5.2 漸化式

では、フィボナッチ数列に戻ろう。

よく数列の n 番目の項を a_n などと表記することがある。すると次の $n+1$ 番目の項は a_{n+1}、$n+2$ 番目の項は a_{n+2} という書き方になるだろう。フィボナッチ数列は、直前の 2 項の和が次の項になっているので、一般に

$$a_{n+2} = a_{n+1} + a_n$$

という書き方ができる。特にいまは 1, 1 から数列を始めているので、$a_1 = 1$, $a_2 = 1$ の条件が等式に付け加わることになる。

これだけの条件と式があれば、この先の項が順次計算できるのだ。このような形式で与えられる式を**漸化式**と呼ぶ。数学の世界に限らず、前後の関係は分かっているのだが、そのものズバリを与える式が不明であることは多い。実はフィボナッチ数列もそんなもののひとつだ。数列の前後の関係は漸化式により明確に分かっている。では、フィボナッチ数列のズバリ第 n 項を求める式は何だろう。これはちょっと複雑な計算をするので、詳しいことは省くが、フィボナッチ数列の第 n 項 a_n は

$$a_n = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right\}$$

で与えられる。

実際に計算すると大変だが $n = 1, 2, 3, \ldots$ を代入すれば

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ \ldots$$

が現れるのだ。式には $\sqrt{5}$ があるのに、フィボナッチ数列には一切登場しない点が興味深い。近似値でよければ、{ } 内の前の項が $(1.618033\cdots)^n$ で、後ろの項が $(0.618033\cdots)^n$ であることを利用しよう。n が大きいほど $(0.618033\cdots)^n$ は 0 とみなせるので、第 a_n 項は近似的に $\dfrac{1}{\sqrt{5}}(1.618033\cdots)^n$ と考えてよい。電卓を使えば、$n = 9$ のときは $a_n \approx 33.994116$ であることが分かる。確かに第 9 項の 34 に一致している。

Python3 には漸化式をうまく扱える仕組みが備わっている。以前、フィボナッチ数列を生成したスクリプトを覚えているね。そして君たちはそれをもとに、整数 n を与えて n 番目のフィボナッチ数を表示するスクリプトを書いただろう。え？　書いてないだって？　だめだなあ、手抜きをしちゃあ。まあ、いいや。ここでは当時とまったく同じ動作をするものを、漸化式から作ってみよう。次のスクリプトも、整数 n を与えると n 番目のフィボナッチ数を表示する。

[py script]

```
>>> def fib(n):
...   if n > 2:
...     return fib(n - 1) + fib(n - 2)
...   else:
...     return 1
...
>>> fib(10)
55
>>> fib(30)
832040
```

たった 5 行のスクリプトではあるけれど、これでフィボナッチ数が計算できる。信じられない？ たとえば 10 を与えれば 55 が返ってくる。間違いないよね。でも、どうして？

仕組みを説明するために、具体的に `n = 4` が与えられたとして処理を追ってみよう。

`n = 4` だから `if n > 2` の条件より `fib(n - 1) + fib(n - 2)` の値が返されるはずである。あれれ？　いまは `fib(何々)` の計算のはずだ。なのに `fib(何々)` の計算に `fib(何々)` を使うのかい？　疑問はもっともだが、それでよいのである。もやもやを抱えつつも先へ行こう。`fib(4)` の場合、返されるのは `fib(3) + fib(2)` である。しかし、`fib(3)` だって `n > 2` の条件を満たすから、`fib(2) + fib(1)` が返されるはずである。おいおい、これじゃ終わらないんじゃない？

心配無用である。`fib(3)` が `fib(2) + fib(1)` になれば、`fib(2)` と `fib(1)` は `n > 2` の条件を満たさないので `else` によって 1 と 1 が返るからだ。結局のところ、`{fib(3)} + fib(2)` が返るということは`{1 + 1} + 1` が返ることに等しいのである。つまり、`fib(4)` に対しては 3 が返るわけだ。

一見すると詐欺にあったような印象を受けるかもしれない。このような呼び出しは**再帰呼び出し**という。前後関係がはっきりしていて、一般の式を求める必要がないときなどに有効である。ただし、再帰呼び出しは計算量が爆発的に増加してしまう。たとえば `fib(6)` の計算でも

```
fib(6)  =  fib(5) + fib(4)
        =  {fib(4) + fib(3)} + {fib(3) + fib(2)}
        =  {(fib(3) + fib(2)) + (fib(2) + fib(1))} + {(fib(2) + fib(1)) + 1}
        =  ...
```

のように、`fib(6)` が `fib(5)`, `fib(4)` を呼び、`fib(5)` と `fib(4)` がそれぞれ `fib(4)`, `fib(3)` と `fib(3)`, `fib(2)` を呼び、さらにそれぞれが...。何ということだ。呼び出しが倍々に増えているじゃないか。

あまり大きな数を与えるととんでもないことになるから、とりあえず `fib(40)` 程度の計算をさせてみよう。さて、君たちのコンピュータは即座に結果を表示してくれただろうか。おそらく即座に結果を返してくれまい。もし、一瞬のうちに結果が返る高速コンピュータを使っているなら、うらやましい限りだ。理由はこうだ。

`fib(2)` や `fib(1)` はふたつの `fib` 関数を呼ぶわけではないので、単純に `fib(n)` の計算が 2^{40} 回行われることはない。実際はもっと少なく、2^{27} 回ほどだ。一口に 2^{27} 回と言うものの、これでも軽く 1 億回を超えている。1 秒間に 1 億回の呼び出しが可能なコンピュータでも 1 秒はかかるので、“即座” に結果が出るわけではない。われわれが普段使っているコンピュータなら、少なくとも数秒は要するはずだ。再帰呼び出しをうかつに利用するのは要注意である。