3.6 三平方の定理

さあ、砂遊びはこれぐらいにして、そろそろ休憩を入れよう。砂浜の表面を少しさらっただけでも、いろいろな生き物に出会えたのではないだろうか。でも、これまでに見てきたものは Swift のほんの一部に過ぎない。もっと広く深く掘れば新しいことが山のように発見できる。そして砂遊びがより有意義に楽しめるというものだ。

中学校最後の話題は**三平方の定理**である。世間では**ピタゴラスの定理**などと呼ぶけれど、ピタゴラスが発見した定理というわけではないようだ。この定理はピタゴラスが発見する以前から知られていた模様だが、ピタゴラスによって一般に知られるようになったからだろう。でも定理の冠に怪しげな人名を使うのを良しとしない人が、“三平方の—” などという野暮ったい名称で呼んでいるのだ。しかし、数学で使われる名称には当人とは無関係の名称がつくことなんてザラにある。**フェルマーの最終定理**—フェルマーが証明した証拠はなく、アンドリュー・ワイルズが証明した。**ペル方程式**—この方程式の研究に功績があるのはブランカーらしいが、オイラーがペルと間違えた。などなど。

それでは最後に、三平方の定理に関するプログラムを書いてから一休みしようじゃないか。プログラムは辺の長さが 100 までの直角三角形を探すものである。

[Pythagoras.playground]

```
 1 import UIKit
 2
 3 for c in 1...100 {
 4     for b in 1...c {
 5         for a in 1...b {
 6             if a*a + b*b == c*c {
 7                 print("(a, b, c) = (\(a), \(b), \(c))")
 8             }
 9         }
10     }
11 }
```

もう、この程度のプログラムはお手のものだろう。`for` 文の入れ子になっているが、`for` 文に与える `a`、`b` の範囲に注意してほしい。ここをすべて律儀に 100 にすると、`(3, 4, 5)` も `(5, 4, 3)` も出力されてしまう。3 辺の比がわかれば、縦横がどっちになろうとかまわない。そのためには $a < b < c$ の条件を付ければよいので、プログラムの中で `if` 文を用いて分岐させることは必要ない。この例のように `for` 文の範囲を工夫すればよいのだ。

しかし、`(6, 8, 10)` は出力される。これは本質的に `(3, 4, 5)` の直角三角形なので、できれば見たくないものである。そのためには 3 辺の最大公約数が 1 でないものはハネてしまえばよい。

最大公約数を求めるコードは前に書いたけど、本当は**ユークリッドの互除法**を使うとプログラムが見やすくなる。

ユークリッドの互除法とは、2 数 m、n の最大公約数を求めるアルゴリズムで、手順は

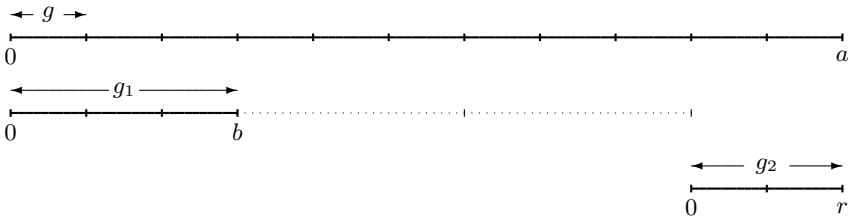
1. まず $m \div n$ を試してみる。割り切れたら n が最大公約数だ。
2. 割り切れなければ余り r が出る。
3. そこで今度は（割った数）n と（余り）r について同じこと、つまり $n \div r$ を試す。
   割り切れたら r が最大公約数だ。
4. 割り切れなければ次の余り s が出る。
5. そこで今度は（割った数）r と（余り）s について同じこと、...

と続けていくものである。この方法でなぜ最大公約数が求められるかは高 1 数学で学ぶはずだが、簡単に説明しておこう。

その前に具体例をいくつか見ておこう。やり慣れないうちは面倒に思えるものだが、(割られる数, 割る数) の組が割り切れる組になったとき、割る数が最大公約数ということだ。割る数に 1 が出てしまったら、2 以上の共通の約数はない。このような 2 数は**互いに素（そ）**と呼ばれる。

- $(30,\ 12) \xrightarrow{\text{余り}\ 6} (12,\ \underline{6})$
- $(30,\ 11) \xrightarrow{\text{余り}\ 8} (11,\ 8) \xrightarrow{\text{余り}\ 3} (8,\ 3) \xrightarrow{\text{余り}\ 2} (3,\ 2) \xrightarrow{\text{余り}\ 1} (2, \underline{1})$
- $(30,\ \underline{10})$
- $(30,\ 9) \xrightarrow{\text{余り}\ 3} (9,\ \underline{3})$
- $(30,\ 8) \xrightarrow{\text{余り}\ 6} (8,\ 6) \xrightarrow{\text{余り}\ 2} (6,\ \underline{2})$
- $(30,\ 7) \xrightarrow{\text{余り}\ 2} (7,\ 2) \xrightarrow{\text{余り}\ 1} (2,\ \underline{1})$

このアルゴリズムのよいところは、余りは割る数より小さくなるので、割り切れない事態が続いても次に割る数がどんどん小さくなる点である。小さくなる限界は 1 なので、必ず終わりは来るということだ。では、なぜ割り切れたところでその数が最大公約数になるのだろう。本当は式を立てて証明するんだけど、略図で勘弁してほしい。



a と b の本当の最大公約数を g とする。g は 1 かもしれないし 2 以上かもしれない。いずれにしても a と b は共通の目盛 g で等分できていることになる。a と b の最大公約数を求めるとき、最大公約数の最初の候補となるのが b 自身である g_1 だ。もし、a が b で割り切れるときは b が最大公約数 g_1 になって終了だ。

いまは割り切れなかったとする。割り切れないときの余りを r とすると、最大公約数の次の候補となるのは r である g_2 だ。ここで割り切れれば最大公約数 g_2 が見つかったことになる。互除法で余りを探す行為は、最大公約数の候補を定める行為でもある。そして r は b より小さく、r も共通の目盛 g で等分できているところがミソだ。

まだ割り切れなければ同じことを繰り返す。割り切れれば割った数（前回の割り算の余り）が最大公約数だが、割り切れなければ同じことがさらに続く。しかし、この作業は g が余りになったときに終了する。なぜなら g は共通の目盛だったので、g を余りにした最後の除数も g で割り切れるはずだからである。割り切れたときがアルゴリズムの終了の合図だったよね。

実は、ユークリッドの互除法は再帰を用いたプログラムに最適だ。被除数・除数・余りを入れ替えるだけで、やることは同じだからだ。ユークリッドの互除法を用いて [Pythagoras.playground] を書き直すとこんな感じになる。

[Pythagoras2.playground]

```
import UIKit

func findGCD(m: Int, n: Int) -> Int {
    let r = m % n
    if r == 0 {
        return n
    } else {
        return findGCD(m: n, n: r)
    }
}

for c in 1...100 {
    for b in 1...c {
        for a in 1...b {
            if a*a + b*b == c*c {
                if findGCD(m: a, n: findGCD(m: b, n: c)) == 1 {
                    print("(a, b, c) = (\(a), \(b), \(c))")
                }
            }
        }
    }
}
```

`findGCD()` だけ抜き出してみよう。見事に m、n の最大公約数を求めるアルゴリズムが再帰関数になっていることがわかるはずだ。

```
func findGCD(m: Int, n: Int) -> Int {
    let r = m % n
    if r == 0 {
        return n
    } else {
        return findGCD(m: n, n: r)
    }
}
```

`[Pythagoras2.playground]` では 3 数の最大公約数を求めたいのだが、あいにく `findCGD()` は 2 数の最大公約数を求めるものである。でも、`findCGD()` は関数にしてあるから、関数の中に関数を二重に使えば 3 数の最大公約数を求められるね。何事も工夫が大事である。

これで直角三角形を作る 3 辺の組が、重複も漏れもなく見つかるだろう。実は、3 辺が整数であるピタゴラス三角形は古くから調べられていて、3 辺の値が

$$m^2 - n^2, \quad 2mn, \quad m^2 + n^2 \qquad ※$$

である三角形に限ることが知られている。たとえば 3, 4, 5 の直角三角形は、$m = 2$, $n = 1$ のとき、5, 12, 13 の直角三角形は $m = 3$, $n = 2$ のときなどだ。このことを知っていれば、何もプログラムを書いて調べる必要はないのである。

ex. 1 ※を利用してピタゴラス三角形の一覧を出力するプログラムを書いてみよう。一覧を見て気づくことはあるだろうか。

実際のところコンピュータは処理が速く、同じことをいつまでも続けてくれる。時間さえあれば間違いなく解にたどり着くはずだ、と考えるのは大間違いである。たとえば一辺の長さが 3 の直角三角形は 3:4:5 のものに限る。これは `[Pythagoras2.playground]` の結果からわかる。本当にそうだろうか。`[Pythagoras2.playground]` は 100 以下の辺の長さの直角三角形をすべて調べた。でも、ちょっと待って。調べたのは “3 辺すべてが” 100 以下の三角形である。

もう少し先を調べると (15, 112, 113) という直角三角形が見つかる。やけに平べったい三角形だね。それなら (3, 数百, 数百 $+ \alpha$) なんてのが見つかるかもしれないね。でも、その答えはコンピュータは教えてくれない。しかし、(3, 数百, 数百 $+ \alpha$) なんて直角三角形が決してできないことを示すことはできるのだ。こんな説明でどうだろう。

3 辺 $(a,\ b,\ c)$ が $a^2 + b^2 = c^2$ のとき、$a^2 = c^2 - b^2 = (c+b)(c-b)$ である。このとき $c - b = \alpha$（数百と較べてかなり小さい）であれば $a^2 \approx c + b$ と見てよいので、$a \approx \sqrt{c+b}$ になる。b, c が数

百のように大きい数なら、$a=3$ なんてことはあり得ない。$a=3$ なら $c+b\approx 9$ でなければならないのだから。

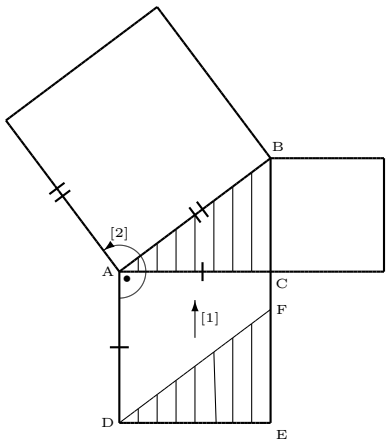
結局のところコンピュータやプログラムを過信してはいけないのだ。プログラムにはプログラムの役割がある。純粋に数学で考えることとプログラムで解決することは、互いに補完し合うのがよい。Swift はアニメーションも比較的簡単にできる。アニメーションは紙にはできないことなのでプログラムを書く価値は高いだろう。

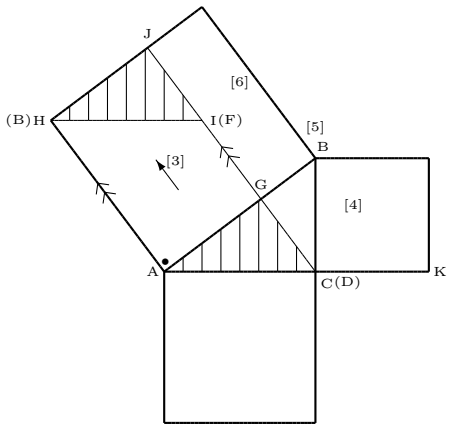
たとえばピタゴラスの定理は “定理” と呼ばれている以上、証明抜きには語れない。※がピタゴラスの定理を満たすことは

$$(m^2-n^2)^2+(2mn)^2=(m^2+n^2)^2$$

であることを確認すればわかることである。しかし、逆に $a^2+b^2=c^2$ を満たす組 $(a,\ b,\ c)$ が $(m^2-n^2,\ 2mn,\ m^2+n^2)$ であることを示すのは難儀するはずだ。でも、こういうことは実際に式をこねくり回さないと導けないから、コンピュータの出番はないのである。

コンピュータの効果的な出演どきは、大量の処理をするときと視覚に訴えるときだと思う。ピタゴラスの定理の証明方法は、数十種類が知られているようだ。そのうちの多くは面積に着目したもので、アニメーションを見ることができれば一目瞭然なものが多い。





図で △ABC の周りに描かれた 3 個の正方形の面積がそれぞれ a^2, b^2, c^2 を表している。目標は小さい 2 個の正方形の面積の和が、斜辺の大きい正方形の面積に等しいことを示すことである。

[1] まず □ACED の辺 CE をスライドさせて BF へ移動する。▱ABFD は同じ面積のはずだ。

[2] ▱ABFD を A を中心に回転させて右図の ▱AHIC とする。これはぴったり重なる。

[3] ▱AHIC の辺 IC をスライドさせて JG へ移動する。これで □ACED は □AHJG へ移った。

以上の変化は紙の上で見るより iPad などでシミュレーションできれば効果的にちがいない。

ex. **2** いちばん小さい正方形が移動する様子 [4]→[5]→[6] を頭の中でシミュレーションしてみよう。きちんと納得できただろうか。

さて、中学数学の教科書の目次を眺めながら、Swift と一緒に数学のほんの一面をのぞいてきた。でも今日はここまで。潮もだいぶ満ちてきたようだし、陽も傾いてきた。続きは家に帰って取り組んでみよう。それが自分一人が楽しむものであれば、習熟するにつれプログラムがどんどんマニアックになってもかまわない。でも、誰かと共同で何かのプログラムを書くのだとしたら、マニアックなコードはもってのほかだ。自分より技能が伴わない人にも修正できるような平易なプログラムを書こう。自分だけでなく、周りの人にも快適なプログラム生活を送ってほしいから。

ex. **3** ピタゴラスの定理のシミュレーションを見せるプログラムを書いてみよう。大変な作業かもしれないが、平行移動と回転のプログラムを組み合わせればよさそうだ。これまでに見たプログラムを少しずつ修正し、壁に当たったらネットで調べればよい。大丈夫。きっとできるよ。