# 嘘つき島の問題

石畑　清（明治大学理工学部）
ishihata@cs.meiji.ac.jp



今回は，2002 年 11 月，金沢大会の問題 G「True Liars」を取り上げる（問題は http://www.kitnet.jp/icpc/ 参照）．孤島に嘘つき人間と正直人間が住んでいる．見かけからは各人がどちらに属するか分からない．いくつかの質問をして，それに対する返答から，誰が正直で誰が嘘つきかを明らかにするという趣旨である．論理パズルによく見る設定だが，プログラミングの問題としてもなかなか面白い．

## ■問題

正直部族の人間が $p_1$ 人，嘘つき部族の人間が $p_2$ 人住んでいる島がある．$p_1$ と $p_2$ は既知である．しかし，合計 $p_1+p_2$ 人の人間，それぞれがどちらに属するかを見かけから判断することはできない．嘘つき人間は常に嘘をつく．つまり，どんな質問に対しても，真実とは違う答を返す．当然ながら，正直人間は常に正しい答を返す．彼らにいくつかの質問をして，その答から，各個人が正直か嘘つきかを判定することが問題である．

質問は，$x$ 番の人間に対して「$y$ 番の人間は正直か」と聞くことしか許されない（$1 \leqq x, y \leqq p_1+p_2$）．$x$ と $y$ は自由に選ぶことができる．たとえば，$x=y$ として，質問した相手自身のことを聞いてもかまわない．

プログラムの入力は，次のような形式になる．

$$
\begin{array}{ccc}
n & p_1 & p_2 \\
x_1 & y_1 & a_1 \\
x_2 & y_2 & a_2 \\
 & \cdots & \\
x_n & y_n & a_n
\end{array}
$$

最初の行には，3 つの整数がある．$p_1$ と $p_2$ はすでに説明したとおり．$n$ は質問の総数である．2 行目以降に，質問とそれに対する答が与えられる．$x_i$ と $y_i$ は，質問した相手とその質問で参照している人間の番号である．$a_i$ は，yes または no の文字列である．

プログラムは，入力された質問と答から，何番の住人が正直かを答える．ちょうど $p_1$ 個の番号を答えることになる．もちろん，質問が不十分なら，誰が正直か正確には突き止められないことがある．複数の組合せが可能性として残ってしまう場合である．このような場合は，単に失敗と報告すればよいことになっている（no と出力する）．

入力データに矛盾がないことは保証されている．$p_1$ と $p_2$ は，それぞれ 300 未満である．また，質問数 $n$ は 1000 未満と規定されている．

## ■問題の分析

この問題には，いくつかの山がある．それが面白いところなのだが，順を追って説明しよう．まず，入力データからどんな推論ができるかを考察しなければならない．

「$y$ は正直か」という質問に $x$ が yes と答えたとしよう．このことから何が分かるだろうか．この時点では，$x$ が正直か嘘つきか分かっていないので，確定的な結果は得られない．しかし，それなりの情報を得ることは可能である．

もし $x$ が正直なら，$y$ も正直である．これは間違いない．逆に $x$ が嘘つきなら，$x$ が言うことはすべて嘘なのだから，$y$ が正直だという答も嘘である．つまり，$y$ も嘘つきであることが分かる．この 2 つを合わせて，yes という答があったときは，$x$ と $y$ が同じ部族の人間であると結論できる．

「$y$ は正直か」という質問に $x$ が no と答えた場合も同様に推論できる．この場合は，$x$ と $y$ が違う部族に属するというのが結論である．

いずれの場合も，$x$ と $y$ の関係は対称である．$x$ が「$y$ は正直だ」と言った場合も，$y$ が「$x$ は正直だ」と言った場合も，得られる情報は同じである．

$x$ 番と $y$ 番が同じ部族であることを $x=y$，違う部族であることを $x\neq y$ と書くことにする．たとえば，$1\neq 2$，$3\neq 4$ が分かった後で，$1=3$ が分かったとする．1 と 3 は同じ部族である．また，2 は 1 と違うし，4 は 3 と違う．1 や 3 と違う部族は 1 つしかないのだから，2 と 4 は 1 や 3 とは逆の同じ部族に属することが分かる．まとめると，1 と 3 が同じ部族，2 と 4 が同じ部族で，2 つのグループは逆の部族である．

このようにして，質問の答が得られるたびに，同種の人間のグループが徐々に大きくなっていく．これが第 1 の推論過程である．推論の途中では，次のようなグループ分けの形で知識が表現される．

$(f_1,\ g_1)$
$(f_2,\ g_2)$
...
$(f_m,\ g_m)$

$f_i$ と $g_i$ は，それぞれ人間の番号の集合である．集合 $f_i$ に属する人間は，すべて同じ部族に属する．集合 $g_i$ に属する人間もすべて同じ部族だが，彼らは $f_i$ に属する人間とは逆の部族である．$g_i$ は空集合になることもある．

$i$ が $j$ に等しくなければ，$f_i$ や $g_i$ に属する人間と $f_j$ や $g_j$ に属する人間の間の関係に関する情報は何もない．

$f_i$ と $g_i$ の対をペア $c_i$ と呼ぶことにしよう．1 人の人間は，1 つのペア，しかもその一方の集合にしか属さない．

$x$ と $y$ の間に関係があると分かった場合は，両者が属するペアを 1 つにまとめる操作を行う．たとえば，$f_i$ に属する $x$ と，$g_j$ に属する $y$ の間に，$x=y$ の関係があると分かった場合は，$f_i$ と $g_j$，$g_i$ と $f_j$ のそれぞれを合併して，$F \leftarrow f_i \cup g_j$，$G \leftarrow g_i \cup f_j$ とすればよい．ペア $c_i$ と $c_j$ はデータ構造から消して，その代わりに新しいペア $(F, G)$ を追加することになる．$x\neq y$ の場合も同様で，いずれにしろ新しい関係が分かった時点でペアとペアとの合併（2 組の集合同士の合併）が起こる．

質問と答の入力が終わった時点で，人間のグループ分けが確定する．しかし，各ペアについて $f_i$ と $g_i$ の 2 つの部族に分かれることが分かっても，$f_i$ と $g_i$ のどちらが正直か分からないので，これだけでは最終的な問題の解決には到達できない．

どの人間が正直かを決めるために最終的に役立つ情報は，正直人間の人数 $p_1$ と嘘つき人間の人数 $p_2$ である．たとえば，グループ分けの結果が次のようだったとする．

$(f_1$ (2人), $g_1$ (1人))
$(f_2$ (3人), $g_2$ (1人))

さらに，正直人間が 4 人，嘘つき人間が 3 人と分かっていたとする．

$f_1$ と $g_1$ のいずれか一方，また $f_2$ と $g_2$ のいずれか一方は正直である．たとえば，$f_1$ と $f_2$ が正直だと仮定すると，正直人間の人数が 5 になって，数が合わない．ほかもすべて試してみると，$f_1$ と $g_2$ が正直なら 3 人，$g_1$ と $f_2$ なら 4 人，$g_1$ と $g_2$ なら 2 人が正直人間ということになる．この中で，前提として与えられた 4 人が正直という事実に合致するのは，$g_1$ と $f_2$ が正直だとした場合だけである．このことから，$g_1$ に属する 1 人と $f_2$ に属する 3 人が正直だと決めてよいことが分かる．

グループ分けが全部終わった時点で，$m$ 個のペアが残ったとすると，左右の集合のいずれかに正直を割り当てる組合せの数は $2^m$ になる．原理的には，これらの組合せをすべて試してみれば，解答に到達できる．これが第 2 の推論過程になる．

ただし，単純にこの手続きを実行に移したのでは時間がかかりすぎる．それをいかに避けるかがアルゴリズム上のポイントである．

一般には，複数の割当てが，正直人間の数と合致してしまうことも起こる．これでは，どちらの割当てが真の解なのか分からない．このような場合は，分からない，つまり no という答を直ちに返して終わりにして差し支えない．

## ■第 1 の推論（グループ分け）の実現法

質問の答から人間をグループに分けていく操作をどのように実現したらよいだろうか．

C++ や Java の API を上手に使えば，集合に関する操作は簡単に実現できる．このような言語の場合は，上に述べた手続きをそのまま書き下すだけでよいだろう．

初めに，$p_1+p_2$ 人の人間をそれぞれ自分 1 人だけからなるペアに入れる．その後で，$x=y$ または $x\neq y$ の関係が入力されるたびに，ペアとペアの合併を行っていく．疑似プログラム風に書き下すと次のようになる．

```
for (i = 1; i <= p1+p2; i++) {
  f_i ← {i};   /* i 番の人だけからなる集合 */
  g_i ← 空集合;
}
for ( 入力された関係ごとに ) {
  x と y がそれぞれ属するペアを探す (x が属する
  ペアを c_i, y が属するペアを c_j とする ).
  if (i !=j) {
    x ∈ g_i なら f_i と g_i を入れ替える.
    y ∈ g_j なら f_j と g_j を入れ替える.
    関係が no, つまり x≠y なら f_j と g_j を入れ
    替える (yes, つまり x=y なら何もしない).
    f_i ← f_i ∪ f_j;
    g_i ← g_i ∪ g_j;
    データ構造からペア (f_j, g_j) を削除する.
  }
}
```

$x$ が集合 $g_i$ の方に属している場合は，$f_i$ と $g_i$ を入れ替える．これによって，合併の直前には，必ず $x$ が $f_i$ の方に属しているように正規化している．この問題の場合，$f_i$ と $g_i$ が対になっていることだけが重要で，特定の人間が属している集合が $f_i$ であろうと $g_i$ であろうとかまわないことに注意してほしい．$y$ についても同様である．$y$ が属するペアについては，さらに関係が $x \neq y$ の場合に $f_j$ と $g_j$ の入れ替えを行う．これは，$x \neq y$ の場合，$x$ が属する $f_i$ と合併するのは，$y$ が属する $f_j$ と反対の $g_j$ の方だからである．

プログラムの動きを具体例で追ってみよう．入力が次のとおりだったとする．

```
5 4 3
1 2 yes
1 3 no
4 5 yes
5 6 yes
6 7 no
```

正直人間 4 人，嘘つき人間 3 人の合計 7 人である．質問は 5 つある．

初期状態では，({1},{}) から ({7},{}) までの 7 つのペアが存在する．ここから次の順序で集合の合併操作が進む．

(1) 1 2 yes：({1},{}) と ({2},{}) を合併して ({1,2},{}) にする．
(2) 1 3 no：({1,2},{}) と ({3},{}) を逆向きに合併して ({1,2},{3}) にする．
(3) 4 5 yes：({4},{}) と ({5},{}) を合併して ({4,5},{}) にする．
(4) 5 6 yes：({4,5},{}) と ({6},{}) を合併して ({4,5,6},{}) にする．
(5) 6 7 no：({4,5,6},{}) と ({7},{}) を逆向きに合併して ({4,5,6},{7}) にする．

最終的には，2 つのペア ({1,2},{3}) と ({4,5,6},{7}) が残る．

上の疑似プログラムを実際のプログラムに書き直すことはさして難しくないと思われる．ここでは省略することにしよう．

## ■再帰呼出しによる方法

少し違う考え方として，次のようなプログラムで，グループ分けを実現することもできる．

```
void make_groups(void)
{
  int i;

  for (i = 1; i <= p1+p2; i++)
    belong_to[i] = 0;
  n_pair = 0;
  for (i = 1; i <= p1+p2; i++)
    if (belong_to[i] == 0) {
      n_pair++;
      mark(i, n_pair);
    }
}

void mark(int i, int g)
{
  int j;

  belong_to[i] = g;
  for (j = 1; j <= p1+p2; j++)
    if (relation[i][j] != 0)
      if (belong_to[j] == 0)
        mark(j, g*relation[i][j]);
}
```

今度は，関係の入力がすべて終わってから，一挙にグループ分けを行うという想定である．入力の結果は，配列 relation に入っていると仮定している．$x$ が $y$ を yes と言った場合は，relation[$x$][$y$] と relation[$y$][$x$] を +1 にしてある．no と言った場合は −1，何も言っていない場合は 0 である．

上のプログラムは，1 人の人間を出発点として，それと直接，間接に関係を定められる人間を全部集めてくるという考え方に基づいている．$p_1+p_2$ 人の人間すべてについて，$i$ 番の人間と yes または no の関

係があれば，その人間を $i$ 番と同じペアのどちらかの集合に入れる．プログラムでは，この人間の番号を $j$ として，belong_to[$j$] にペアの番号を入れることによって，この操作を表現している．すなわち，$i$ 番の人間のペア番号を $k$ とすると，$i$ 番と同じ部族の人間の belong_to には $+k$ が残り，逆の部族の人間の belong_to には $-k$ が残るようにしている．

関数 mark は，$i$ 番の人間の belong_to に値 $g$ を入れる．$g$ の値は負の場合もある．$i$ と直接関係する人間には再び mark を呼び出してペアに組み入れる．したがって，関数 mark の呼出しは再帰的に行われる．

前の章と同じ例であれば，配列 belong_to に残る値は次のとおりである．

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| +1 | +1 | −1 | +2 | +2 | +2 | −2 |

これから 1 番と 2 番が同じ部族，3 番がそれと逆の部族であることが分かる．ペアで表現すると，({1,2},{3}) である．当然ながら，前章のプログラムで求めた結果と一致する．

## ■ union-find 問題との関係

グループ分けの問題は，有名な union-find 問題の変種であることに気づいた読者も多いと思う．前章と前々章で述べた方法は，いずれも分かりやすいが，計算量の観点から最善のものではない．union-find 問題には非常に高速な解法が知られているので，それを利用すれば，計算量を極限まで小さくすることができる．本題からはそれるが，union-find 問題とその解法について簡単に解説しておこう．

union-find 問題は，嘘つき島問題で答に yes しかない場合と同様である．$n$ 個のものがあって，それらがいくつかにグループ分けされている．この状況で，$x$ 番と $y$ 番が同じグループに属するかという質問（find）と $x$ 番の属するグループと y 番の属するグループを 1 つにまとめる操作（union）の 2 つを処理できるようにすればよい．

この問題の高速解法は，グループを木で表現するものである．普通，データ構造として木を使うときは，親から子を指すポインタを用意するものだが，ここでは逆に子の側に親を指すポインタを持たせる．

あるものが属するグループは，それが属する木の根で表現する．各ノードから親を指すポインタをたどることによって，木を遡っていく．それ以上遡れないノードに達したら，それが木の根である．find 操作は，2 つのものそれぞれの属する木の根を求め，それらを比較するだけで実現できる．根同士が同じなら同じグループ，違えば違うグループである．

union 操作は，同様にして 2 つの根を求めた上で，一方を他方の親にするだけである．子になった方の木全体がもう一方の木の一部になるので，全体を 1 つのグループにまとめたことになる．

この手法に，高速化のための工夫をいくつか追加する．その 1 つは，木の根を求めるために木を遡った後で，途中のノードすべての直接の親を木の根に変えてしまうことである．この解法で使う木は，1 つのグループが 1 つの木に対応していることだけが重要で，ノードとノードの親子関係には意味がない．そこで，木の根に達する操作を速くするために，各ノードを極力木の根の直接の子供にする．高速化のための工夫その 2 は，union 操作でどちらの木を親にするかの決め方である．明らかにノード数の多い方の木を親にした方がよい．

これらの高速化をすると，find や union 操作の 1 回あたりの平均計算量は $O(\log^* n)$ にまで小さくできる．ここで，$\log^* n$ は，$n \to \log_2 n \to \log_2(\log_2 n) \to \log_2(\log_2(\log_2 n)) \to \ldots$ と計算していって，値が 1 以下になるまでのステップ数である．この関数は，$n$ を大きくしていけば，限りなく大きくなるのだが，その増え方がきわめてゆっくりとしている．たとえば，$\log^* n$ が 5 を超えるのは，$n$ が $2^{65536}$ より大きいときである．こんなにたくさんのデータを扱うことは実際上考えられないので，$O(\log^* n)$ は $O(1)$ に等しいと考えて差し支えない．

union-find 問題は，その解法（特に逆向きの木を使うデータ構造）が特異で，計算量もほかの問題には例を見ない変わったものである．アルゴリズムに関する基礎知識として身に付けて損はない．

しかし，ここでうっちゃりを食わすのだが，嘘つき島の問題の場合，union-find 問題の高速解法の適用は考えない方が無難だろう．グループ分けに要する計算時間が問題になることはないからである．前章に述べた単純な方法でも，グループ分けは一瞬で終わる．高速なアルゴリズムを一生懸命実現するよりも，単純な方法を手早くプログラミングする方が得策である．

この問題で計算量を心配しなければいけないのは，次の第 2 の推論の方なのである．

## ■第2の推論の単純な実現法 ―再帰的探索

第2の推論は，各ペアの2つの集合のどちらを正直人間に割り当てれば正直人間の数が $p_1$ に等しくなるか調べることである．

単純な方法としては，バックトラック法による再帰的な探索（しらみつぶし）が考えられる．関数 backtrack で $i$ 番目のペアについての割当て方を決める．$f_i$ を正直とする場合と，$g_i$ を正直とする場合の2つである．それぞれの場合について，backtrack を再帰的に呼び出して $i$+1 番目のペア以降の割当て方を調べるようにすればよい．

この方法で問題が解けることは間違いない．しかし，問題は計算時間である．問題の分析の項で述べたとおり，このやり方の計算量は，ペアの総数を $m$ としたとき，$O(2^m)$ になる．バックトラックの $m$ 個の段階それぞれで，$f_i$ と $g_i$ の2つの選択肢を調べるからである．

この問題で，ペアの数が最も多くなるのは，質問が1つもなかった場合である．$p_1$+$p_2$ 人の人間それぞれが自分1人だけからなるペアを作るので，ペアの総数 $m$ は $p_1$+$p_2$ に等しくなる．この数は最大600（正確には598）に達することがある．したがって，再帰的探索では，最悪の場合の計算の量として $2^{600}$ 程度を覚悟する必要がある．これは膨大な計算時間であり，到底耐えられない．

一般に，再帰的探索は枝刈りによる効率改善が可能な場合が多い．途中での打切りを加えることによって，探索する状態数を大幅に減らせる場合である．この問題の場合は，それ以降各ペアのどちらの集合を選んでも目標の人数に達しない状態，どちらの集合を選んでも目標の人数を超えてしまう状態などを検出して，途中で打ち切るようにすれば，計算時間を短くできる．これは一種の分枝限定法である．しかし，このような高速化を頑張ってみても，すぐ答が返るようなスピードにはなかなかならないようである．

## ■第2の推論の高速な実現法 ―動的計画法

再帰的な探索では遅すぎる，枝刈りも思うにまかせないとなれば，動的計画法の適用を考えるのが順序である．動的計画法（dynamic programming）は，連載第3回（2002年6月号）でも紹介されているが，要するに，計算の途中結果を入れる表を用意して，これをある組織的な順序で埋めていく手法である．

この問題の場合は，ある人数に等しくなるような集合の選び方が何通りあるかを求めることがポイントである．たとえば，3つのペアがあって，それぞれの2つの集合の人数が (1, 2)，(1, 3)，(1, 4) だったとしよう．ちょうど3人や9人になるような集合の選び方は1通りしかない．3人なら1+1+1だけ，9人なら2+3+4だけである．しかし，6人なら2通りの集合の選び方がある．1+1+4でも6になるし，2+3+1でも6になる．

このような集合の選び方の数をあらゆる人数に関して求めた表を作る．そして，これを更新していく．更新というのは，ペアを1つ1つ順に調べながら，そこまでの範囲のペアだけを使った選び方の数を表に残すようにするという意味である．

上の3つのペアの例だと，表の更新の様子は次のようになる．

| 人数 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 初期状態 | 1 | - | - | - | - | - | - | - | - | - |
| (1,2) まで入れると | - | 1 | 1 | - | - | - | - | - | - | - |
| (1,3) まで入れると | - | - | 1 | 1 | 1 | 1 | - | - | - | - |
| (1,4) まで入れると | - | - | - | 1 | 1 | 1 | 2 | 1 | 1 | 1 |

この図の横1列が，ある時点での表の内容である．見やすくするために，0が入るべき欄はハイフンで表した．

$k$ 番目のペアを調べるときの表の更新操作は簡単である．$k$−1 番までのペアについて求めた表を old，$k$ 番を入れてできる新しい表を new とする．それぞれ人数を表す添字をとる配列である．集合 $f_k$ の要素数を fcount[$k$]，$g_k$ の要素数を gcount[$k$] で表すことにする．

```
for (i = 0; i <= p1+p2; i++)
  new[i] = 0;
for (i = 0; i <= p1+p2; i++) {
  if (old[i] == 0)
    continue;
  new[i+fcount[k]] += old[i];
  new[i+gcount[k]] += old[i];
}
```

これで論理的には正しいのだが，現実の計算機で計算するには，もう少し配慮が必要である．このままでは，簡単にオーバーフローが起こるからである．表の中の数の合計は最大 $2^{600}$ になる．1つ1つの数も非常に大きくなることは容易に想像できるだろう．

この問題の場合，値を表現する組合せが2通り以上存在するなら，何通りであるかを正確に知る必要はない．何通りであろうと，1通りに定めることが不可能であるという結論に変わりはないからである．いずれにしろ，誰が正直人間か正確には分からず，noと出力して終わりにすべきケースに当たる．

このことを利用して，表の中に入れる数を2以下に限るようにする．表の中に現れる値は，0，1，2の3種類ということになる．2通り以上の組合せが存在する場合は，すべて2という値で表す．これによってオーバーフローの発生を避けることができる．

この表現法を採用して，動的計画法の部分を最初から書き下すと，次のようになる．

```
table[0][0] = 1;
for (i = 1; i <= p1+p2; i++)
  table[0][i] = 0;
for (k = 1; k <= n_pair; k++) {
  for (i = 0; i <= p1+p2; i++) {
    table[k][i] = 0;
    which[k][i] = 0;
  }
  for (i = 0; i <= p1+p2; i++) {
    if (table[k-1][i] == 0)
      continue;
    j = i+fcount[k];
    table[k][j] += table[k-1][i];
    if (table[k][j] > 2)
      table[k][j] = 2;
    which[k][j] = +k;
    j = i+gcount[k];
    table[k][j] += table[k-1][i];
    if (table[k][j] > 2)
      table[k][j] = 2;
    which[k][j] = -k;
  }
}
```

ペアの番号も人間の番号も1から始まることにしている．0は何もないことを表す特別の添字である．

今度は，組合せの数を入れる配列tableを2次元にしている．第1の添字は何番目のペアまで調べたかを表す．配列whichには，2つの集合のどちらを選んだかを記録する．これは，解が見つかった後で，正直人間の番号をプリントするために使う．

上の計算が終わったら，配列要素table[$m$][$p_1$]を調べる（プログラム中ではn_pairという変数名で$m$を表している）．これが1なら，whichを調べて，各ペアの該当する側の集合から正直人間の番号を集めて出力すればよい．table[$m$][$p_1$]の値が2なら，noと出力するまでのことである．table[$m$][$p_1$]の値が0になることはない（入力データに矛盾はない）と保証されている．

プログラムから明らかなように，この方法の計算量は$O(m^2)$である．最大でも$600^2$にしかならない．現在の計算機なら，まったく問題にならない計算の量である．動的計画法を採用すれば，第2の推論もアッと言う間に終わる．

## ■別解の追求

動的計画法を使えば第2の推論を実現できることは分かった．しかし，これ以外の方法ではなかなかうまくいかない．つまり，普通のプログラミング技法の中では，動的計画法がほとんど唯一の解法である．

1種類の解き方しか成立しないのでは面白くない．そこで，何とか実用的な時間で解ける別解を見つけてやろうと試みた．

手がかりは，各ペアの2つの集合の人数差にある．まず，どれか1つのペアの2つの集合が同一の人数であれば，直ちにnoと答えてよいことが分かる．どちらの集合を選んでも同じ人数になるから，どちらを選べばよいか決められない．

また，2つの集合の人数差の同じペアが2つ以上ある場合，それらは全部1つにまとめて考えてよいことが分かる．例で説明しよう．3つのペアがあって，それらの人数が(1, 3)，(2, 4)，(5, 7)だったとする．3つとも，2つの集合の人数差は2で同じである．また，人数の少ない方の集合を左に書くことに統一してある．

このとき，ペアの左からも右からも選んで作る人数，たとえば1+4+5（左，右，左）が解だったとすると，この解以外にも解がある．左から選んだペアと右から選んだペアを1つずつ見つけ，それぞれ反対側の集合を選ぶことにすれば，人数が同じになるからである．例の場合は，3+2+5（右，左，左）という選び方で同じ人数が得られる．

つまり，このような場合に唯一の解が存在するとすれば，全ペアとも左から選ぶか，全ペアとも右から選ぶか，2通りしかない．1+2+5か3+4+7かのいずれかである．

このような考察から，3つのペアを1つにまとめて，(1+2+5, 3+4+7)=(8, 14)にしてしまうことが考えられる．こうすれば，ペアの総数が600になることはなくなる．人数差が同じペアは全部1つにまとまるので，最もペアの数が多くなるのは差が1のペア，2のペア，3のペア，...が1つずつの場合である．1+2+3+...+$m$が600を超えるのは$m$が35のときだから，

最大でも35個のペアということになる．$2^{35}$ではまだちょっと厳しいが，それでも$2^{600}$よりはだいぶましであろう．再帰的な探索でも何とかなるかもしれない．

ところが，実はこの方法はうまくいかないのである．正直人間の決め方が1通りしかないのであれば，確かに正しい解が得られる．しかし，複数の決め方が可能な場合に，1通りしかないと誤った答を導く可能性がある．たとえば，(0, 2)，(0, 2)，(0, 2)，(0, 4)の4つのペアがあって，正直人間を4人にする場合を考える．明らかに2通り以上の割当て方が存在する例である．ところが，差が同数のペアを1つにまとめると，(0, 6)と(0, 4)になって，4人にする組合せが1通りに決まってしまう．これは誤りである．

この例の場合は，差が2のペアを1つにまとめるときに，その倍数の差4のペアも一緒にすれば正しい解（no）が得られる．しかし，一般の場合に，どのようなペアを1つにまとめればよいか，簡単には決められない．たとえば，(0, 2)，(0, 2)，(0, 3)，(0, 3)，(0, 5)の場合に，差が2のペア2つ，差が3のペア2つをそれぞれ1つにまとめると，0+0+5と2+3+0が同じであるという事実が隠れてしまう．

結局，差が同数のグループを1つにまとめるという方法は定式化が難しいし，たとえ定式化できたとしても，元の問題よりむしろ難しくなっているようである．この方針は諦めざるを得ない．

## ■実現可能な別解

差が同数のペアを1つにまとめるのは駄目だと分かった．しかし，2つの集合の人数差が同じであるペアに注目して高速化を図るという考え方が間違っているわけではない．前章における観察を再帰的探索で利用すれば，実用的な時間で解を求められる別解に到達することができる．

最初に述べた再帰的探索では，探索の1つのレベルが1つのペアに対応していた．各レベルで，該当するペアが持つ2つの集合のどちらを選ぶか決めた．今度は，再帰的探索の1つのレベルを人数差が同じペアの集まりに対応させる．

たとえば，人数差$s$人のペアが$t$個あったとする．これらのペアを組み合わせてできる人数は，$t$個のうち何個のペアについて人数の少ない方の集合を選ぶかだけで決まる．選択肢としてあり得るのは，人数の少ない方の集合を選ぶペアの個数を0，1，2，…，$t$とした，合計$t$+1種類の場合だけである．再帰的探索の1つのレベルで，これら$t$+1種類の選択肢をすべて試すことにすればよい．単純な再帰的探索で$2^t$種類調べるところを$t$+1種類調べるだけで済むのだから，大幅なスピードアップになるに違いない．

この方法で解を探索して，解が見つかったときには，少し後始末が必要である．まず，2つ以上の解が見つかった場合は，noと出力して実行を打ち切る．さらに，解が1つしかなくても，それが不適当な場合もある．解が見つかったら，その解に至る探索の各レベルについて，0～$t$個のうち何個と決めたかを調べる．それが0でも$t$でもなければ，探索によって見つかった解は1つでも，元の問題の解は複数存在する．理由は前章で説明したとおりである．この場合も，2つ以上の解が見つかったときと同様に，noと出力して，実行を打ち切ってよい．

この考え方に従ってプログラムを書いたところ，10秒くらいでコンテスト本番の入力データを処理できた．筆者の少し古い計算機を使ってのことである．さらに，分枝限定法を適用して，見込みのない探索を打ち切るようにしたところ，それこそアッと言う間に答が求まるようになった．

前章で述べたとおり，最悪の場合は$2^{35}$程度の計算の量になる恐れがあるが，いろいろな理由による探索途中の打切りが可能なので，そこまで遅くなることはめったにないようである．実際にどの程度遅くなることがあるのか，正確なことは分からない．

以上のとおり，別解を見つけることはできた．しかし，振り返ってみると，このやり方は決してスマートでないし，プログラミングが楽なわけでもない．計算時間の保証もないので，動的計画法より優れているとはとても言えない．やはり，この問題の解法としては動的計画法が決定版であろう．

（平成15年4月9日受付）