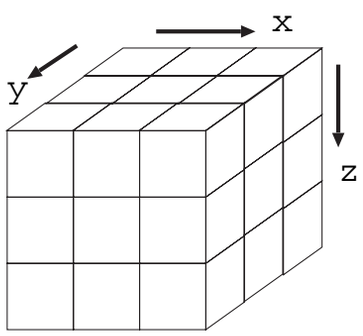

図 -2　積み重ねたサイコロ

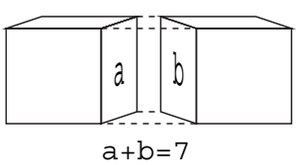

図 -3　サイコロの隣接制限

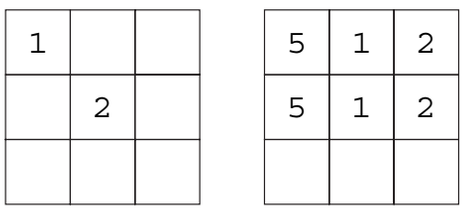

図 -4　上面（左），正面（右）の目の指定の例

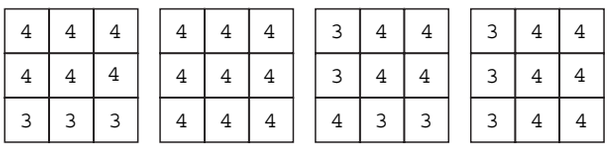

図-5　図-4のもとでの右側面の配置

最初の3行が上面を，次の3行が正面を表し，0はどの目でも構わないこと（図では空白で示した）を表している．

出力として求められているのは，右側面の目の合計を昇順にソートしたもので，この問題例の場合には

```
32 33 36
```

が答である．実際の右側面の配置は図-5に示す4通りとなる．

他の多くの問題と異なり，この問題にはサイズに関するパラメータはなく，したがってその値の範囲などの制限も一切ない．

■単純解法

まず，出発点として，バックトラックを用いて27個のサイコロの向きをすべて試みる方式を考えてみよう．

サイコロ1つを置く向きは，上面の目が6通り，それぞれについて鉛直軸周りの回転が4通りあるので，合計すると24通りとなる．したがって，まったく制限のない場合の探索空間の広さは

$$24^{27} \simeq 1.8 \times 10^{37}$$

となり，容易ならざるサイズである．

ただ，幸いなことに「隣接する目の合計は7」という制約がある．このことは，表面から見たときに奥に向かってならんでいる3個のサイコロの目がすべて同じになることを示している．表面は3×3が3面あるから27個の目が見えており，それらが1から6までの値をとり得るから

$$6^{27} \simeq 1.0 \times 10^{21}$$

となり，よほど改善されたとはいえまだまだである．

実際には，表面の27個の目にはまだまだ制約がある．たとえば，あるサイコロを表面に投影した個所

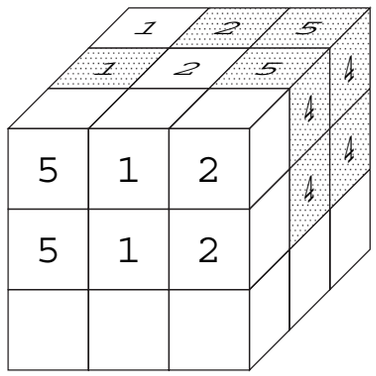

図-6　図-4で必然的に定まる部分（灰色部）

（3面ある）は，そのサイコロの目そのものになるから，2つが決まればもう1つは自動的に決まる．つまり，27個のうちの1/3は自動的に決まることになり，

$$6^{18} \simeq 1.0 \times 10^{14}$$

まで下がってきた．

しかし，いずれにせよ力で押し通すには大き過ぎるし，このコンテストの性格として何らかの工夫が要求されているはずなので，単純解法はいったんおいて，工夫を考えてみよう．

■工夫—1

このように方針が立たないときには，人間が問題を解くプロセスに学ぶという方法がある．

人間であれば，まず向きが決まるサイコロだけを次々に決定していき，それが行きづまったところで，はじめて複数の候補のどちらかを選ぶという分岐の処理に入るであろう．たとえば最初に述べた例のパターンを考えよう．正面のパターンと上面のパターンから必然的に向きが定まるサイコロが12個もある（図-6）．ここでやむなく分岐の処理に入るが，実際に発生する枝の数はたった4つで，これなら手で計算してもたいした労力ではない．

この方法は，要するに，向きを定めるサイコロの順番を適切に選ぶことによって，探索における分岐を減らし，必要となる計算をおさえようとするものである．このような処理を，機械的なバックトラックと対比して知的バックトラック（intelligent backtrack）といい，発見的方法（heuristics）の1つと位置付けることができる．

しかし，この方法は，この問題例のような「たちの

いい」入力データには有効であるが，すべての入力データに有効とは限らない．

たとえば極端な例として，正面と上面の目が1つも指定されていないケースを考えると，いかに知的バックトラックといえども順番についてあまり工夫のしどころがなさそうである．

また，知的バックトラックに付随する問題点として，アルゴリズムが複雑になる点も見逃せない．

発見的方法というのは「大勢を占めるたちのいい問題をいかにうまく解くか」が眼目なので，コンテストの問題のように「あらゆる可能な入力に対して正解を求める」種類の問題にはあまり有効ではないように思う．

■工夫—2

もう1つ別なアプローチとして，状態（この問題でいえば，サイコロの向き）の集まりを1つの値として保持し，多数の状態を一度に処理してしまうというアイディアがある．

サイコロ1つは24通りの向きのいずれかをとるから，その集合を表現するには24ビットのビットベクタがあればよい．まったく制限のない場合には1が24個並んでいるが，制限が加わる（たとえば上面の目が決まる）たびに1のビットを減らしていくという方針でいく．

多数の状態を1つに重ね合わせるというあたりに何やら量子計算を思わせるようなテイストもあるし，何より24ビットなら整数型に収まるという利点もある．

しかし，この方法は，状態の間に相互作用が生じた段階で無力になる．たとえば，あるサイコロが2つの状態をとり得るとして，別のサイコロが前者との対応関係に応じて2つの状態のいずれかになる，という状況を表現する手段がないのである．どちらも2つの状態をとる，ということは表現できても，それら2つを合わせたときに状態がどうなるかが分からないのである．

というわけで，この方針もうまくいかないことが明らかになってしまった．

コンテスト参加者の立場だと，これ以上の工夫を思いつかない限りこの問題はあきらめることになるかもしれない．しかしまた，コンテストの問題は「必ず正解がある」という性質も有している．だとすると，残った方法は単純な方法でとにかくやってみる，ということになろうか．

■単純な方法ふたたび

単純なバックトラックによって，サイコロの向きを順次定めていく方法でプログラムしてみよう．

前に検討した通り，サイコロの向きを定めるということはすなわちそれが投影される3つの表面の目を定めることに相当する．

まず，用いるデータは以下の通り：

```
/* サイコロの表現 D[top][front]=right */
int D[7][7];
/* 各面の制約
   サイコロ (x,y,z) の
   正面 FR[z][x]
   上面 TP[y][x]
   右側面 RT[z][y]
 */
int FR[3][3];  /* 正面 */
int TP[3][3];  /* 上面 */
int RT[3][3];  /* 右側面 */
```

2次元配列Dは，サイコロの形状を表している．2つの添字に上面と正面の目（1から6の整数）を与えたとき，右側面にくる目が配列の値である．矛盾のある上面と正面を指定したときには，値が0となって検知できるようにしてある（そのため，本来なら24通りしかない配置のために6×6の場所を使っている．さらに，添字を1からはじめるために無駄をした）．

2次元配列FR,TP,RTは，正面，上面，右側面の表面に見えている目を保持する．値0は，まだ定まっていないことを示すのに使う．この配列の値を定めることがすなわちサイコロの向きを決定していくことに対応する．

まず，サイコロ形状の配列Dを初期化するコードから説明する．たかだか49要素しかないし，実際のコンテストでは手で計算して初期化してしまうのも良いかもしれないが，ここで間違いが入ると重大である．プログラムで初期化してみよう．

```
/* 上面をtに固定して，水平面で回転 */
void init_D_rot(int t, int f, int r)
{
  int j;
  int tmp;
  for(j=0; j<4; j++){
    D[t][f] = r;
    tmp = f;
    f = r;
    r = 7-tmp;
  }
```

```
}

/* 上面を 6 通りすべて変えて D[][] を設定 */
void init_D(void)
{
  /* 正しいサイコロの一状態 */
  int t = 1, f = 2, r = 3;
  int i, tmp;

  for(i=0; i<4; i++){
    /* 水平面で一周 */
    init_D_rot(t, f, r);
    /* ひとつ後ろへ回す */
    tmp = t;
    t = f;
    f = 7-tmp;
  }
  /* 右側面を上面へ */
  tmp = r; r = f; f = t; t = tmp;
  init_D_rot(t, f, r);
  /* 底面を上面へ */
  t = 7-t; tmp = f; f = r; r = tmp;
  init_D_rot(t, f, r);
```

それではいよいよ処理の本体．まず，入力データに基づいて，あらかじめ定まっている正面と上面の値を配列 FR,TP にセットする．それ以外の場所は未定をあらわす 0 にしておく．

処理は，27 個のサイコロを順次訪れることで進める．注目したサイコロに対して，上面，正面，右側面の制約のかかり方に応じて以下のような場合わけになる．

- 一意に定まる
  そのまま次のサイコロの処理に再帰する．分岐がないので，バックトラックも起こり得ない．
- 矛盾がある
  失敗したのでバックトラックのために帰る．
- 複数の候補がある
  候補を順次選んでは次のサイコロの処理に再帰する．戻ってきたところでバックトラック処理として次の候補を選び直す．

これらを条件判断で振り分けてもよいのだが（プログラムの初期の版はそのようにしてあった），複数の候補があるとして扱うことでプログラムが非常に簡潔になるので，ここではその版を示してある．

```
void solve1(int x, int y, int z, int v);

/* 右側面の一ます (RT[y][z]) を決めて進む */
void solve3(int y, int z){
  int v;

  for(v=1; v<=6; v++){
    RT[y][z] = v;
    solve1(0, y, z, v);
    RT[y][z] = 0;
  }
}

/* サイコロひとつ (x,y,z) の右の目 (v) を
   決めて進む */
void solve1(int x, int y, int z, int vr){
  if(x >= 3){
    /* サイコロ 3 個 ( 同じ右側面 ) の処理完了 */
    if(y >= 2){
      if(z >= 2){
        /* 全部のサイコロ完了 */
        solved();
      } else {
        /* 次の列へ */
        solve3(0, z+1);
      }
    } else {
      /* 次の列へ */
      solve3(y+1, z);
    }
  } else {
    /* 同一の y, z の列内 */
    int vf;
    for(vf=1; vf<=6; vf++){
      /* 正面の目を順に試みる */
      if (FR[z][x] == 0 || FR[z][x] == vf)
{
        /* 正面が未決定か , 整合していれば */
        int vt = D[vf][vr];
        /* 上面を算出して */
        if (vt != 0 &&  /* 矛盾がなく */
            (TP[y][x] == 0 ||
             TP[y][x] == vt)) {
          /* 上面が未決定か , 整合していれば */
          int vf_old = FR[z][x];
          int vt_old = TP[y][x];
          FR[z][x] = vf;
          TP[y][x] = vt;
          /* 次のサイコロへ */
          solve1(x+1, y, z, vr);
          TP[y][x] = vt_old;
          FR[z][x] = vf_old;
        }
      }
    }
  }
}
```

solve3 は，側面を新たに定めるときに呼び出す関数で，特に main からは，solve3(0,0) を呼ぶ．

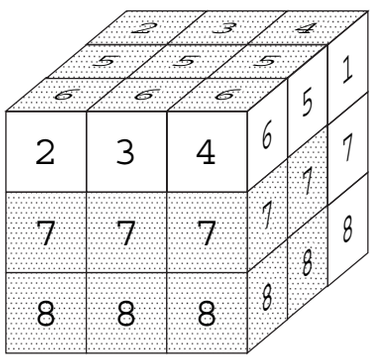

図-7　制約なしのもとで目を決める順

solve1 は，側面を 1 つ定めたのち，同じ側面を共有するサイコロを 1 つずつ吟味するための関数である．再帰の順は，solve3 → solve1 → solve1 → solve1 → solve3... となる．

関数 solved は解が一通り見つかるごとに呼び出される．実際の問題では，右側面の目の和を計算する処理を行うことになろう．

関数 solve1 の冒頭で，サイコロの処理が横一列完了したかどうかの判定をしている部分のコーディングが見苦しい．本来ならば，x,y,z に関する 3 重のループで処理したいところなのだが，次の処理に進むというのが再帰呼び出しである以上，ループでは記述できないのである．

■結果

プログラムは一応できたが，果してこれが常識的な時間で動くであろうか．コードを見たところでは，分岐が発生するのは比較的限られた状況だけのようにも見える．

実際に，最悪と思われる「上面と正面に一切制約なし」の入力でやってみた．驚いたことに，計算時間は 0.1 秒以下で済んでしまい，求まる解の数（関数 solved の呼ばれた回数）もたったの 15,360 にすぎなかった．指数爆発を予想していたのがまったくはずれてしまったのである．

実は，この問題の探索空間は，本稿のはじめで述べたような巨大なものではなかったのである．「一切制約なし」の入力に対して，「表面の目を順に決めていく」という前述のアルゴリズムに従って進めていく際の，可能な分岐数を見積もってみよう．図 -7 の白色の 1, 2, … は，目を定めていく際に複数の可能性がある場所と，その検討順序を示している．まず白色の 1 の目を定め，次に白色の 2 の目を定めるように処理が進行する．灰色の 1, 2, …は，対応する白色の目によって必然的に定まる場所を示している．たとえば，正面の白色の 2 を定めることによって，側面の白色の 1 と合わせて 2 面が定まるサイコロができる．そこでは上面が必然的に定まるので，そこを灰色の 2 としてある．これをみると，自由に決めることのできる場所は 8 カ所で，そのうちの 1 だけが 6 通り，残りはすべて 1 つの面が定まっているために 4 通りとなる．したがって，全体としての探索空間は

$$6 \times 4^7 = 98304$$

と，信じられないくらい小さかったのであった．

さて，この問題の教訓は何なのだろうか．

一見すると指数爆発しそうな問題であっても，実際にはきわめて分岐が少ないことを見抜いて，迷わず単純バックトラックのプログラムを作るのが期待されているのだろうか．

それとも，困難そうに見える問題でも，まずとにかく単純なプログラムを作って，その振る舞いからさらなる改良の道を考えていくという方針が期待されているのだろうか．

筆者は残念ながら探索空間の小ささを見抜くことができず，ずいぶん遠回りしてしまった．

この問題は，プログラム技術そのものよりも問題に関する深い洞察が要求されるという点において，異色のものといえるのではないかと思う．

（平成 16 年 12 月 10 日受付）