3.2 平方根

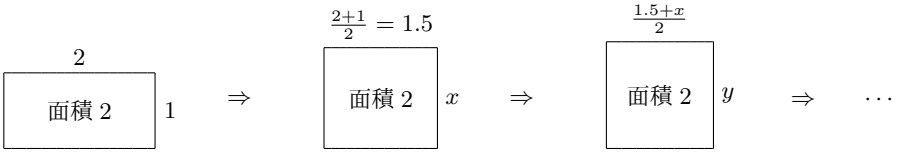
平方根の定義は『$x^2 = a$ を満たす x の値を "a の平方根" という』である。4 は、$4^2 = 16$ を満たすので 16 の平方根である。-4 も、$(-4)^2 = 16$ を満たすので 16 の平方根である。つまり 16 の平方根は 4 と -4 の 2 個で、まとめて ± 4 と書く。一方で $\sqrt{16} = 4$ であり、-4 は出てこない。**根号**の使い方を混同してないだろうね。

根号 $\sqrt{\ }$ は、a の平方根は$\dot{あ}\dot{る}$のだが正確な値として表す代わりに使う記号で、**正の平方根**に対して使うものである。負の平方根に対しては $-\sqrt{\ }$ を使う。だから、$x^2 = 17$ を満たす x は、4 より少し大きい値と -4 より少し小さい値であるのは確かでも、正確に表すことができないので $\sqrt{17}$ と $-\sqrt{17}$ が 17 の平方根となる。もちろん 16 の平方根を正確な値 4 で表す代わりに $\sqrt{16}$、$-\sqrt{16}$ としてよい。

こんなことは少し練習問題を解くうちに、意識しなくても区別できるようになるものだ。混乱するのは、定義をきちんと理解して積み上げる作業を怠るからである。要するに、表面的な解釈でパッパッとやるからいけない。おっと、なんだか自問自答しているみたいで嫌だな。

Swift は平方根の値を倍精度実数で提供してくれる。`sqrt(17)` と書くだけである。では、実際に $\sqrt{17}$ の**近似値**を計算するにはどうすればよいだろう。教科書的には、$4.1 \times 4.1 = 16.81$ は 17 に足りないので少し大きくする。$4.2 \times 4.2 = 17.64$ は 17 を超えるので 4.1 と 4.2 の間の値に違いない。刻みを細かくして $4.11 \times 4.11 = 16.8921$ ではまだ小さいので、次は $4.12 \times 4.12 = \cdots$ とやるのだろう。

しかし、平方根の求め方は他にもある。たとえば 2 の平方根を例に説明してみよう。2 の平方根が面積 2 の正方形の一辺の長さであることに着目した方法がある。



考えとしては、面積が 2 になるのは何も正方形に限らず 2×1 の長方形でもよいということだ。横の長さ 2、縦の長さ 1 では正方形から程遠いが、2 辺の平均である 1.5 を一辺に持つ長方形なら少し正方形に近づく。ただし 1.5×1.5 では面積が 2.25 になってしまうので、もう一辺の長さは $x = 2 \div 1.5 = 1.333\cdots$ として面積が 2 になるように調整する。

しかし $1.5 \times (1.333\cdots)$ では正方形ではないので、次は 1.5 と $1.333\cdots$ の平均 $1.416\cdots$ を一辺に持ち面積を 2 に保つために縦が $y = 2 \div (1.416\cdots) = 1.411\cdots$ である長方形にする。この繰り返し

だ。そうすると、2 辺の平均の長さと面積 2 を割り算した長さはどんどん近づいていく（本当は証明すべきことなんだが）。$\sqrt{2}$ は**無理数**だから、ふたつの値が一致して操作が終了することはないが、ある程度まで一致すればそれが $\sqrt{2}$ の近似値である。したがって、この手順をプログラムすればよいので、次のようになるだろうか。今回は機器の画面を使うこともないため、Playground の Debug Area に出力している。

[SquareRootOf2.playground]

```
 1 import UIKit
 2
 3 let S = 2.0
 4 var a = S
 5 var b = 1.0
 6 while abs(a-b) > 1/10000 {
 7     a = (a+b)/2
 8     b = S/a
 9 }
10 //
11 print(a)
```

面積 S は一定なので `let S = 2.0` で宣言し、長方形の 2 辺にあたる a、b には面積の値 S と 1.0 を代入している。小数値で代入しないと Swift は整数値の計算をしてしまい正しい結果が出ないことになる。`let S = 2.0` の直後に `var a = S` とするなら、初めから `var a = 2.0` だけでよさそうなものだが、それではまずいことはわかるね。

やっていることは、2 辺の平均値と、面積をその平均値で割った値に替えて、代入を繰り返すだけである。計算機の**丸め誤差**を考慮すると a と b が一致しない可能性があるので、`a == b` で判定せず、2 数の差が微小になるまで計算を行うことにしている。`1/10000` をさらに小さくすれば精度が高まる。

これはこれで問題ないのだが、代入を繰り返して同じことをするプログラムなら別の方法もある。**再帰**と呼ばれるアルゴリズムだ。

最初に例を示すことから始めよう。数学には**階乗**と呼ばれる計算がある。$n!$（n の階乗）は $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$ と定義されている。要するに n から 1 までの数を全部掛けるのだ。よって $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ だ。

じゃあ、6! は？　$6 \times 5 \times 4 \times 3 \times \cdots$、あ、待って！（←これは、待って×待っつ×待っち×$\cdots$にはならない。いや、そうじゃない。）5! はたったいま求めたじゃないか。ならば、$6! = 6 \times 5! = 6 \times 120 = 720$ でいいでしょう。律儀に 6 から 1 まで掛けなくてもよいのだ。つまり、$n!$ を知りたければ、ひとつ前の $(n-1)!$ を利用すればよい。Swift は次のようなプログラムで 10! の値を求めてくれる。

[Factorial.playground]

```
 1 import UIKit
 2
 3 func factorial(n: Int) -> Int {
 4     if n < 2 {
 5         return 1
 6     } else {
 7         return n * factorial(n: n-1)
 8     }
 9 }
10 //
11 print(factorial(n: 10))
```

factorial の名前で定義した関数の中に、同じ名前の関数が使われているので変な感じがすると思う。Microsoft Excel だったら自己参照をしている廉（かど）で罪に問われかねない。でも、Swift はまさに数学の定義通りに $n!$ を、n と $(n-1)!$ に分けて掛けている。これがきちんと動作するのは、factorial(n: n-1) がさらに (n-1)*factorial(n: n-2) に直るからだ。ただし、もし関数の中がこの 1 行だけならプログラムは終了しない。なぜなら factorial(n: 1) は 1*factorial(n: 0) に、factorial(n: 0) は 0*factorial(n: -1) に、... と延々続いてしまうからである。

プログラムが終了して結果を返すのは、factorial(n: n) を n*factorial(n: n-1) に直すのが n >= 2 のときに限るからである。だから factorial(n: 1) が 1*factorial(n: 0) になることはない。それは単に 1 になって、もう factorial の出番はないからだ。その結果、factorial(n: 10) は 10*9*...*1 を計算してくれる。

ex. 1 $n!$ を求める関数は for 文や while 文を使っても書くことができる。for 文か while 文を用いて $n!$ を求めるプログラムを書いてみよう。

[SquareRootOf2.playground] を再帰の考えを使って書き直したのが次のプログラムだ。

[SquareRootOf2.playground]

```
 1 import UIKit
 2
 3 let S = 2.0
 4 func compare(a: Double, b: Double) {
 5     if abs(a-b) < 1/10000 {
 6         print(a)
 7     } else {
 8         compare(a: (a+b)/2, b: S/((a+b)/2))
 9     }
10 }
11 //
```

```
12 compare(a: S, b: 1)
```

かえって面倒なことになっているようにも見えるね。慣れないうちは確かにそうだろう。でも面倒に見えても考えることが大事だ。いずれそれが自然になる。高校生にもなれば $f(n)$ のひとつ前が $f(n-1)$ で、$f(n)$ の次が $f(n+1)$ と書けることは自然に身につくものだ。このことは**数列**を学べばわかるし、ごく当たり前のことになる。

だから $n-1$ の階乗計算は $(n-1)!$ であるから、$f(n-1)=(n-1)\times(n-2)\times\cdots\times1$ と書けるのだ。このことと $n!=n\times(n-1)\times(n-2)\times\cdots\times1$ を合わせて考えれば、$f(n)=n\times f(n-1)$ になることが見える。そうなれば Swift の言葉に置き換えるのはたやすい。

では横の長さ a、縦の長さ b で面積が常に S の長方形に関してはどうだろう。今度は文字が 2 個あるので難しいかもしれないし、$f(n)=\cdots$ という式を作りにくいだろう。しかし「関数は式である」という固定観念は捨ててしまおう。この場合の関数は「現在の：（横の長さ）と（縦の長さ）」の比較である。そして次は「次の：（横の長さ）と（縦の長さ）」の比較となる。そして何を実行するかというと、横と縦の長さがほぼ等しくなったら横の長さを出力することだ。この関数において

次の：（横の長さ）　は　現在の：$\left(\frac{\text{横の長さ}+\text{縦の長さ}}{2}\right)$　に

次の：（縦の長さ）　は　現在の：$\left(\frac{S}{\frac{\text{横の長さ}+\text{縦の長さ}}{2}}\right)$　に

変化するのだから、横の長さ a、縦の長さ b、面積 S に対して

$$a\to\frac{a+b}{2}、\qquad b\to\frac{S}{\frac{a+b}{2}}$$

となっている。この関数表記が `compare()` なのである。ただ、階乗計算のものとはちょっと異なる書き方—`return` で値を返さない—をしているので戸惑うかもしれないけれど。

ところで再帰に慣れると、つい再帰を用いたくなることがあるかもしれない。再帰というのは、ひとつの関数を使い回すので計算量が**指数関数**的に増えるものである。下手に使うと、再帰を用いないプログラムより実行速度が遅くなるので注意が必要だ。また再帰の仕組みは、前へ前へと遡（さかのぼ）るように処理が進むので計算順序が特徴的だ。何のことを言っているのかわからなければ、`print(a)` を `compare(a: (a+b)/2, b: S/((a+b)/2))` の下の行にも書いて実行しよう。`a` の値が 2 から $\sqrt{2}$ へ近づく様子を見れば “遡る” と言った意味がわかるだろう。

ちなみに式 $S/((a+b)/2)$ は考え方に忠実に書いたが、これは $2S/(a+b)$ が自然な式だ。ただし、プログラムというのは後日見返すと意味不明になってしまうことがよくある。コメントを残す習慣はあったほうがよいだろう。

再帰に限ったことではないのだが、プログラムを汎用（はんよう）的にするか専属的にするかは考えどころである。[SquareRootOf2.playground] は専属的である。なぜなら、平方根の値を求めて出力することに特化しているからだ。平方根の値を十分な精度で計算する関数、という考えで書き直せば汎用的な関数になる。では、どういう風になっていれば汎用的と言えるだろう。この場合は return 文を用いて

```
func compare(a: Double, b: Double) -> Double {
    if abs(a-b) < 1/10000 {
        return a
    } else {
        return compare(a: (a+b)/2, b: S/((a+b)/2))
    }
}
```

とするほうが、まだ使いやすいが十分でないのは明らかである。なんで？

a 平方根の値を求める汎用的な関数とは、a を与えて $\sqrt{a}$ の値が返ってくるものである。たとえば sqrtOf(2) で 1.41421356 が得られるのが望ましい。compare() は引数に a = 2 と b = 1 が必要である。アルゴリズムがそうなっているからだが、計算の始まりは $a = S$、$b = 1$ なのだから本当は引数に b = 1 は必要ないのだ。

と言っても compare() の中で b = 1 から始めるわけにはいかない。それは、compare() が compare() を呼ぶのが再帰の仕組みなので、呼ばれるたびに b = 1 が実行されてしまうからだ。さらに、面積である S は一定の値として使われるものだが、これを compare() の引数に与えると再帰呼び出しのたびに値が変わってしまうのも困る。だから S は関数の外で定義しないといけない。

すると、この場合は再帰を用いてプログラムを書くのは無理なのだろうか。でも、大丈夫。たとえば sqrtOf(2) がプログラムの中で、S = 2 にした上で compare(a: S, b: 1) を計算する仕組みにすればよいのだ。

[SqrtOf.playground]

```
 1 func sqrtOf(a: Double) -> Double {
 2     let S = a
 3     func compFrom(a: Double, b: Double) -> Double {
 4         if abs(a-b) < 1/10000000 {
 5             return a
 6         } else {
 7             return compFrom(a: (a+b)/2, b: S/((a+b)/2))
 8         }
 9     }
10     return compFrom(a: S, b: 1)
11 }
12 //
```

```
13 let x = sqrtOf(a: 3)
14 print(x)
```

ほら、できた。関数の中で別の関数を定義して使うという荒技を使っているが反則ではない。そもそも関数 compFrom 自体が関数内で compFrom() を使っているわけだから、関数の中に関数が入っていることは問題ない。違いといえば、関数が関数の中で定義されるか外で定義されるかである。もっとも Swift は平方根を求める関数をすでに持っているので、sqrtOf() の使い道はないだろうけど。

ex. 2 再帰にこだわらず、また面積アルゴリズムにこだわらなくても、平方根を求めるプログラムを書くことができる。自分なりの工夫で平方根を求めるプログラムを書いてみよう。