7.3 循環節の謎に近寄る

われわれは、計算可能な循環節を手に入れることができた。[ParaCalc.java] に手を加えて、掛け算によって循環節が巡回する様子を見てみよう。まずはプログラムの提示からだ。

programming list [Cyclical.java]

```
 1: import java.util.Scanner;
 2:
 3: public class Cyclical {
 4:
 5:     static int[] p = new int[1000]; // p[0] is invalid.
 6:     static int[] q = new int[1000]; // q[0] is invalid.
 7:
 8:     public static void main(String[] args) {
 9:         Scanner s = new Scanner(System.in);
10:
11:         System.out.print("input 'b' of 1/b: ");
12:         int b = s.nextInt();
13:
14:         if((b % 2 == 0) || (b % 5 == 0)) {
15:             System.out.println("this 'b' is invalid.");
16:         } else {
17:             int r = 1, cyc = 1;
18:             do {
19:                 p[cyc++] = (r * 10) / b;
20:                 r = (r * 10) % b;
21:             } while(r != 1);
22:
23:             int num = 1;
24:             do {
25:                 recurf(cyc-1, num);
26:                 disp(cyc-1, num);
27:             } while(num++ < cyc - 1);
28:         }
29:     }
30:
31:     public static void recurf(int c, int n) {
32:         for(int i = 1; i <= c; i++) {
33:             q[i] = p[i] * n;
34:         }
35:         for(int i = c; i >= 1; i--) {
36:             q[i-1] = q[i-1] + q[i] / 10;
37:             q[i] = q[i] % 10;
38:         }
39:     }
```

```
40:
41:     public static void disp(int c, int n) {
42:         System.out.print("(" + n + ") ");
43:
44:         int i = 1;
45:         while(c-- > 0) {
46:             System.out.print(q[i++]);
47:         }
48:         System.out.println();
49:     }
50: }
```

ちょっと手を加えるつもりが、これまでにない長さのプログラムになってしまった。原因は、掛け算を配列でしなくてはならないことにある。以前やったことと同じなのだが、当時は常に 2 だけを掛けていたことを覚えているかい？　それに対して今回は、分母が 999 の整数なら、最悪 998 を掛ける必要に迫られる。そのため処理が複雑になった。まあ、とやかく言うよりも順に見ていこうじゃないか。

まず、`main()` メソッドに目をやることにしよう。今回の `main()` メソッドで `[ParaCalc.java]` のそれと違う部分は、25:行目の `recurf()` メソッドだけである。もちろん他にもいくつかの違いが目に付くが、それらは `recurf()` メソッドのせいで必要になっているだけだ。たとえば 23:行目の `num = 1` がそうであり、また、24:–27:行目が `do-while` 構文で囲まれているのがそうである。

`[Cyclical.java]` では、ただ一つの循環節を表示するだけでなく、そこに一定の数を掛けた状況を示したい。一定の数とは、1 から循環の回数までのすべての数である。1/17 なら循環節は 16 なので、1 から 16 までの数を掛けた状況が示されるようにするのだ。そのための `do-while` 構文であるが、27:行目の条件がそのことを表している。

25:行目の `recurf()` メソッドは、掛け算をするためのメソッドである。そのためには、循環がどれぐらい続くかのデータ `cyc`、それにいまいくつの数を掛けているかを示す値 `num` を渡してあげなければならない。これらがきちんと渡せれば、あとは `recurf()` が処理をしてくれる。

では、`recurf()` が何をしているか探っておこう。

循環回数と掛ける数を受け取った `recurf()` メソッドは、32:–34:行目にかけて、各配列を定数倍している。そして定数倍した値を、配列 `p` ではなく `q` に代入している。なぜそうしているのかといえば、配列 `p` は定数倍するための基準値として固定しておきたいからだ。ちなみに、配列 `p`, `q` は `int` 型で、いまのところ 1000 桁の循環までに対応しているので、掛けられる定数は 1000 未満

だ。よって、桁あふれの心配はない。むしろ桁余りが生じるから、気になったら short 型を使えばよい。

しかし、結果の表示は 1 桁ずつだから、余分な値は上の配列に繰り延べていかなくてはならない。それが 35:–38:行目の処理だが、この方法はすでに登場した [PowerOf2.java] の焼き直しに過ぎない。

EX. [PowerOf2.java] では配列の繰り延べが p[i+1] = p[i+1] + p[i] 云々（うんぬん）だったが、ここでは q[i-1] = q[i-1] + q[i] 云々である。配列の繰り延べ方が逆になっている理由を考えよ。

実は 32:–38:行目までの二つの for 構文は、一つにまとめて書くことが可能だ。だが、プログラムは短ければよいという代物ではない。多少長めでも、処理が自然に追えるほうが分かりやすいものだ。

recurf() メソッドが void 型なのは、この関数が掛け算の結果を配列に納めるだけだからである。値を返す代わりに、広いスコープの変数を使って、別のメソッドに処理を委ねるのだ。

処理を委ねられるのが disp() メソッドだ。本当は [ParaCalc.java] での disp() メソッドと同じく、disp(cyc) だけでよかったのだけれど、掛けている数を表示させたいために num の値も渡している。そのため、while 構文に入る前に、42:行目によって掛けている数の表示を済ませている。45:–47:行目の処理内容は同じだが、i = 0 で始めるか i = 1 で始めるかに違いを認めることができる。System.out.println; 文を } の直後に書いたのは、単に “お作法” の問題である。

TRY! なぜ同じ処理でありながら、[ParaCalc.java] の disp() メソッドでは i = 0 から始め、[Cyclical.java] の disp() メソッドでは i = 1 から始めたのか？

TRY! 早速 1/17 や 1/61 などの循環節について調べてみよ。1/17 や 1/61 は 1/7 と同じ性質を持つが、他の分数はどんな性質になっているだろうか。