2.4 べき乗の計算

さあ、べき乗の計算でかなり大きな数が現れても、桁の処理をうまくやれば何とかなることが分かった。さっそく 2^x あたりの計算に利用してみよう。しかし、いくら大きな桁を扱えるといっても所詮コンピュータのことである。ある程度の限界というものはある。たとえば 2^{1000} まで扱うためには、何桁の数が扱えなければならないのだろう。

ここで指数の考えが登場する。10^1 が 2 桁の数、10^2 が 3 桁の数、10^3 が 4 桁の数、... とくれば、10^x が $x+1$ 桁の数であることは容易に想像がつくというものだ。正確には、10^1 から 2 桁の数が始まる、と言うべきだろう。そのため、$10^{1.1}$, $10^{1.2}$, $10^{1.3}$, ... ,$10^{1.9}$ などはすべて 2 桁の数であり、10^2 となったところから 3 桁の数が始まるのだ。

したがって 2^{1000} が何桁の数であるかを知るには

$$2^{1000} = 10^x$$

を解いて、x の値を知ればよいことになる。この方程式は容易に解けるものではないが、対数を使えばその限りではない。両辺について底 10 の対数をとれば、

$$1000 \log_{10} 2 = x$$

であるが、$\log_{10} 2 \approx 0.30103$ であることを使って $x \approx 301.03$ を知ることになる。つまり 2^{1000} は 302 桁の整数なのだ。

よーし、それならここでは、400 桁までの数が扱えるようにがんばってみよう。と言いたいところだが、説明の都合で 40 桁までにさせてもらおう。プログラムは 2^{100} の計算例だ。

programming list [PowerOf2.java]

```
 1: import java.text.DecimalFormat;
 2:
 3: public class PowerOf2 {
 4:
 5:     public static void main(String[] args) {
 6:         int[] p = new int[11]; // p[10] is invalid.
 7:
 8:         p[0] = 1;
 9:         p[1] = p[2] = p[3] = p[4] = p[5] = p[6] = p[7] = p[8] = p[9] = 0;
10:
```

```
11:         for(int n = 1; n <= 100; n++) {
12:             for(int i = 9; i >= 0; i--) {
13:                 p[i+1] = p[i+1] + (p[i] * 2) / 10000;
14:                 p[i] = (p[i] * 2) % 10000;
15:             }
16:         }
17:
18:         DecimalFormat df = new DecimalFormat("0000");
19:
20:         for(int i = 9; i >= 0; i--) {
21:             System.out.print(df.format(p[i]));
22:         }
23:         System.out.println();
24:     }
25: }
```

はじめに import 文に新しい記述が現れたね。以前キーボードからの入力を受け付けるために、java.util.Scanner（java.util パッケージに属する Scanner クラス）を使ったのを覚えているかな。ここでは出力する値の書式を整えたいので、java.text.DecimalFormat（java.text パッケージに属する DecimalFormat クラス）を使う。後述するが、数値の出力をきっちり 4 桁の整数で出力したいのだ。

6:行目で初期化した配列の説明をしておこう。ここで p[0]〜p[10] までの 11 個のインスタンスが生成されたわけだが、実は p[10] はダミーである。なぜダミーが要るかはすぐあとで分かる。さて、8:行目で p[0] = 1 を代入した以外は、9:行目で p[1] から p[9] までを 0 で初期化したことになるが、これはこういうことだ。用意した配列は、p[0] がいちばん下の桁で p[9] がいちばん上の桁を想定している。つまりこの時点で、配列を p[9]...p[0] と並べると、0...01 なる数ができたことになる。0...01 は何桁の数だろうか。配列の変数を 10 個用意したから 10 桁の数？　残念でした。あとからの説明にも関係するのだが、これは 40 桁の数になる。なぜなら各 p[] には 4 桁の数が与えられるからである。

p[] に 4 桁の数が与えられるのには理由がある。むかし **C** で組んだプログラムを **Java** へ持ってきたからだ。古い **C** は int 型の整数が 2 バイトの可能性もある。その場合は数万の大きさで頭打ちになってしまう。このことは 9999 までは保証されるが 99999 までは保証されないことを意味する。よって 9999 まで保証されるものを 10 個つなげれば、40 桁の数が保証されるのである。

11:行目の for 構文で、n を 100 回繰り返しているが、これは続く for 構文のブロック—これはす

べての p[] を 2 倍することに相当する—の 100 回の繰り返しである。これで 2^{100} を計算しているわけだ。

このプログラムの主要部分は、12:行目からの for 構文にある。ここは for 構文であるが、たった 1 回だけ 40 桁の数を 2 倍するところだ。単純に各 p[] を 2 倍しただけでは、繰り上がりを無視することになってしまう。

そのため 13:行目の (p[i] * 2) / 10000 により、繰り上がりがあった場合に、上の桁—ここでは p[i+1] を上の桁に想定している—に繰り上がった数を加えている。ところで for 構文が i = 9 から代入を始めていることに気付いているかい？　繰り上がりの関係で i = 0 から始めるとまずいのだ。

EX.　i = 0 からの代入で生じる不都合は何か考えてみよ。

14:行目の (p[i] * 2) % 10000 は、繰り上がらない部分の処理だ。除算に / を使う場合と % を使う場合で、計算結果がどうなるか理解できるだろうか。if 構文を使っても同様の効果を得ることはできるが、わざわざ手の込んだ式を使っている。ちょっとした頭の体操程度に思ってもらえればよいのだが。

EX.　ところで 13:行目と 14:行目の処理順を逆にすると正しい結果にならない。なぜか？

さて、これで 2^{100} の計算はできた。次はこれを表示しなくてはならない。表示は上の桁から順に並べればよい。そのために 18:行目で表示の仕方を指定している。この指定がないとたとえば p[9] = 1234、p[8] = 987 のときは、本来 12340987... となるべきものが 1234987... となってしまうからだ。"0000" の書式指定は 4 桁を保証するために必要だったのである。これでめでたく 12340987... のように表示されるわけである。

TRY!　3^{50} の値を求めてみよ。

TRY!　**Java** では int 型は 4 バイトだから p[] に 8 桁の数を与えられる。10 個つなげて 80 桁だ。2^{200} の計算ができるようにプログラムを変更してみよ。

では、p[10] がどうなったかに触れておこう。12:行目の for 構文により最初は i = 9 で始まる。するとプログラムは 13:行目で p[10] = p[10] + (p[9] * 2) / 10000; の代入をすることになる。

もし配列変数が `p[9]` までしか用意されていないと、**Java** はエラーを返すことになるだろう。だからダミーの配列変数 `p[10]` が必要だったのだ。