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。これまでにマクロの蓄積があるので、比較的書きやすいはずだ。マクロ\twinp を少し手直ししておこう。\twinp では $6m \pm 1$ の 2 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 2 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。

```
\newif\ifprime \newif\ifunknown
\newcount\n \newcount\p \newcount\d \newcount\a
\def\testprimality{{\d=3 \global\primetrue
  \loop \trialdivision \ifunknown \advance\d by2 \repeat}}
\def\trialdivision{\a=\p \divide\a by\d
  \ifnum\a>\d \unknowntrue \else \unknownfalse \fi
  \multiply\a by\d
  \ifnum\a=\p \global\primefalse \unknownfalse \fi}
%
\newcount\q \newcount\r \newcount\cntr
\newcommand\goldbach[1]{%
  \q=#1 \r=3 \advance\q-\r
  \loop \ifnum\r>\q\else \p=\r \printifprimes
  \advance\r2 \advance\q-2 \repeat}
\newcommand\printodds{\ifnum\cntr=0\else, \fi(\number\r, \number\q)}
\newcommand\printifprimes{\testprimality
  \ifnum\r=3 \primetrue \fi
  \ifprime \p=\q \testprimality
  \ifprime \printodds\advance\cntr1 \fi\fi}
```

これで、『44 のゴールドバッハ和は\goldbach{44}である。』と書けば『44 のゴールドバッハ和は (3, 41), (7, 37), (13, 31) である。』が出力される。ゴールドバッハ和などと大層なことを言ってみたが、この蝶道だけの言葉遣いである。

マクロは再度\testprimality、\trialdivision の借用である。以前のマクロを使い回せるのはよいことだが、そのために制限に振り回されることもある。今回はそんな例だ。このマクロは、与えられた 1 個の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出力させるには、もうひとつ\loop〜\repeat が必要なはずだ。それは、君たちの課題にしておきたい。

では、マクロ\goldbach を見ていこう。

変数\q と\r は 2 つの素数に分解するとき、それらを代入するための変数になる。あとは\q と\r がともに素数であることが判定できれば終了である。ただ、ちょっとだけ工夫をしている。まず、偶数を分解するときは、たとえば 44 なら $3 + 41$ から調べればよいから\r=3 が初期値である。これはどんな偶数でも同じだ。そして引数から\r を引いたものが\q となる。ところで 44 を 2 個の

奇数に分解する場合、$3+41$ を調べたら $41+3$ は調べる必要がない。どこまで調べればよいかというと、たとえば $22=11+11$ の例からも分かるように、$r \leqq q$ の場合まで調べればよいことになる。しかし、残念ながら TEX のマクロでは大小関係は “<” でしか調べられなかった。そこでトリックとして、$r>p$ の場合に何もせず、それ以外の場合を調べるようにしている。それが`\ifnum\r>\q\else ...` と判定する理由である。これで、たとえば 22 の場合でも $11+11$ はテストにかけられることになった。

`\printifprimes` にも、余分な`\ifnum\r=3` の判定が挿入されていることに気づいただろうか。これが必要な理由は`\testprimality` をそのまま借用したからだ。というのは、`\testprimality` は素数の判定をする関数だが、これは 5 以上の奇数を判定することしか想定していない。しかし偶数の分解は $3+q$ から調べ始める。`\testprimality` は 3 を受け取ると、それを素数とみなしてくれないのだ。よって、$3+q$ を正しく判定するために余分な 1 行が加えられた。ただし、$6=3+3$ の分解は正しく判定できない。こんな自明なことはマクロで調べるまでもないからである。どうしてもマクロで正しく出力させたければ、次の`\ifprime` の前にも`\ifnum\r=3` の判定文を挿入しなくてはならない。たったひとつ、$6=3+3$ を出力させるためにすることじゃないね。おっと、あとひとつ、$4=2+2$ も出力できないんだ。でも、自明のことだから構わないだろう。

さて、マクロは偶数を見事素数の和に分解したとき、`\printodds` によってペアで出力される。しかし、ここでも、余計な手間が必要だった。それは、どこですべてのペアが出尽くすか分からないからである。出力は “,” で区切って示したいが、常に “,” 区切りで出力すると最後にまぬけな “,” が残ってしまう。かと言って、最初に付けるといの一番に “,” が出力されてしまう。仕方ないので、`\cntr` で最初の “,” が出力されないように制御することにしたのである。まったく手間がかかったが、このマクロ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はずである。

さて、これで 6 より大きい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、マクロ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 r が入力されると、マクロはこれを 3 以上の奇数 r と $q-r$ に分けて素数判定に回す。しかし $q-r$ 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、`\testprimality` は奇数を受け取ることしか想定してないからだ。`\testprimality` 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、`\ifodd` をひとつ追加すれば十分である。これで奇数入力の危機は一応去る。しかし、マクロは 6 より大きい偶数を引数に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道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**の蝶道にも入ってみたい。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,  p_1 = 3,  p_2 = 5,  p_3 = 7,  ...`

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、素数表を用いないマクロにしてみた。

```
\newcommand\intfacto[1]{%
  \newif\ifdivtry
  \newcount\n \newcount\d \newcount\s \newcount\p
  \n=#1 \d=2 \divtrytrue
  \loop \ifdivtry
    \s=\n \divide\n\d \p=\n \multiply\n\d
    \ifnum\s=\d \divtryfalse \else
    \ifnum\s=\n \n=\p \number\d*\else
    \n=\s \advance\d1 \fi\fi \repeat
  \number\d
}
```

これで、『$100 =$ `\intfacto{100}`』と書けば『$100 = 2 * 2 * 5 * 5$』が出力される。素数に対しては、『$17 =$ `\intfacto{17}`』と書けば『$17 = 17$』が出力される。

マクロ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`\d=2` であり、最後に試す数は $\sqrt{n}$ までで十分である。なぜなら、整数 n が $\sqrt{n}$ より大きい数で割れるときは、それ以前に $\sqrt{n}$ より小さい数で割れているからだ。しかし、平方根を求めるルーティンを作っていないので、余分な数でも割り算を試している。

`\n` が`\d` で割り切れたときは、その`\d` が因数であるから`\d` を出力すればよい。ちょっと見栄えを良くするために、続けて "*" も出力させている。引数に 100 を与えれば、この時点で "$2 *$ " が出力されるはずだ。そして、`\n` を`\d` で割った数を新しい`\n` としてテストを続ける。割り切れない

ときは、\d の値を 1 ずつ大きくして割り算テストをする。本当は、\d は 3 から始めて 2 ずつ大きな数で割る方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、最後にちょうど\s と同じ値数が\d に残る。それが最後の因数であるから、そこで処理を終了し、最後に\d だけを出力すれば素因数分解の完成である。

余談ながら素因数分解ができると、平方数の簡略化に役立てられる。以下のマクロ\expandrt は、\intfacto にコードを少々追加して、平方数を開平して表示するようにしたものだ。

```
\newcommand\expandrt[1]{%
  \newif\ifdivtry
  \newcount\n \newcount\d \newcount\s \newcount\p
  \newcount\a \newcount\b \newcount\t
  \n=#1 \d=2 \divtrytrue \a=1 \b=1 \t=1
  \loop \ifdivtry
    \s=\n \divide\n\d \p=\n \multiply\n\d
    \ifnum\s=1 \divtryfalse \else
    \ifnum\s=\n \n=\p
      \ifnum\t=\d \multiply\a\d \t=1 \else
      \multiply\b\t \t=\d \fi \else
    \n=\s \advance\d1 \fi\fi \repeat
    \multiply\b\t
  \ifnum\b=1 \number\a \else
  \ifnum\a=1 \sqrt{\number\b} \else
  \number\a\sqrt{\number\b} \fi\fi
}
```

これで、『$\sqrt{24}$, $\sqrt{25}$, $\sqrt{26}$, $\sqrt{27}$, $\sqrt{28}$はそれぞれ、$\expandrt{24}$, $\expandrt{25}$, $\expandrt{26}$, $\expandrt{27}$, $\expandrt{28}$である。』と書けば『$\sqrt{24}$, $\sqrt{25}$, $\sqrt{26}$, $\sqrt{27}$, $\sqrt{28}$ はそれぞれ、$2\sqrt{6}$, 5, $\sqrt{26}$, $3\sqrt{3}$, $2\sqrt{7}$ である。』が出力される。