## 2.3 大きな数の扱い方

指数関数の計算をしてみると分かることだが、計算結果は爆発的に大きくなってしまう。一般にコンピュータでは、一つの変数が扱える整数は場合によっては数万程度の大きさしかないから、すぐに桁あふれを起こしてしまうのだ。$y = 2^x$ といった単純な関数でさえ、コンピュータはついて行けない。$x = 30$ ではコンピュータは悲鳴をあげるだろう。$x = 80$ ともなればお手上げだ。

コンピュータで大きな数を扱うことは不可能なのだろうか？　もちろんそんなことはない。解決のヒントは、われわれが何気なく使っている数に隠れている。われわれは機械と違って、数を記憶するための制限を持たない。だから、10 億でも 1 兆でも扱える。本当にそう思っている？　実はわれわれのほうがコンピュータより貧弱だ、と言ったらどう考えるだろうか。だが、これは事実なのである。コンピュータは一つの変数で数万程度の大きさしか扱えないと言ったが、われわれは一つの桁で 9 までの数しか扱えないことを知っているだろうか。

そろそろ私の言いたいことが見えてきたことと思う。つまり、こういうことだ。われわれは 0〜9 までの 10 種類—今後のためにあえて言うが、1〜0 までの 10 種類ではない—の数しか扱えないが、位取りをしているために望むだけ大きな数を扱えるのである。これをコンピュータにも応用しよう。一つの変数で数万の数しか扱えなくても、位取りの考えを持ち込むことで大きな数を扱うことができるのである。普通コンピュータは 2 進数で演算していると言われるが、一つの変数を一つの桁にすれば数万進数で演算することも可能なのだ。われわれが貧弱と言ったのは、そういう意味である。

では、指数関数の計算に入る前に**配列**について話しておこう。

programming list [Array.java]

```
 1: public class Array {
 2:
 3:     public static void main(String[] args) {
 4:         int[] f = new int[3];
 5:
 6:         f[0] = 2; f[1] = 5; f[2] = 6;
 7:
 8:         int fn = 100 * f[0] + 10 * f[1] + f[2];
 9:         int rn = 100 * f[2] + 10 * f[1] + f[0];
10:
11:         System.out.println("  a first num: " + fn);
```

```
12:         System.out.println("a reverse num: " + rn);
13:     }
14: }
```

---

配列による変数を宣言する場合は、それが単なる整数型に留まらず整数配列型であることを明示するために `int[]` を用いる。配列の初期化は `new` を用いて 4:行目のようにする。前にインスタンス生成のために使った書き方だ。4:行目は

```
int[] f;
f = new int[3];
```

をまとめて書いたものである。また、`int[] f` は `int f[]` と書いてもよい。書き方に融通が効くのはよいのだが、人それぞれ好みが異なるので戸惑うこともあるだろう。

さて、初期化の際 `f = new int[3]` と書いたら 3 個の変数 `f` が、`f = new int[10]` と書いたら 10 個の変数 `f` が用意されるのだが、注意しなくてはならないことがある。それは、`int[3]` で用意される 3 個の変数は `f[0]`, `f[1]`, `f[2]` の 3 個であり、`int[10]` で用意される 10 個の変数は `f[0]`, `f[1]`, ..., `f[9]` の 10 個である点だ。つまり `int[n]` という配列は、(配列の名前が `f` のとき) `f[0]` から `f[n-1]` までの $n$ 個の変数になるのである。このことは、$n$ 進法で 0 から $n-1$ までの $n$ 個の数が使われることと合致している。さっき 0〜9 までの 10 種類の数を扱うと強調したのは、ここでの説明を見越してのことだ。そういうわけで、いまは `f` という名の配列を宣言し、`f[0]`, `f[1]`, `f[2]` の 3 変数が準備されたことになる。

プログラムは 3 桁の整数を模している。4:行目の `int[3]` によって 3 桁分の "位" を用意したと思ってもらえばよいだろう。位というのは少し変だが、この場合の処理ではそうしている。つまり `f[0]` の位、`f[1]` の位、`f[2]` の位が用意されたことになる。

6:行目で位に使われる数字をそれぞれ 2, 5, 6 にしたところだ。だからと言ってこの時点で、256 の数ができたわけではないことを注意しておこう。

8:行目の代入で、`f[0]`, `f[1]`, `f[2]` の順に百の位、十の位、一の位の数が決まるので、ここではじめて 256 という数ができたことになる。すると 9:行目の代入は、各位を逆順にした数になっていることに気付くだろう。したがって、ここでは 652 という数ができたのだ。それが 11:, 12:行目で表示される。

プログラムはこれだけだが、このネタをもう少し調理しよう。

2 進数というのはコンピュータではよく使われる、1001011 や 11111011 といった数のことだ。わ

れわれは普段 10 進数を使っているので、これらの数が 77 や 251 だということに気付かないものだ。コンピュータは 1 バイトという単位を使うが、1 バイトは 8 ビットである。え？　何を言ってるの分からないって？　つまりこういうことだ。1 ビットとは 0 か 1 だけが使える桁のことである。だから 8 ビットとは 0 か 1 が使える桁が 8 個あることを指している。そしてこれが 1 バイトになる。要するに 1 バイトで 00000000〜11111111 までの数が扱えるのだ。次のプログラム例は 1 バイトの 2 進数がいくつになるかを調べるものだ。

programming list [BinToDec.java]

```
 1: public class BinToDec {
 2:
 3:     public static void main(String[] args) {
 4:         int[] f = new int[8];
 5:
 6:         f[0] = 1; f[1] = 1; f[2] = 1; f[3] = 1;
 7:         f[4] = 1; f[5] = 0; f[6] = 1; f[7] = 1;
 8:
 9:         int dec = 2 * (2 * (2 * (2 * (2 * (2 * (2 * f[0] + f[1]) +
                              f[2]) + f[3]) + f[4]) + f[5]) + f[6]) + f[7];
10:
11:         System.out.println("decimal: " + dec);
12:     }
13: }
```

プログラムは一部が美しくないけれど、正確な値を出してくれる。美しくないのは 6:, 7:行目で、ご丁寧にも一つ一つの配列変数に代入している。この代入によって 2 進数の 11111011 がセットされたことになる。実はこのセットの仕方も美しくない原因になっているのだが気にしないでおこう。

さて、10 進数なら十の位が $10^1$、百の位が $10^2$、千の位が $10^3$、... である。2 進数では十の位とは呼ばないので、下の位から順に $2^0$ の位、$2^1$ の位、$2^2$ の位、... である。したがって 11111011 を 10 進数にするには $2^7 f[0] + 2^6 f[1] + \ldots + 2^0 f[7]$ の計算—ほらね、美しくないでしょ。だって指数と配列番号が一致してないもの—をすればよい。しかしコンピュータはべき乗の計算を得意としていないのだ。だから $2^7 f[0] + \cdots$ の計算をさせたければ、`dec = 2*2*2*2*2*2*2*f[0]+` $\cdots$ と書かなくてはならない。

そこで 9:行目のような、中途半端に因数分解した式を使うことになるのだけれど、これがコンピュータにとって実に好都合なのだ。

**EX.** `dec = 2*(2*(2*(2*(2*(2*f[0]+ ··· )+f[7]` が `dec = 2*2*2*2*2*2*2*f[0]+ ··· +f[7]` に等しいことを確認せよ。

`dec = 2*(2*(2*(2*(2*(2*f[0]+ ··· )+f[7]` と `dec = 2*2*2*2*2*2*2*f[0]+ ··· +f[7]` を較べてほしい。掛け算をする回数がかなり減っていることが分かるはずだ。実際、前者は 7 回の掛け算で済んでいるが後者は 28 回も掛け算をしている。コンピュータは掛け算に時間がかかってしまうものである。この程度の計算では恩恵が目立たないが、9:行目に使われた式の方が、掛け算の回数が少ないから計算時間が短縮されるのだ。

**TRY!** 5 進数 $\rightarrow$ 10 進数にするプログラムにしてみよ。

ところで話題をプログラム `[BinToDec.java]` に戻すことにしよう。ここでは 2 進数の 11111011 を 10 進数に直しているが、実行すると 251 が返ってくるはずだ。だが、`int` 型では最高位の 1 は負の数であることを示している。そのため `int` 型の 11111011 は $-5$ を表すのだ。このプログラムが 11111011 $\rightarrow$ 25 となるのは、純粋に数学の定義に則（のっと）って計算しているからである。

では、コンピュータの定義ならどうなるだろう。8 桁では分かりにくいだろうから、4 桁の 2 進数を例にとろう。

コンピュータは最高位が 0 か 1 によって正の数と負の数の区別がつくので、0111 は正の数で 1111 は負の数である。しかし 1111 は、正の数 0111 にマイナスの意味で最高位の 1 を付けているわけではない。1111 は $-1$ である。なぜなら $1111 + 1 = 0000$ だから、$1111 = -1$ であることが分かるというものだ。2 進数は 1111, 1110, 1101, 1100, 1011, ... という具合に減るから、これらは 10 進数で $-1$, $-2$, $-3$, $-4$, $-5$, ... を表している。

もし 4 桁の 2 進数が用意されれば、全部で $2^4 = 16$ 通りの数が表せる。その内、0xxx で表される正の数が 8 通り、1xxx で表される負の数が 8 通りだ。0xxx の正の数には 0 も含まれるので、正の数は 0 から 7 までだが、1xxx の負の数は $-1$ から $-8$ までとなる。お分かりかな？　よって 4 桁の 2 進数が扱える数は $-8$ から 7 までである。

ちょっと前に、4 バイト—これは 32 桁の 2 進数に相当している—なら $256^4 (= 2^{32}) = 4294967296$ 通りの数が使えるから、ざっくり $-2147483648$〜2147483648 の整数が扱えると言った。4294967296 通りの半分はたしかに 2147483648 通りであるが内状はちょっと違う。負の数は $-1$ から始まるが

正の数は 0 から始まる。だから実際に扱える数は $-2147483648$〜$2147483647$ となるのだ。