

4.4 偶数の分解と素因数分解

それではゴールドバッハの予想を確認してみよう。これまでにスクリプトの蓄積があるので、比較的書きやすいはずだ。twinp 関数を少し手直ししておこう。twinp 関数では $6m \pm 1$ の二つの数の素数判定をしたので、その部分を替えるだけで済むだろう。つまり、偶数を二つの奇数に分けたとき、それらの素数判定をするように組めばよいだけだ。

まず前節の pchk 関数『

```
\begin{luacode*}
function pchk(n)
  i = 3
  while i <= math.sqrt(n) do
    if n % i == 0 then
      return false
    end
    i = i + 2
  end
  return true
end
\end{luacode*}
```

』を再掲しておく。今節を前節とまとめて処理するなら不要だが、いまはこの節だけを処理するのに必要となった。肝心のゴールドバッハ和を出力する関数は『

```
\begin{luacode*}
function goldbach(n)
  r = 3
  while r <= n // 2 do
    if pchk(r) and pchk(n - r) then
      tex.print('(' .. r .. ', ' .. n - r .. ')')
    end
    r = r + 2
  end
end
\end{luacode*}
```

』である。これで『\directlua{twinp(100)}』と書いて『(3, 41) (7, 37) (13, 31)』が出力される。

スクリプトは何のことはない。pchk 関数は以前の借用である。このスクリプトは、与えられた偶数を素数の和に分解するものである。与える偶数の上限を決めて、そこまでの偶数の分解の一覧

を表示させるには、もう一つ for 構文を必要とする。それは、君たちの課題にしておきたい。

では、goldbach 関数を見よう。

r は、for 構文で n を二つの素数に分解するとき、一方の素数を代入するための変数になる。では、もう一方の素数を代入する変数は用意しなくていいの？ そう、その必要はない。なぜなら、もう一方の素数は $n - r$ だからだ。スクリプトには不必要な変数は使わないほうがよろしい。

and 文では双子素数と同様の処理をしている。r と $n - r$ が共に素数かどうか調べているのだ。共に素数のときに限り判定は真となる。調べる範囲が $r \leq n // 2$ であることに注意してほしい。もちろん、調べる範囲が $r \leq n // 2$ で十分な理由は分かるね？

そして見事素数の和に分解されたとき、`tex.print` 文によってペアで表示される。この関数は、入力された偶数に対して確実に素数の和に分解してくれるだろう。それも、すべてもれなく表示してくれる。ちょっとだけ困るのは、 $4 = 2 + 2$ を示すことができない点だ。 $4 = 2 + 2$ の表示は自明のことだから、かまわないといえにかまわないのだけれど。

さて、これで 6 以上の偶数の分解がコンピュータ任せにできた。ところが 6 以上の偶数に対しては問題ないのだが、スクリプトには致命的な欠陥があるのだ。それは、うっかり奇数を入力してしまった場合に表面化する。奇数 n が入力されると、スクリプトはこれを 3 以上の奇数 r と $n - r$ に分けて素数判定に回す。しかし $n - r$ は偶数なのだ。

このときは大変困ったことになってしまう。というのは、関数 `pchk(n)` は奇数を受け取ることしか想定してないからだ。`pchk(n)` は受け取った数を、3 から先の奇数で順次割り算を試して、どうにも割れないときに素数と判定する。ところが偶数を受け取った場合、3 から先の奇数で順次割り算を試しても、どうにも割れない数がある。たとえば 8 がそうだ。すなわち、偶数が素数と判定されてしまうのだ。

これを回避するには、偶数が入力されたときだけ処理をすればよい。それには、`if` 文を一つ追加すれば十分である。

```
if n % 2 == 0 do
  (字下げして以下同様)
end
```

これで奇数入力の危機は一応去る。しかし、スクリプトは 6 以上の偶数を入力しないと正しく機能しない。2 や 4 が入力されては困る。それを回避するには、もうちょっとコードを書き加えなく

てはならない。これは君たちに任せよう。

ゴールドバッハの予想の路から少し外れるけれど、せっかく素数に馴染んできたところなので、素因数分解の小路に入ってみたい。素因数分解とは、ある数を素数の積で表すことである。具体例は $100 = 2 \cdot 2 \cdot 5 \cdot 5$ や $365 = 5 \cdot 73$ などである。当然、素数は素因数分解できない。

素因数分解をするには候補となる数を次々と素数で割っていき、割ることができた素数の積で構成すればよい。たいていのプログラミングでは、

$$p_0 = 2, \quad p_1 = 3, \quad p_2 = 5, \quad p_3 = 7, \quad \dots$$

のような素数の一覧表を用意して、候補となる数を p_i で順次割ることになる。

このように素数表を用意すれば、余計な除数で割り算をする無駄が省けて、大変効率的である。しかし、素数表をどこまで準備しておくかが重要な問題である。そこで、ここでは少々効率が悪い割り算をするが、素数表を用いない小さいスクリプトを書いてみた。それが『

```
\begin{luacode*}
function intfactor(n)
  i = 2
  while i <= math.sqrt(n) do
    if n % i == 0 then
      tex.print(i .. ' * ')
      n = n // i
    else
      i = i + 1
    end
  end
  tex.print(n)
end
\end{luacode*}
```

』という関数で、『`\directlua{intfactor(100)}`』を処理すると『`2 * 2 * 5 * 5`』が出力され、『`\directlua{intfactor(17)}`』を処理すると『`17`』が出力される。

全体的に見なれたコードだろう。スクリプトは基本的に与えられた数を 2 から順に割り算のテストにかけるだけである。

割り算テストで最初に試す数は $i = 2$ であり、最後に試す数は $\text{math.sqrt}(n)$ までで十分である。なぜなら、 i は 1 ずつ増やしているのだから、エラトステネスの篩の理屈によって、 \sqrt{n} より大きい i で割れる数は生き残った素数である。

n が i で割り切れたときは、その i が因数であるから `tex.print` 文で表示すればよい。ちょっと見栄えを良くするために、続けて `*` も表示させている。`tex.print(i .. ' ')` で空白を入れながら表示されたので、`tex.print(i .. ' * ')` なら `*` を入れながら表示される。具体的に 50 が与えられれば、この時点で `2 *` が表示されるはずだ。そして、 n を i で割った数を新しい n としてテスト続ける。割り切れないときは、 i の値を 1 ずつ大きくして割り算テストをする。本当は、 i は 3 から始めて 2 ずつ大きな数で割るほうが効率が良いのだが、そこまで効率にこだわっていない。

このことを繰り返していくと、最後に 2 より大きな数で i で割り切れない数が n に残る。それが最後の因数であるから、そのまま `tex.print` 文に回せば素因数分解の完成である。蛇足ながら `n = n // i` が `n = n / i` ならば `tex.print(n)` で出力される数値に小数点がつく。