1.4　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 $0.333\cdots$ 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、$0.333\cdots$ は**循環小数**と呼ばれる数で、円周率 $3.141592\cdots$ 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 $\dfrac{1}{6}$ 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。

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

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ひと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、初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。$0.1666\cdots$ であれば $x = 0.1666\cdots$ とおいて

$$
\begin{array}{rrcl}
 & 100x & = & 16.666\cdots \\
-) & 10x & = & 1.666\cdots \\ \hline
 & 90x & = & 15
\end{array}
$$

のようにすれば、$x = \dfrac{15}{90}$ 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。$0.999\cdots$ のことだ。これも同様に $x = 0.999\cdots$ とおいて

$$\begin{array}{rrcl} & 10x & = & 9.999\cdots \\ -) & x & = & 0.999\cdots \\ \hline & 9x & = & 9 \end{array}$$

としてみよう。あれ？　$x = \dfrac{9}{9}$ になったぞ。ということは $0.999\cdots = 1$ なんだろうか。その説明の前にぜひ $0.4999\cdots$ や $0.6999\cdots$ を分数にしてほしい。

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 $0.\alpha_1\alpha_2\alpha_3\alpha_4\cdots$ の形に表したね。このとき、0.5 のような小数は $0.5000\cdots$ 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 $0.5000\cdots$ 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 $0.4999\cdots$ の形で登場していたのだ。有限小数はすべて $999\cdots$ 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 $0.999\cdots$ へ戻る。$0.999\cdots = 1$ である。なぜなら $\dfrac{9}{9}$ を実際に割り算してみると、$0.999\cdots$ であることが確認できるのだから。

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

これは何だか不思議な計算だ。でも、合っている。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。ひとつは素直に割り切ってしまう計算である。そうすれば $\dfrac{9}{9} = 1$ となる。そしてもうひとつは、いまの例のように $999\cdots$ 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 $999\cdots$ 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。おっと、少し深堀りしすぎたかな。庭いじりが遺跡の発掘作業になる前に、循環小数の話へ戻ろう。

循環小数を計算してみると、$\dfrac{1}{6} = 0.1666666\cdots$ のように**循環節**が短いものと $\dfrac{1}{7} = 0.1428571\cdots$ のように循環節が長いものがある。$\dfrac{1}{6}$ は余りが最大で 5 種類出る可能性がある。$\dfrac{1}{6}$ 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 $\frac{1}{7}$ 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 **Haskell** 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？

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。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 $\frac{22}{7}$ のような分数は、必ず $3+\frac{1}{7}$ 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 $\frac{1}{7}$ 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。たとえば次のスクリプトは、まったく **Haskell** らしからぬ操作だが、$\frac{1}{7}$ の循環節を調べるものだ。

(ghci env.)

```
Prelude> let r1 = [1]
Prelude> let r2 = r1 ++ [(10 * last r1) `mod` 7]
Prelude> let r3 = r2 ++ [(10 * last r2) `mod` 7]
Prelude> let r4 = r3 ++ [(10 * last r3) `mod` 7]
Prelude> let r5 = r4 ++ [(10 * last r4) `mod` 7]
Prelude> let r6 = r5 ++ [(10 * last r5) `mod` 7]
Prelude> r6
[1,3,2,6,4,5]
```

まず、分数は $\frac{a}{b}$ という形だが、いま考えている分数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いる。そこで $\frac{r}{b}=\frac{1}{7}$ と見ている。この際、変数 `r1` に `1` を代入したいのだが、変数への代入は `let` で定義する（`let` は省略してもよい）。そして、まったく不細工なことに余りを順次 `r2`、`r3`、... へ代入するということをしている。**Haskell** に対しても失礼極まりない。

でも、ちょっと我慢して。代入が `r1 = 1` でなく、`r1 = [1]` であることに注意してほしい。単に余りを調べるだけなら `r2 = (10 * r1) `mod` 7` で十分なのだ。しかし、いかにも手続き型言語と同じ処理で余りを求めるのは罪深いものがある。`r1 = [1]` はリストで処理するためである。結果を見れば分かるように、リストには順に余り 1, 3, 2, 6, 4, 5 が格納されることになる。

`r2 = r1 ++ [(10 * last r1) `mod` 7]` が何をしているか説明しよう。まず、`last` はリストの末尾の要素を取り出す関数である。したがって `last r1` は、リストである `r1` —最初の状態は `[1]` だよ—に格納された末尾の値が取り出され `10` が掛けられる。計算の優先順位は関数が先だったよね。これは筆算において、引き算された値に 0 を下ろしてくる操作に当たる。そこに`mod` 7

を施すことで、余りの 10 倍を 7 で割った余りが求められる。そのときに出る余りが次の余りである。具体的には、$10 \times 1 \pmod 7 \to 3$ が計算されている。

ところで mod 関数は、多くのプログラミング言語がそうであるように、**引数（ひきすう）**を与えて `mod A B` のように使う。しかし mod は**中置関数**として `A ‘mod‘ B` のようにも使える。こうすることで、見かけが演算子のようにできるのだ。動作が同じことを示しておこう。

(ghci env.)

```
Prelude> 10 ‘mod‘ 7
3
Prelude> mod 10 7
3
```

さて、求めた余りはリストに加えたい。`r1 ++ [x]` は、リスト `r1` の最後尾にリスト `[x]` をつなげて新たなリストにするものだ。“++” はリストとリストを合わせる関数であることに注意しよう。よって、ここでは新たに出た余りが（リストの形で）既存のリスト `r1` に追加されるて `r2` となる。たったいま求めた余りは 3 だったから、リスト `r2` は `[1, 3]` となるだろう。以上のことを `r6` まで繰り返すと、リストには 7 で割った余りが 6 回分格納されるのである。

さて、余りの列はどうなっただろうか。それは `r6` を打ち込めばよい。

というわけで、与えられた分数の余りを一覧で見ることができたが、余りを眺めていても楽しいわけではない。それに、いちいち変数を取り替えながら入力するのは馬鹿げている。これは、きちんとスクリプトを書いて実行すれば解決することなので、あとでやり直してみよう。いまはリストがどういうものか見ただけである。だが庭いじりは始めたばかりだ。われわれには、まだ知るべきことが山ほどある。変化した庭の様子を眺めるのは、もう少し経ってからだ。