0.5 循環小数の秘密

もうしばらく小数の話題を続けてみよう。

小数には 0.333... のような**無限小数**と、0.125 のような**有限小数**がある。また、ひと口に無限小数といっても、0.333... は**循環小数**と呼ばれる数で、円周率 3.141592... のように**循環しない小数**とは区別している。実は、すべての有理数は有限小数か循環小数になる。言い換えれば、分数は必ず有限小数か循環小数にできるということで、決して循環しない無限小数にはならない。その逆に、循環しない無限小数は決して分数にすることはできない。どういうことだろうか。

たとえば 1/6 は循環する無限小数である。実際に割り算を行ってみれば一目瞭然だろう。

$$
\begin{array}{rr}
 & 0.\ 1\ 6\ 6 \\
\cline{2-2}
6\ ) & 1.\ 0\ 0\ 0 \\
 & \ \ 6 \\
\cline{2-2}
 & \ \ 4\ 0 \\
 & \ \ 3\ 6 \\
\cline{2-2}
 & \ \ \ \ 4\ 0
\end{array}
$$

割り算は途中で計算を止めているが、このあとは 6 が続くだけである。その理由は簡単だ。割り算の最後の行に余りである 40 がある。そしてこれと同じ余りが一つ前にも出ているね。そう、小数が循環する理由は、以前の余りと同じものがでるからなのだ。

分数を小数に直すときは、分子を分母で割るはずである。そのときの余りは、割る数である分母より小さい数しかありえない。具体的には、6 で割り算をすれば余りは 0, 1, 2, 3, 4, 5 の 6 種類に限られる。つまり余りの種類は、0 を含めて**高々**分母に使われた数だけしかないのだ。そのせいで余りにあたる数は、いつか必ず同じものになってしまう。一度同じ余りになれば、あとは循環するしかないし、余りが 0 になれば割り切れるということなのだから。

それでは、循環する無限小数は、はじめどんな分数だったか気にならないだろうか？　しかし、循環する無限小数をもとの分数に復元するのは簡単である。0.1666... であれば $x = 0.1666...$ とおいて

$$\begin{array}{rrl} & 100x & = \quad 16.666\ldots \\ -) & 10x & = \quad 1.666\ldots \\ \hline & 90x & = \quad 15 \end{array}$$

のようにすれば、$x = 15/90$ であることが分かるのだ。この方法はどんな循環小数にも使える。コツは循環する部分がそろうように、**適当な** 10 の倍数を掛けてやればよい。

ところで循環する無限小数のうち、ひときわ目を引くものがあるだろう。$0.999\ldots$ のことだ。これも同様に $x = 0.999\ldots$ とおいて

$$\begin{array}{rrl} & 10x & = \quad 9.999\ldots \\ -) & x & = \quad 0.999\ldots \\ \hline & 9x & = \quad 9 \end{array}$$

としてみよう。あれ？　$x = 9/9$ になったぞ。ということは $0.999\ldots = 1$ なんだろうか。その説明の前に次の問題をやってほしい。

EX.　$0.4999\ldots$ を分数にしてみよ。また、$0.6999\ldots$ を分数にしてみよ。

これで雰囲気がつかめただろうか。話をちょっと前に戻すけれど、小数の濃度を調べているときに、すべての小数を $0.\alpha_1\alpha_2\alpha_3\alpha_4\ldots$ の形に表したね。このとき、0.5 のような小数は $0.5000\ldots$ とでもするのかなと考えなかっただろうか？　実を言うと、そこには 0.5 や $0.5000\ldots$ のような、いわゆる有限小数は含まれていなかったのである。そこでは 0.5 は $0.4999\ldots$ の形で登場していたのだ。有限小数はすべて $999\ldots$ を含む、循環する無限小数になっていたのである。有限小数は無限小数で表記しておくと都合がよいのだ。そうしておけば、同じ数を 2 回数えることはなくなるから。

さて、話題は $0.999\ldots$ へ戻る。$0.999\ldots = 1$ である。なぜなら 9/9 を実際に割り算してみると、$0.999\ldots$ であることが確認できるのだから。

$$\begin{array}{rr|l} & & 0.\ 9\ 9\ 9 \\ \hline 9 &) & 9.\ 0\ 0\ 0 \\ & & 8\ 1 \\ \hline & & \quad 9\ 0 \\ & & \quad 8\ 1 \\ \hline & & \quad\ \ 9\ 0 \end{array}$$

これは何だか不思議な計算だ。でも、あっている。このようなことが起こるのは、割り切れる割り算に 2 通りの表記法があるからだ。一つは素直に割り切ってしまう計算である。そうすれば $9/9 = 1$ となる。そしてもう一つは、いまの例のように $999\ldots$ と商を立て続けてしまう計算である。こちらが濃度の話に登場した表記なのだ。

それにしても $999\ldots$ には悩まされそうだ。その底流をなすのは無限であることは間違いない。深みにはまる前に、循環小数の話へ戻ろう。

循環小数を計算してみると、$1/6 = 0.1666666\ldots$ のように**循環節**が短いものと $1/7 = 0.1428571\ldots$ のように循環節が長いものがある。1/6 は余りが最大で 5 種類出る可能性がある。1/6 は割り切れないので、0 は余りの種類に含めていない。このことは循環節が最大で 5 になる可能性があるわけだが、実際の循環節は 1 だ。ところが 1/7 は循環節が最大で 6 になる可能性を持ち、その通り 6 の循環節を持っている。どんな有理数が、可能な限度を目一杯使うのだろう。いくら **Java** が浮動小数点数を扱えるといっても、無限に小数点以下を計算してくれるわけではない。そんなときはどうしよう？

それでは、ある有理数がどれぐらいの長さの循環節を持つか調べよう。この段階のプログラムとしては少々手強いかもしれない。でも、がんばって。

いちばんの問題は、循環節の長さの調べ方だ。コンピュータが無限に小数を表示してくれれば楽だが、そうはいかない。そこで発想を変えよう。小数は以前と同じ余りが出たときに循環を繰り返す。すると商を調べるのではなく、余りを調べればよいことに気付くだろう。しかも余りは整数値だから、変数は `int` 型を用意すれば十分だ。また、分数は分子が分母より小さいと決めつけてよい。なぜなら 22/7 のような分数は、必ず $3 + 1/7$ のような形にできる。この場合、循環する鍵を握っているのは 1/7 のような、分子が分母より小さい分数である。したがって、分子が分母より大きい分数を考える必要はない。

programming list [RecurDec.java]

```
1: import java.util.Scanner;
2:
3: public class RecurDec {
4:
5:     public static void main(String[] args) {
6:         Scanner s = new Scanner(System.in);
7:
8:         System.out.print("input 'a b' as a/b: ");
```

```
 9:          int a = s.nextInt();
10:          int b = s.nextInt();
11:
12:          int r = a;
13:          for(int i = 1; i < b; i++) {
14:              r = (r * 10) % b;
15:              System.out.println("R[" + i + "] = " + r);
16:          }
17:      }
18: }
```

このプログラムはいくつか重要なポイントがある。順に見ていこう。

まず、プログラムは分数の分子・分母の 2 値を受け取って実行する必要があるので、理想を言えば `1/7` のように入力したい。ただ、そうすると文字列を分子と分母に分けなければならず、いまの段階では少々ハードルが高い。そこで 1/7 なら、`1 7` のようにして入力することにした。というより、`nextInt()` メソッドがそういう**仕様**なのだ。

では早速、`% java RecurDec` を実行しよう。"input 'a b' as a/b: " に対して `1 7` を入力したものとして処理を追ってみよう。8:行目まではいつもどおりである。

さて、9:, 10:行目は異なる変数に対して同じ処理をしているのだが、実は `nextInt()` メソッドは、空白で区切られた整数値を順に取り込むことになっている。そのため、取り込むものが 2 個あれば、`nextInt()` を 2 回行うことで順に値が代入されるのである。これで **Java** は割られる数 `a` と割る数 `b` の値を知ったことになる。いまは割られる数が割る数より小さいので、割られる数はすでに余りになっていると考えてもよい。そこで `a` を `r` に代入し、分子を最初の余りとして扱っている。これが 12:行目だ。

そして、いよいよ 13:–16:行目の `for` 構文だ。繰り返しの条件が `(int i = 1; i < b; i++)` となっていることに注意してもらいたい。`i` は `1` から始めて `b` より小さい数で終わる。`b` が分数の分母であったことを思い出してほしい。循環小数は分母の数より多い循環節を持たないので、分母 `b` の回数の割り算をするうちに必ず同じ余りが出るものだ。もし、そこまでに同じ余りが出現しなければ、その分数の循環節は許される最大の $b-1$ であることが分かるのである。

さらに、肝心な余りを求める計算は 14:行目の `r = (r * 10) % b;` で求めている。等号は両辺が等しいことを意味しない。前に述べたように、右辺の結果を左辺へ代入するのである。まず右辺だが、`(r * 10)` は余りを 10 倍している。これはわれわれが割り算をする際、上から 0 を下ろす操作

である。続く `% b` は演算記号である。コンピュータプログラムにはいろいろな演算記号があるものだが、`%` はその一つだ。`A % B` と書いた場合、この演算は "`A` を `B` で割ったときの余り" を求める計算なのである。よって一連の `(r * 10) % b` は、余りの 10 倍をもう一度割って、次の余りを求める操作になっているわけだ。そして、次の余りを `r` に代入してやれば、再び同じ式でさらに次の余りを求められることになる。

15:行目の `System.out.println;` 文は、もう説明の必要はないだろう。**Java** は画面に一旦 "R[" まで出力する。続いて `i` が連結されるので、そのときの `i` の値が出力される。さらに画面には "] = " が出力されて体裁が整ったところで `r` の値が連結される仕組みだ。慣れないとややこしいが、たとえば 3 番目の余りが 5 なら "R[3] = 5" のように表示されるのである。

TRY! 1/113 の余りがどのようになっているか調べてみよ。

というわけで、入力した分数の余りが次々と表示され、分数の循環節が長いか短いかは目にできるようになった。しかし、余りを眺めていてもさほど楽しいわけではない。やはり商を眺めて、その分数の特徴を知りたいだろう。だが豆旅は始まったばかりだ。われわれには、まだ知るべきことが山ほどある。ここの景色をじっくりと眺めるのは、帰り道で再びこの地に来たときにしよう。