Program Promenade

連載

倍数の和による整数の表現

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

プログラミングコンテストでは，やさしい問題から難しい問題まで，さまざまな難易度の問題を取り揃えて出題する．これは，参加チームのプログラミング能力に大きな差があることに対する対策である．弱いチームにも1問は解いてもらいたい（1問も解けずに落胆させることは避けたい）し，逆に強いチームにも手ごたえを感じてもらいたいと考えると，難易度にバラエティを持たせることは必然だと思われる．

審判から見ると，難易度に変化をつけることによって，チームの強弱の差がはっきりするという利点がある．やさしい問題ばかりだと皆解けてしまうし，難しい問題ばかりだと誰も解けないという結果になる可能性が大きい．このいずれの場合でも，成績に有意な差は認めにくくなる．難易度に変化があれば，どの問題まで解けたかで，強弱の差がはっきり現れるだろう．

逆に，参加チームから見た場合，どの問題がやさしくて，どの問題が難しいかを見分けることがまず重要になる．そして，やさしい問題を手早く片付けることに努めるべきである．勝敗の分かれ目は，難しい問題を解けるかどうかにあることが多いが，同じ問題数の場合は，問題を解くまでの時間の短い方が勝ちだからである．

この連載では，今まで難しい問題をいかに解くかということに重点を置いてきた．プログラムの書き方に関する解説なのだから，これは当然である．誰にでも解けるような問題では，解説する甲斐がない．しかし，やさしい問題の中にも議論すべきテーマを含んでいるものがある．今回は，そのようなやさしい問題の1つを取り上げることにした．1999年の国内予選問題B「Unable Count」である．

この問題は，どうやっても解けるといえるほどやさしいのだが，問題を解くに至る道筋が一通りでない．さまざまな方向からのアプローチが可能で，解法のバラエティが豊かである．いろいろなプログラミング技法を比較検討するプロムナードの題材としてピッタリだと考えた．

■問題：倍数の和で表現できない整数の個数

3つの整数 n, a, b が与えられている．いずれも，1以上100万以下である．1以上 n 以下の整数のうち，$a \times i + b \times j$ の形で表せないものの個数（これをUnable Countと呼ぶことにする）を求めよ．ここで，i も j も0以上の整数とする．

たとえば，a=2, b=5だったとする．1～10の範囲の整数の表現法を調べてみると，次のようになる．

2=2×1+5×0
4=2×2+5×0
5=2×0+5×1
6=2×3+5×0
7=2×1+5×1
8=2×4+5×0
9=2×2+5×1
10=2×5+5×0=2×0+5×2
.....

5以上の数がすべて $2 \times i + 5 \times j$ の形で表せることはすぐに分かるだろう．奇数かつ5以上なら $2 \times i + 5 \times 1$ と書けるし，偶数なら $2 \times i + 5 \times 0$ と書けるからである．結論として，この形で表せない正の数は1と3の2つだけである．$n \geqq 3$ ならば，この問題の答は2とすればよい．n=1または2なら，1が答である．

a と b の最大公約数を $\gcd(a, b)$ と書くことにすると，$a \times i + b \times j$ は必ず $\gcd(a, b)$ の倍数になる．つまり，$\gcd(a, b)$ の倍数以外の数は，初めから表せないことが分かっている．プログラムの中で，この事実を利用

するかどうかは自由だが，いちおう頭の隅にとどめておくとよい．

以下のプログラムでは，変数 n, a, b に 3 つの値が入力済みだと仮定する．さらに，a と b は交換しても結果が変わらないはずなので，a≦b になるよう，次のような前処理を加えておくことにする．

```
if (a > b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

a≦b であることを利用すると，ある種のプログラムの計算時間を短くできることがある．

Unable Count を求めるプログラムは，

```
int unable_count(void);
```

で定義される関数 unable_count の形に書くことにする．この関数の返り値が求める Unable Count の値である．

■ $O(n^2)$ の単純な解法

最初に，参加チームが提出したプログラムのほとんどに採用されていた単純明快な方法を示す．1 〜 n の数それぞれについて，$a\times i+b\times j$ の形で表せるかどうか調べるという方法である．

この方法によるプログラムの例を示す．

```
int unable_count(void)
{
  int v, x, count;

  count = n;
  for (v = 1; v <= n; v++) {
    x = v;
    while (x >= 0) {
      if (x%a == 0) {
        count--;
        break;
      }
      x -= b;
    }
  }
  return (count);
}
```

調べる対象の数を v としている．j を増やしながら，$v-b\times j$ が a で割り切れるかどうか順次調べる．割り切れれば，v が $a\times i+b\times j$ の形で表せることが分かるので，v に関する処理を打ち切って，次の数の処理に移る．プログラムでは，変数 x の初期値を v として，これから b を繰り返し引くことによって，$v-b\times j$ を得るようにしている．

こんな簡単な方法でよければ何の苦労もないのだが，そうは問屋がおろさない．この方法の計算量は $O(n^2)$ になる．内側の while 文の繰り返しは，最悪の場合 v/b 回程度である．これを $1\leqq v\leqq n$ の範囲で繰り返すから，全体の計算量は n の二乗を b で割ったものに比例する．具体的に計算時間が最もかかるデータとしては，$n=10^6$, $a=b=2$ の組合せがある．このデータに対して，上のようなプログラムを走らせたところ，筆者の計算機で結果が出るまでに 7 時間ほどかかった．

インターネット経由で開催される予選の問題なので，プログラムの実行時間に制限はない（大会の場合は 3 分を上限とすることが普通）．しかし，競技開始から終了まで 3 時間半しかないのだから，7 時間もかかるプログラムでは，答が出る前にコンテストが終わってしまう．失格間違いなしである．

ところが，審判団はこのような単純なプログラムが多く提出されるとは予想していなかった．このプログラムが苦手とする上記のようなデータを用意していなかったので，単純なプログラムでも合格になってしまった．

これは審判団の一員として残念なことだったと考えている．確かに，$a=b=2$ のような（問題の本質から考えればおかしな）データを用意しようとは普通なら考えないだろう．しかし，審判の重要な仕事の 1 つは，どの程度遅いプログラムまで許すか決めることである．そして，その限度以上に遅いプログラムでは時間がかかりすぎて失格になるようなデータを用意しなければならない．この問題は，やさしい部類に属するので，このような配慮が必要だとは考えていなかったが，結果として手抜かりだった．

■ Hamming の問題

出題者は，Hamming の問題に対する有名な解法を流用すれば，Unable Count の問題も解けると考えていた．ちょっと横道にそれるが，この問題とその解法を紹介しておこう．

Hamming の問題とは，次のようなものである．

- $2^i \times 3^j \times 5^k$ の形の正の整数を小さい方から順に 700 個生成せよ．

別の言い方をすると，2, 3, 5 以外の素因数を持たない整数の生成である．最初のいくつかを示すと，1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, 36, 40, ... となる．

最初のうちは，$2^i \times 3^j \times 5^k$ の形で表せる数の密度が高いが，しだいにまばらになる．たとえば，696 番目の数 5668704 と 697 番目の数 5760000 の間には，この形で表せない数が 91295 個もある．1 から順に整数を生成し，それぞれがこの形で表せるかどうか調べるという単純な方法では，計算時間がかかりすぎる．2, 3, 5 以外の素因数を持たない整数だけを無駄なく生成する方法が必要とされる．

この問題には効率的な解法が存在する．他の問題にはあまり現れない変わった形のプログラムになるので，非常に印象的で一度聞いたら忘れられない．しかし，文献に取り上げられることがあまり多くなかったようで，知る人ぞ知るの状態になっているのは不思議なことである．

この解法を C の関数として書くと，次のようになる．

```
void hamming(int t[700])
{
  int p, q, r, s, x;

  p = 0;
  q = 0;
  r = 0;
  t[0] = 1;
  s = 1;
  while (s < 700) {
    x = 2*t[p];
    if (3*t[q] < x)
      x = 3*t[q];
    if (5*t[r] < x)
      x = 5*t[r];
    t[s] = x;
    s++;
    if (x == 2*t[p])
      p++;
    if (x == 3*t[q])
      q++;
    if (x == 5*t[r])
      r++;
  }
}
```

このプログラムは，マージの手法を応用している．それまでに生成された数それぞれを 2 倍，3 倍，5 倍した 3 本の列があると想定して，これらをマージする．生成済みの数は $2^i \times 3^j \times 5^k$ の形で表せるものばかりなので，それを 2 倍，3 倍，5 倍したものも同じ形で表せることは間違いない．また，この 3 種類で生成すべき数すべてを尽くしていることもすぐに分かる．

実際には，3 本の列を別に用意する必要はなく，元の列の各要素を 2 倍，3 倍，5 倍したものを仮想的な列と考えればよい．マージは，それぞれの列の中に現在位置（2 倍の列が p, 3 倍の列が q, 5 倍の列が r）を設け，現在位置にある数の中で最も小さいものを出力に移すという標準的な方法を採っている．

■ Hamming の方法の応用による解法

Unable Count の問題が Hamming の問題と同じ構造をしていることは容易に理解できると思う．Hamming の問題のべき乗を掛け算に変え，掛け算を足し算に変えると Unable Count の問題になる．一方は表せる数の列挙，もう一方は表せない数の個数の計算という違いはあるが，この違いは重要なものではない．当然，Hamming の問題の解法を流用して，Unable Count の問題を解くことが可能である．

プログラムは次のようなものになるだろう．

```
int unable_count(void)
{
  int p, q, s, x;
  int t[1000001];

  p = 0;
  q = 0;
  t[0] = 0;
  s = 1;
  for (;;) {
    if (t[p]+a <= t[q]+b)
      x = t[p]+a;
    else
      x = t[q]+b;
    if (x > n)
      break;
    t[s] = x;
    s++;
    if (x == t[p]+a)
      p++;
    if (x == t[q]+b)
      q++;
  }
  return (n-s+1);
}
```

Hammingの問題の場合と同じく，t[...]+aとt[...]+bが2本の仮想的な列に当たる．この2本の列をマージして，$a\times i+b\times j$の形で表現できる数を順次生成していく．添字pとqがそれぞれの列の現在位置である．生成できた数は，配列tに順に入れる．最終的な結果はnから生成できた数の個数（s−1, 1を引くのは0が勘定に入らないから）を引いたものである．

この方法の計算量は$O(n)$である．while文の繰り返しは，最悪の場合でも$n+1$回にしかならない．while文の中には繰り返しが含まれていないので，$O(n)$で確実に終わる．

この方法を改良した解法も考えられる．たとえば，連続したa個の整数（gcd(a, b)の倍数のa個の連続）が生成できれば，そこから先すべての整数が生成できることは間違いない．この判定条件を加えれば，while文の実行回数を少なくすることができるだろう．しかし，これはプログラムを複雑にするばかりで，実行時間の得は大きくない．ここでは，これ以上詳しく調べることはしない．

後から反省してみると，Unable Countの問題の解法として，ここに示したような難しいものを想定したのは，少し考えすぎであった．Unable Countは，Hammingの問題よりずっとやさしいので，これほど高級な考え方を採用する必要はないのである．もっと単純で，しかも同じくらい速い解法を次に示す．

■マーク付けによる解法

Unable Countの問題は，素数の表を作るための「Eratosthenesのふるい」と同じような考え方でも解ける．数1つにつき1ビットのメモリを用意する．そして，$a\times i+b\times j$の形で表現できる数それぞれにマーク（印）を付けていく．該当する数すべてにマークを付け終わってから，1 〜 nの範囲でマークの付いていない数の個数を数えればよい．

これは，非常に素直な考え方で，プログラミングも簡単である．しかも，この方法でも$O(n)$の計算量が得られる．Hammingの解法の応用が大げさだと言ったのは，このような事情があるからである．

マークは，次のような順番で付けていく．

0	a	$2a$	$3a$	$4a$	.....
b	$b+a$	$b+2a$	$b+3a$	$b+4a$	.....
$2b$	$2b+a$	$2b+2a$	$2b+3a$	$2b+4a$	.....
$3b$	$3b+a$	$3b+2a$	$3b+3a$	$3b+4a$	.....
.....					
$(a\text{-}1)b$	$(a\text{-}1)b+a$	$(a\text{-}1)b+2a$	$(a\text{-}1)b+3a$	$(a\text{-}1)b+4a$	.....

横方向の繰り返しは，nに達するまで続ける．一方，縦方向の繰り返しは$b\times j$について$0\leqq j\leqq a-1$の範囲だけでよい．たとえば，$j=a$の列を考えると，先頭の要素は$a\times b$だが，これは最初の列（0が先頭）の要素としてマーク済みだからである．

たとえば，a=3, b=4の場合のマークの付け方は次のようになる．

0	3	6	9	12	15	18	...
4	7	10	13	16	19	22	...
8	11	14	17	20	23	26	...

1, 2, 5の3つがマークされずに残るので，Unable Countの値は3だと分かる（$n\geqq 5$であれば）．

この方法によるプログラムの例は次のとおりである．

```
int unable_count(void)
{
  int i, j, x, count;
  int table[1000001];

  for (i = 0; i <= n; i++)
    table[i] = 0;
  for (j = 0; j < a; j++) {
    x = b*j;
    if (x > n)
      break;
    while (x <= n) {
      table[x] = 1;
      x += a;
    }
  }
  count = 0;
  for (i = 1; i <= n; i++)
    if (table[i] == 0)
      count++;
  return (count);
}
```

プログラムの中ほどにある2行，

```
if (x > n)
  break;
```

は，論理的には不要である．x が n より大きいなら，次の while 文を 1 回も実行しないことになるので，無理に途中で脱出しなくても，結果は同じになるはずである．しかし，現実にはオーバーフローという問題がある．j が大きくなると，b*j がオーバーフローする可能性が生じる．こうなると結果がどうなるか分からなくなる．そうなる前，b*j が n より大きくなった時点で繰り返しを打ち切ってしまえば，このような問題を避けられる．

この方法の計算量は $O(n)$ である．上に示した図の横 1 列のマークに $O(n/a)$ の計算量がかかる．これを a 回繰り返せばよいのだから，$O(n/a)\times a$ で $O(n)$ となる．

このような簡単なプログラムで，十分な速度を持つプログラムが書けることが分かったので，出題者の意図は大きくくじかれたが，実はこれで終わりではなかった．さらに簡単で，しかも速いプログラムが可能である．この方法について次に述べる．

■マーク付けの方法の改良

前ページのプログラムでは，縦方向の繰り返しの範囲を $0\leqq j\leqq a-1$ としているが，これには無駄がある．$\gcd(a, b)$ が 1 でないならば，$0\leqq j\leqq a/\gcd(a, b)-1$ の範囲で十分である．

たとえば，$a=6, b=10$ のケースでは，

0	6	12	18	24	30	36	...
10	16	22	28	34	40	46	...
20	26	32	38	44	50	56	...

の 3 列だけマークすればよい．次の 30 から始まる系列は，0 から始まる系列でマーク済みだから，改めてマークしなくてよい．この場合，$a=6, b=10$ だから，$\gcd(a, b)=2$ である．j の範囲は，$0\leqq j\leqq a/\gcd(a, b)-1=6/2-1=2$ とした．

この考え方をプログラムに取り入れるのは簡単である．1 つ前のプログラムの for 文の上限の式を変えるだけである．改めてプログラムを示すことは控えよう．

マークの出発点となる $b\times j$ の j の範囲をこのように限定すると，同じ数が表の中の 2 カ所以上に現れることはなくなる．数学的な用語でいうと，$a\times i+b\times j=a\times p+b\times q$, $0\leqq j, q\leqq a/\gcd(a, b)-1$ になるとすれば，$i=p, j=q$ の場合しかあり得ない．このことは簡単に証明できる．

1 つの数に 1 回しかマークしないと聞いて，性能がよくなると安心してはならない．これではマークの存在価値がなくなるのである．そもそも，マークが役に立つのは，1 つの数が 2 カ所以上に現れ得る場合である．1 つの数に何回マークしようとも，最後に数えるときは 1 つにしかならないということがマーク付けの方法のポイントである．そのポイントが効果を発揮しないような状況なら，マーク付けの方法の意義そのものを疑わなければならない．

j の範囲を $0\leqq j\leqq a/\gcd(a, b)-1$ に限定すると，ある数にマークしようとしたときに，その数にすでにマークが付いていることはない．それなら，それぞれの系列で何個マークするかを計算して，それを加えるだけでマークの総数が求まる．系列ごとのマークの個数は，割算で簡単に計算できる．マークのための配列をことさら用意する必要などない．

この考え方でプログラムを書くと次のようになる．

```
int unable_count(void)
{
  int g, j, count;

  g = gcd(a, b);
  count = n+1;
  for (j = 0; j < a/g; j++) {
    if (b*j > n)
      break;
    count -= (n-b*j)/a+1;
  }
  return (count);
}
```

変数 count の初期値が $n+1$ になっているのは，0 もマークの対象になっているからである．これを勘定からはずすために 1 を加えている．

このプログラムの繰り返しの回数は $a/\gcd(a, b)$ と n/b のどちらか小さい方である．$b\times j$ が n を超えたら，そこで打ち切って差し支えないからである．最悪のケースとして，$\gcd(a, b)=1$ の場合を考えると，a と n/b のどちらか小さい方ということになる．最初に述べたとおり，一般性を失うことなく $a\leqq b$ としてかまわないので，この値が最大になるのは $a=b=\sqrt{n}$ の場合であり，その場合の値は $\sqrt{n}$ であることが分かる．したがって，このプログラムの計算量は $O(\sqrt{n})$ である．

最大公約数 $\gcd(a, b)$ を求める部分の計算量は，Euclid の互除法を使えば $O(\log n)$ になるので，$O(\sqrt{n})$ より小さく，全体の計算量には影響しない．

■数学の問題かプログラミングの問題か

最後の方法の計算量は，$O(\sqrt{n})$ である．出題者の想定していた方法より，だいぶ速くすることができた．しかし，この方法でやっている仕事の内容は，実につまらないものだといわざるを得ない．割算の答をいくつか加え合わせているだけである．プログラミングとしては，まったく面白味のないものになってしまった．

結局，この問題のポイントは数学にあったことが分かる．最後の解が目標だとすれば，そこに至るまでの数学的な考察を行えるかどうかが鍵であり，プログラミングの能力は重要でない．これはプログラミングコンテストの問題として望ましいことではない．Unable Count は，コンテスト問題として失敗作だったという評価があり得る．

一方で，学生がすぐに最後の方法に到達できるはずもなく，途中でいろいろな方法の得失を体験するはずだと考えれば，教育的に見て非常によい問題だったという評価もあり得るだろう．実際，筆者は，ここまでに示したような考察を大いに楽しんだ．

この問題をどう評価するかは見解の分かれるところだと思う．

プログラミングコンテストの問題を作る過程で，数学に力点があるか，プログラミングに力点があるかということをよく議論する．数学の部分だけが難しくて，そこを突破すれば簡単なプログラミングだけという問題はなるべく避けたいと多くの審判が思っている．しかし，これは言うは易く，行うは難しであって，なかなか思うようにはいかない．また，ここまでが数学で，ここからがプログラミングとはっきり切れるものではないし，数学的な分析を要するプログラミング問題が必ず悪いというものでもない．ほどほどにというしかないのかもしれない．

なお，Unable Count の問題に戻って，a と b が互いに素であって，かつ n が十分大きければ，

$$(a-1)\times(b-1)/2$$

という閉じた式で表せることが分かっている．n が十分大きいとは，曖昧な言い方だが，たとえば $a\times b$ より大きければ，十分大きいといってよい．

2 つの条件のうち，互いに素の方は，成立していなくても対処できる．a と b が互いに素でない場合は，最大公約数で割ってから同じような計算をすればよい．しかし，もう一方は簡単に対処できそうもない．n が小さい場合にまで適用できる一般的な式があるかどうかは不明である．n が小さい場合は，今までに示した方法のどれかを使うようなプログラムを書くしかなかろう．

n が小さい場合は別としても，最大公約数の計算に $O(\log n)$ の計算量がかかるので，閉じた式とはいっても，最後に示した方法に比べて格段に優れているとまではいえないようである．

（平成 14 年 12 月 2 日受付）