## 6.2 完全数を探す

プログラム [DispOfCM.java] で約数をすべて列挙できたので、それらの和をとれば完全数を探すのに役立つだろう。和を求めるために変数 sum を追加しておこう。

programming list [PNsearch.java]

```
 1: import java.util.Scanner;
 2:
 3: public class PNsearch {
 4:
 5:     public static void main(String[] args) {
 6:         Scanner s = new Scanner(System.in);
 7:
 8:         System.out.print("input your candidate: ");
 9:         int n = s.nextInt();
10:
11:         System.out.print(n + " ::  1 ");
12:         int sum = 1;
13:         for(int i = 2; i <= n / 2; i++) {
14:             if(n % i == 0) {
15:                 System.out.print("+ " + i + " ");
16:                 sum += i;
17:             }
18:         }
19:         System.out.println("= " + sum);
20:     }
21: }
```

今度のプログラムは、自分自身を含まない約数の和を "8 :: 1 + 2 + 4 = 7" のように表示させるのが目的である。そのため各 System.out.print; 文は、目的に添うように少々手を加えてある。:=が :: に変わっているのもその一環だ。それぞれの System.out.print; 文を注意深く見てもらいたい。ちなみに先の例では、8 は完全数ではないので和が 8 になっていないことも指摘しておこう。

プログラムは sum に関わる部分だけが変更されている。12:行目に追加したのは sum = 1; である。この時点で画面には "** :: 1 " が表示されているので、同時に sum に 1 を代入したのだ。

同様に 16:行目は、約数を一つ表示するたび、それを sum に加えている。この際、sum += i; の表現が目新しい。これは sum = sum + i; を省略して書いたものだ。何も省略した書き方をしなくてもよいけれど、こういう書き方にも慣れておく方が得策だ。**Java** のプログラムを見ると、この

書き方をよく目にするはずだ。それにこの書き方は、以前 `i += 2;` で目にしている。意味は `i` が 2 ずつ増えることだったと思うが、`i = i + 2;` の省略形だったね。

19:行目で和を表示したら終了だ。自分自身は和に加えないので、この時点での `sum` の値を表示させれば十分だ。

**TRY!** 6 と 28 が完全数であることを確認せよ。また、いくつかの数を入力して完全数を探してみよ。運良く見つかれば大したものだ。

おそらく [PNsearch.java] では完全数を見つけられなかったのではないか？ それもそのはず、100 や 200 までには完全数は存在しないのだ。そうなると行き当たりばったりの入力では発見は難しいだろう。そこで、和が入力した数と等しくなったとき、その数を出力するようにすればよい。そうすれば、コンピュータは次の完全数を見つけるまで計算をしてくれる。君たちはお茶でも飲みながら、気楽に待っていればいいのだ。もっとも、コンピュータはお茶を入れる時間を与えてくれないだろうが。

programming list [PNfind.java]

```
 1: public class PNFind {
 2:
 3:     final static int SUP = 1000;
 4:
 5:     public static void main(String[] args) {
 6:         for(int n = 6; n <= SUP; n++) {
 7:             int sum = 1;
 8:             for(int i = 2; i <= n / 2; i++) {
 9:                 if(n % i == 0) {
10:                     sum += i;
11:                 }
12:             }
13:             if(n == sum) {
14:                 System.out.println(n + " is a perfect number.");
15:             }
16:         }
17:     }
18: }
```

プログラムは [PNsearch.java] に手を加えたものである。しかし、入力を要求しないので "御用聞き" 要らずとなった。

その代わり上限 `SUP` までにある完全数を見つけるため、3:行目で `SUP` を設定し、6:行目から `for`

構文を用いて探すことにしてある。最初の完全数は 6 であることが分かっているので、`n = 6` から始めている。

7:–12:行目は `[PNsearch.java]` と同様だが、いちいち約数を表示する手間を省いている。そのため `System.out.print;` 文はない。

13:行目が、完全数かどうかの判断をするところだ。完全数であれば 14:行目でそのことが知らされる。

**TRY!** 3 番目の完全数を探してみよ。さらに、`SUP` を `10000` にして、4 番目の完全数を探してみよ。

やってみると容易に分かるように、完全数は非常に少ない。5 番目の完全数を探すのはさらに骨が折れるだろう。それもそのはずで、完全数は $2^{n-1}(2^n-1)$ の形をしている。これでは指数関数的に大きな数になってしまうから、完全数がどんなにたくさんあったとしても、気軽に発見できないのだ。

**EX.** 4 番目までの完全数が $2^{n-1}(2^n-1)$ の形になっていることを確認せよ。

そこで忠告をしておこう。このプログラムで 5 番目の完全数を探す暴挙に出ないように。5 番目の完全数は 8 桁の数だ。これは `int` 型の守備範囲だと考えないでほしい。ここでの**アルゴリズム**は、8128 を見つけるのでさえ、少々時間を要する。単純に考えても、1 桁増えるごとに計算時間は 10 倍になるから、8 桁の解を見つけるには 10000 倍の時間がかかる。8128 を 0.1 秒で見つけられても、5 番目の完全数を見つけるには 1000 秒（16 分 40 秒）もかかるのだ。しかし悪いことばかりではない。今度は、お茶を入れる時間が十分与えられるからね。

万が一にも、このプログラムで 5 番目の完全数を探す誘惑にかられたら、どのぐらいの時間がかかるか予測すべきだ。たとえば `n` の値によって、`switch` 構文で分岐させるのも一つの手だ。`case 10000:`, `case 100000:`, `case 1000000:`のたびに `n` の値を表示させれば、8 桁の数にたどり着く時間の目安が立つ。すると、このプログラムが役立たずであることが発覚するはずだ。そこで $2^{n-1}(2^n-1)$ を利用するのが得策だと気付くだろう。これを利用すれば、5 番目の完全数はすぐ見つかる。

結局のところ、コンピュータの計算速度を過信してはいけないということだ。そのために、どうしても人の手で、効率的なアルゴリズムが必要になるのである。

**TRY!** このままのプログラムでは、約数の表示がない。約数の和が表示できるよう、プログラムに修正を加えよ。

**TRY!** $2^{n-1}(2^n-1)$ の形の数だけ調査するプログラムに変更し、5 番目までの完全数を表示させてみよ。