6...の散歩道

6.1　完全数

6 は興味深い数のひとつだ。それはどういうことかというと、6 の約数は 1, 2, 3, 6 だが、

$$1+2+3=6$$

となっている。この性質を満たす数は 6 の近辺にはない。たとえば 8 は、約数として 1, 2, 4, 8 を持つが、明らかに $1+2+4 \neq 8$ である。約数をもれなく取り出すことができれば、6 の次にこの性質を満たす数が 28 であることを知るのは容易（たやす）い。実際、28 の約数は 1, 2, 4, 7, 14, 28 であるから

$$1+2+4+7+14=28$$

になっている。この性質を持つ数を**完全数**と呼ぶ。

性質を誤解のないように言えば、完全数とは

自分自身を除く約数の総和が自分自身に等しい数

であると言える。なんと、日本語にするほうが式で説明するより難しくなってないかい？　数学とはそんなものだ。

それなら、28 の次の完全数はいくつだろうか。ちょっと計算すればすぐに見つかりそうだ。素数のように、明らかに完全数になり得ないものは省ける。約数の数が少ないものや (素数)^n のような数もたいてい省けそうである。このように候補となる数を絞っていけば、意外に早く見つかるかもしれない。悪いね。そんな簡単には見つからないのだ。よほど根気よくなければ、次の完全数を見つけるのは難しい。根気が勝負になるときこそ **Python3** の出番というわけだ。

いきなり完全数を求めるスクリプトを書くのも大変だろうから、まずは約数を見つけるスクリプトから見ていこう。

[py script]

```
>>> def measures(n):
...   print(1, end=' ')
...   for i in range(2, n//2+1):
...     if n % i == 0:
...       print(i, end=' ')
...   print(n)
...
>>> measures(45)
1 3 5 9 15 45
```

いままでの知識さえあれば十分理解できるスクリプトだ。コードは与えられた数に対して、約数をすべて表示することになっている。ところで、n の約数に 1 と n があるのは当然なので、これらは最初と最後に表示することにした。これが最初と最後の `print` 文である。

`for` 構文は `n` を 2 から順々に割っていくものだ。約数を求める行為は素数を求める行為と違うので、2 で割れても 4 で割れるか試す必要があるし、1 ずつ大きな数で割る必要もある。ただし、$\frac{n}{2}$ までの数で試せばよい。それより大きい数が約数になることはないのだから。でも **Python3** の仕様上、指定する範囲は `range(2, n//2+1)` である。

`n % i` が割り切れれば `i` は約数であるから出力すればよい。そうでなければ次を試すまでだ。最後に残った数を表示すれば終了だ。

スクリプトは約数を列挙するものだが、素数を見つける役にも立つ。なぜなら、素数は 1 と自分自身しか約数を持たない数だから、たとえば 17 に対しては 1 と 17 だけが表示される。

ところで、ここに提示した約数を求めるスクリプトは、単純きわまりないアルゴリズムであるだけに無駄が多い。当然のことながら、たとえばある数 n が i で割れたなら、その商である $\frac{n}{i}$ も約数である。つまり約数は基本的にふたつ同時に見つけることができる。基本的と表現したのは、49 が 7 で割れたからと言って、$\frac{49}{7}$ が別の約数とは言えないからだ。しかし、除数 i と商 $\frac{n}{i}$ を一緒に表示してしまえば、割り算は $\sqrt{n}$ まで試せばよいことになる。これは以前、素数を求めた際に使った手法だし、計算量を減らす効果も十分だ。余裕があればこの考え方でスクリプトを書き換えてほしい。

だが、完全数を求めるためには、これではまったく不十分なのだ。あとで分かるように、完全数は素数とは較べものにならないくらい稀にしか現れない。つまり、計算量が減ること自体は喜ばしいけれど、結局ほとんどの数で調査が不発に終わる。効果的に調べるには、計算量を減らすのではなく、調査しなくてもよい数を飛ばすことである。

さて、約数をすべて列挙するスクリプトが書けたので、それらの和をとれば完全数を探すことが

できる。

[py script]

```
>>> def pnfind(n):
...   for n in range(6, n):
...     sum = 1
...     for i in range(2, n//2+1):
...       if n % i == 0:
...         sum += i
...     if n == sum:
...       print(n)
...
>>> pnfind(10000)
6
28
496
8128
```

スクリプトは関数 measures に、和を求めるために変数 sum を追加したものである。sum は自分自身以外の約数の和を加算する変数だ。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。

sum に 1 を代入したのは、最初の約数が自明の 1 だからである。あとは、約数が見つかるたびに sum に加えるだけだ。そして、与えられた数と約数の合計が等しくなったら、それが完全数である。

やってみて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ためには相当の時間を覚悟しなくてはならない。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。いま探した四つの数でちょっと確認してほしい。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。

そこで忠告をしておこう。このスクリプトで 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 桁の数だが、**Python3** の守備範囲だと考えないでほしい。このアルゴリズムは、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。だけど、その間にお茶を入れる時間ができるので、必ずしも悪いことばかりではない。

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。手っ取り早く 5 番目までの完全数を知りたいなら、$2^{n-1}(2^n-1)$ の形の数だけ調査するスクリプトに変更すればよい。驚くほど早く結果を目にすることができる。でも、今度はお茶を入れる暇がなくなるけど。