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

$$a_{n+2} = a_{n+1} + a_n \tag{5.1}$$

という書き方ができる。特に今は 1, 1 から数列を始めているので、$a_1 = 1$, $a_2 = 1$ の条件が (5.1) 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。(5.1) 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 (5.1) 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、{ } 内の初めの項が $(1.618033\cdots)^n$ で、後の項が $(0.618033\cdots)^n$ 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 $1/\sqrt{5}\,\{1.618033\cdots\}^n$ 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。

C++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するプログラム `[fibonac2.cpp]` を覚えているね。そして君たちは `[fibonac2.cpp]` をもとに、整数 n を入力すると n 番目のフィボナッチ数を表示するプログラムを作っただろう。え？　作ってないだって？　だめだなあ、手抜きをしちゃあ。まあ、いいや。前回のプログラムは、漸化式とは無縁だし。

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。次のプログラムも、整数 n を入力すると n 番目のフィボナッチ数を表示する。

programming list [nthfib.cpp]

```
 1: #include <iostream>
 2:
 3: int fib(int n) {
 4:     if(n > 2) {
 5:         return (fib(n-1) + fib(n-2));
 6:     } else {
 7:         return 1;
 8:     }
 9: }
10:
11: int main() {
12:     int num;
13:     std::cout << "input a number : "; std::cin >> num;
14:     std::cout << "the " << num << "th Fibonacci number is " << fib(num)
                                                                << std::endl;
15:
16:     return 0;
17: }
```

これは少々親切な表示をするプログラムである。画面の入力要求 “input a number : ” に対し、例えば 10 を入力すると、“the 10th Fibonacci number is 55” と返してくれる。以前のプログラムに較べて複雑な印象を受けるのは、親切な表示をしたためではない。関数 `fib()` のせいである。

プログラムの仕組みを説明しておこう。まず先に、11:行目からの `main` 関数を見ていくことにする。

`main(){}`は、`return 0;` を除いてわずか 3 行のプログラムでしかない。しかも、大部分は文字列の表示に費やされている。こんなんで正しい結果が得られるのは、陰でこっそりやることをやっているからである。

12:行目で、何番目かのフィボナッチ数を特定するための変数 `num` を用意した。今までは `int n;` などと書いてきたが、少々味気ないので意味が分かりやすい変数名にしてみただけだ。

13:行目では入力を促すメッセージを画面に表示し、`std::cin` により変数 `num` の値を取り込む。

その結果、14:行目の `std::cout` で “the *th Fibonacci number is **” のように画面に表示されるのである。では、** に表示される `fib(num)` の値は、一体どこでコソコソ計算しているのだろうか。

具体的に `num = 4` が与えられたとして処理を追ってみよう。処理は 3:行目からの関数 `fib()` の受け持ちだ。

13:行目で 4 の入力を受け取ったコンピュータは、14:行目の命令により、画面に “the 4th Fibonacci number is fib(4)” と表示したくなっている。

一方 fib(4) は、関数 `fib()` により計算がされる。このとき `fib()` が受け取る 4 は `n` の値として

受け取るわけで、それは条件文 `if` の `n > 2` を満たしているから、5:行目の `fib(n-1)+fib(n-2)` に代入され、`fib(3)+fib(2)` が返される。あれれ？　今は `fib()` の計算のはずだ。なのに `fib()` の計算に `fib()` 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。

`return` 文で `fib(3)+fib(2)` が返された結果コンピュータは、今度は "the 4th Fibonacci number is fib(3)+fib(2)" と表示する必要に迫られる。しかし、`fib(3)` は再び関数 `fib()` の 5:行目の処理により fib(2)+fib(1) にして返されてしまうのだ。

ところが、`fib(2)` のほうは `n > 2` の条件を満たさないので、こちらは 6:行目の `else` 文により `1` が返される。その結果コンピュータは "the 4th Fibonacci number is fib(2)+fib(1)+1" を表示しようとする。

しかしながら、まだ `fib(2)` と `fib(1)` は関数を呼んで計算を続けるので、さらに `fib()` により `1` と `1` が帰ってくる。これでようやくコンピュータは "the 4th Fibonacci number is 1+1+1" を表示すればよいことが分かり、結局 "the 4th Fibonacci number is 3" となるのだ。

一見すると詐欺にあったような印象を受けるだろう。隠れてコソコソ計算したくなるのも頷（うなず）けるというものだ。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。例えば `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。

TRY! 3:行目の `int fib(int n)` を `long fib(int n)` にして、`fib(47)` の計算をさせてみよ。`long` 型を使う理由は、`int` 型が 4 バイトの場合、関数 `fib()` が桁あふれを起こす。`int` 型が 8 バイトなら変更の必要はない。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)`, `fib(1)` は 2 つの `fib` 関数を呼ぶわけではないので、単純に `fib()` 計算が 2^{47} 回行われることはない。実際はもっと少なく、2^{31} 回ほどだ。一口に 2^{31} 回と言うものの、これでも軽く 20 億回を超えている。1 秒間に 10 億回の呼び出しが可能なコンピュータでも 2 秒はかかるので、“即座” に結果が出るわけではない。私たち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。