イプシロン　2017. Vol.59, 7 − 18

ICT を利用した算数・数学での探究のサイクルについて
ー完全数などについての探究事例を手がかりにー

愛知教育大学　数学教育講座　**飯　島　康　之**

1.　はじめに

1.1 ICT を利用した探究を実践する上で出会う「壁」

今年の附属高校でのシンポジウムでは，二つの教科(理科・体育)で公開授業を行った。その中の理科(物理)では，距離センサーを使った実践に取り組んだ。11/8 の公開授業にいたるまで,4 回にわたって，距離センサーを使った実践を深めて行ったが，特に最初の取り組みなどを拝見すると,ICT を利用した実験あるいは探究に共通の「壁」を実践者(足立先生)が感じていることがよくわかった。

その「壁」とは何かというと，通常の授業の感覚で機器を与え，やり方を説明し，実践していくと，「考えさせない実践」になってしまうことである。算数でのもっとシンプルな例でいえば，計算問題をするための道具として，電卓を与えるとする。数や演算の入力の仕方を説明した後，通常の計算問題として与えた課題に取り組むとすると，子どもに求められるのは，問題に書かれている数値と演算を「正しく入力し，得られた結果を正しく転記すること」であって，そういう機械的な作業に，「数学的な思考」の入る余地はほとんどない。そういうレベルにとどまるのであれば，そういう機器を使わないで取り組むときに想定している教育目標を阻害するだけのことなので，そのような ICT 機器は使わない方がまし，ということになってしまうのである。

便利な道具を使うことによって，今までできなかったことを思考の対象とし，今までとは異なる「思考のプロセス」が可能にならない限り，実践する価値はない。しかし，そこで求められる「今までの道具ではできなかったこと」を意識化し，焦点を当てるのは簡単ではない。これが，実践者にとっての「壁」である。

この壁を認識し，「その道具を使うからこそ実現可能な探究」を考えることが大切だということを実感して初めて,ICT を利用した探究に関する教育実践が地に足がついたものになっていくのではないだろうか。

1.2 探究における「サイクル」という考えと本論文のねらい

このような探究の様相を表現するための，一つのキーワードは「サイクル」ではないだろうか。問題を解決するにあたって，とても強力な道具を一度使うことで解決するのではなく，問題に対して，まずある試みを行い，その結果を解釈・検討し，次の探究を行い，さらに，... と，それぞれは小さな一歩かもしれないけれども，少しずつ前進していくことによって，最終的に最初の問題が解決されたり，あるいは途中で別の問題に発展し，その問題が解決されるというような，探究

のサイクルが円滑に進展していくのを支援する道具として，ICT が位置づけることができるのではないだろうか。たとえば，いわゆる「アクティブラーニング」という言葉は，「主体的・対話的で深い学び」という言葉に置き換えられ，使われているが，上記のようなサイクルを想定すると，ここでの「主体的，対話的，深い」という三つのキーワードはとてもしっくりと理解される。

また，たとえば学習指導要領解説(数学編)では，次の図が算数・数学の問題発見・解決の過程として占められているが，ここでも，現実の世界あるいは数学の世界との関わりの中でのサイクルが表現されている。

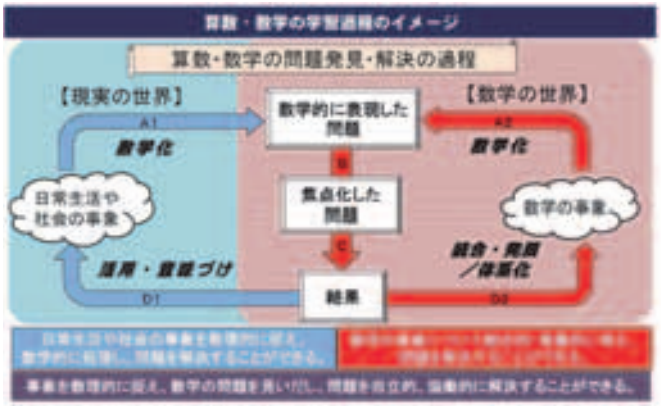

図-1 算数・数学の学習過程のイメージ(学習指導要領解説(数学編)より)

私自身，これまで GC を使いながら図形に関する探究や教材開発を中心に扱ってきた。今回，ラズベリィパイという教育用コンピュータを使って生徒がどのような探究が可能かを検討することを目的として，整数の問題について取り組んでみた。その概要を素描し，検討することを手がかりとして，数学的探究のサイクルについて考察してみたい。

2. ラズベリィパイのインパクト

2.1 教育用コンピュータとしてのラズベリィパイ

昨年から注目し，取り組んでいるものに，ラズベリィパイという名の，教育用コンピュータがある。iPad などのタブレットが，コンピュータとしての存在感を消し，直観的なインターフェイスや操作によって「便利な道具」として教育などに浸透しているのと対照的に，ラズベリィパイは，それがコンピュータであることを意識し，いろいろなことを学ぶための教材として機能している。Linux の仕組みを理解したり，プログラミングを学んだり，GPIO を利用してさまざまなセンサのデータを取得・活用したり，サーボモータを制御するなど，これからの STEM 教育を推進していくための強力な教具の一つである。

ラズベリィパイ 3B で本体価格 4500 円程度，ラズベリィパイゼロにいたっては 5 ドル(650 円程度)と，いずれプラモデルくらいの感覚で扱えるところも魅力的であり，さまざまな観点から，算数・数学教育での利用可能性を探り始めていた。

2.2 ラズベリィパイに搭載されている，算数・数学を追求するための標準的なソフト

ラズベリィパイには，pilot 版の mathematica が搭載されていて，無償で使えることは，まず注目すべき点である。これまでも，高校数学などでの mathematica の教育利用も研究されてきたが，実際に教育現場で使うとなると，経費は大きな障壁で，研究に関わる人々もかなり限られてきた。そして，もう一つ注目されるのは，Python である。Python は人工知能なども含めて非常に幅広い利用ができるので，今とても注目を集めているプログラミング言語の一つであるが，初心者が接しやすい言語でもある。少なくとも私たちが BASIC に接したのと同じくらいの感覚で使うことができる言語である。さらに，特に数学的探究にとっては，任意多倍長の整数を扱えることができる点に特徴がある。つまり，事前に宣言することなく，必要に応じた桁数まで拡張しながら，使えることが保証されている。たとえば，2^1000 のような計算をしてみると，特別な宣言をしなくても，次のように正確に表示され，とても大きな整数であっても正確な実験が可能である。

```
>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

2.3 ラズベリィパイで実験してみる素材としての「完全数」

そこで，算数でも理解できるくらいの素材に注目し，実際にラズベリィパイ(および搭載されているソフト)を使って探究してみて，どういうことに出会い，どういうことを考え，そしてどういう解決がなされ，どういう問題が発見されるのか，そういうことを大学院の授業の一環として，体験してみることにした。取り上げた素材は，「完全数」である。

3. コンピュータを使った，完全数などに関する数学的探究

3.1 完全数の定義と最初の探究(探究 1)

まず，完全数を，次のように定義しておこう。

定義：完全数とは，自分自身を除く正の約数の和と等しくなる自然数のことをいう。

完全数にはどんな数があるかを調べたいというのが素朴な最初の探究であろう。たとえば，次のような(ここでは Python3 で記述)プログラムが，それを調べるためのものになる。

```
for n in range(1,100000):
    sum=0
    for i in range(1,n):
        if ((n // i)*i == n):
            sum = sum + i
    if (n == sum):
        print(n)
```

出力される結果は，6，28，496，8128 の後，ほぼ何も反応がない時間がつづく。実際，次の完全数は，33550336 とのことなので，「この 4 個以上のことはいくら時間をかけても，全然進展しない」状況になる。そこで，探究の方針を少し変えてみることにした。

3.2 問いを変える（対象を広げプロセスを観察するために，「ある数列の性質」を調べる）（探究 2）

完全数は「自分以外の約数の和」が自身と等しい。逆に言えば，ほとんどの場合，違う数になる。その数の約数の和が，最初の数に戻る場合，この二つの数は友愛数と呼ばれる。二つではなく，三つ以上の数がチェーン状になっているとき，これらの数は社交数と呼ばれる。とすると，完全数だけではなく，友愛数や社交数も含めて，一気に調べてみることはできないかと考えた。

完全数を判定するときに使うのが，「自分以外の約数の和」である。自然数 n に対して，新しい整数を対応させるものとして「自分以外の約数の和」という関数を考えるとすれば，完全数というのは，この関数において，自分自身が自分自身に対応するものとして理解できる。

また，友愛数や社交数もあるが，次のような数列を考えてみることで，数列の特徴を調べることで，完全数などについて調べることにつなげてみようと考えた。

a(1)=n, a(i+1) = a(i)の自分以外の約数の和　（i = 2,3,4,…）

すると，完全数，友愛数，社交数などは，次のように調べることができることになる。

a(1)=n のとき，a(2)=a(3)=a(4)=… となり，n は完全数になる。

a(1)≠a(2), a(3)=a(1) のとき，a(1)と a(2)は友愛数になる。

a(m)=a(1) で，1 ≦i< j ≦ m で，これ以外のときには a(i)≠a(j)，のとき，a(1),a(2),…a(m)は社交数になる。

初期値 n に対して，完全数，友愛数，社交数を求めることを想定しつつ,数列 a(i)がどのような値をとるかを観察してみることに，問いを変えることにした。

次のプログラムでは，初項が 1 から 100 までについて，10 項までを調べている。

プログラム 1

```
def SumOfDevisors(n):
    sum=0
    for i in range(1,n+1):
        if n%i==0:sum=sum+i
    return sum

for n in range(1,100):
    print (n, end=",")
    for i in range(0,10):
        n= SumOfDevisors (n) - n
        print (n, end=",")
    print ("")
```

主な結果例は次の通りである。

1,0,0,0,0,0,0,0,0,0, 2,1,0,0,0,0,0,0,0,0, 6,6,6,6,6,6,6,6,6,6,

12,16,15,9,4,3,1,0,0,0, 25,6,6,6,6,6,6,6,6,6,

28,28,28,28,28,28,28,28,28,28, 30,42,54,66,78,90,144,259,45,33,15,

この結果を観察から次のことが分かった。

・数列の値が素数になると，次の値は 1 になって収束する(かなり多い)。

・完全数なら同じ数の数列が生まれ，友愛数なら循環の長さが 2 の循環数列に，社交数なら長さが 3 以上の循環数列になる。

・循環の出発点は最初の項とは限らない。

・数列の値は小さくなるとは限らなくて，変動する場合もある。

そして，次に注目したいこととして次のことが挙がった。

・循環する数列(完全数，友愛数，社交数)をさらに求める。

・(n=30 のように)10 項まででは収束するのか，循環するのか，あるいは発散するのかがわからない場合もあるが，調べる項数を増やしながら，項数が多い n を調べてみる

3.3 1 から 10000 までの n に対して，10 項まで調査してみる(探究 3)

完全数なら第 2 項まで，友愛数なら第 3 項までで十分なわけだが，社交数に関しては，循環することが分かれば，いくらでも長い鎖のものはありうるわけだが，10 までの長さと割り切り，10000 までについて調べてみた。その結果の概要は次の通りである。

サイクルになる	193 個
1 に収束する	5478 個
まだ続くのでわからない	4569 個

この中で，サイクルになるものには次のようなものがある。

(1) 完全数 : 6,28,496, 8128

(2) (95→25→6 のように)完全数に帰着される : 25, 95, 119, 143, 417,445,650, 652 など

(3) 友愛数 : 220 と 284, 1184 と 1210, 2924 と 2620, 5564 と 5020

(4) (562→284→220→284 のように)友愛数に帰着される : 562, 1064, 1188 など

(5) 社交数に帰着される : 9464→ (12496, 14288, 15472, 14536, 14264)

ここでの結果は，10 項までに収束あるいは，循環していることが判定できる場合をもとにしているが，約半数は，その判定ができていない。さきほどの n=30 の場合なども含めて，どれくらいの項まで調べれば，数列の挙動を知ることができるのか，項数を増やしてみることにした。

3.4 初項の値 n がどんな場合に収束するまでの項数が多くなるかを調べてみる(探究 4)

まず，n=100 まで，20 項まで調べてみた。さきほどの n = 30 に関して次のような結果を得た。

30,42,54,66,78,90,144,259,45,33,15,9,4,3,1,(第 15 項で 1 に収束)

n=100 までで，1 に到達するのが 10 項以上になったものは，次である。

n=6 (循環，完全数)，25(循環，6 に帰着)，28(循環，完全数)，30(15 項)，42(14 項)，54(13 項)，

60(11 項), 66(12 項), 78(11 項), 90(10 項), 95(循環, 6 に帰着)

観察からわかることは, 次の通りであった。

・循環するケースは 6, 25, 28, 95 があり, 自身が循環する完全数の場合(6, 28)と, それらに帰着する場合(25, 95)がある。

・10 項以上になるものは, 循環するものを除くとすべて 6 の倍数である。

さらに, n=200 まで, 20 項まで調べてみた。

10 項以上になる n は, 102, 108, 114, 118, 120, 126, 132, 138, 143(6 に帰着), 148, 150, 152, 160, 166, 168, 174, 180, 186, 190, 192, 196 であり, 必ずしも 6 の倍数とは限らない。

驚いたのは, 138 の登場である。

138,150,222,234,312,528,960,2088,3762,5598,6570,10746,13254,13830,19434,20886,21606,25098,26742,26754, と, 20 項以上続く。このように, 20 項以上続く n(n≦200)は, 138, 150(138 の系列), 168, 180, の 4 つがある。これらは, 6 の倍数である。

3.5 長く続くのは 6 の倍数の場合なのか(探究 5)

初項の値が大きくなると, 項数が増えるのは当然のことともいえるので, 1000 以下で 100 項を越えるものを調べてみると, すべて 6 の倍数に限定された。しかし, さらに調べると, 1372 など,6k+4 の形のものや, 1521(6k+3), 1316(6k+2), 3025(6k+2)なども見つかり, 6 の倍数にあまりこだわっても仕方がないのではないかという手応えを得た。

3.6 n=138 の場合を解決するためのアルゴリズムの改良(探究 6)

138 に限定して, プログラム 1 のアルゴリズムで調べてみた。8 時間ほど計算させて到達したのは, a(115)= 40381357656。現在の調べ方では限界に達している。

そのため, 約数の和を求めるためのアルゴリズムを変え, 計算量を減らすことが必要となってきた。

改良 1: たとえば, 100 の約数は, 1,2,4,5,10,20,25,50,100 である。50 から 100 までは「ない」。つまり, n/2+1 から n までには存在しないので, それを考慮してプログラムを修正すると, 計算量は, 半分になる。

改良 2: 上記の 100 の約数をペアにすると, (1x100), (2x50), (4x25), (5x20), (10x10)となる。そのため, n に対して, $\sqrt{n}$ までの数 k について, k が約数なら, k, n/k を約数として拾いだせば, 計算量は, $\sqrt{n}$ まで激減する。

この改良をすることによって, n=138 に関しては, 次のように, 178 項で 1 に収束することが分かった。

138,150,222,234,312,528,960,2088,3762,5598,6570,10746,13254,13830,19434,20886,21606,25098,26742,26754,40446,63234,77406,110754,171486,253458,295740,647748,1077612,1467588,1956812,2109796,1889486,953914,668966,353578,176792,254128,308832,502104,753216,1240176,2422288,2697920,3727264,3655076,2760844,2100740,2310856,2455544,3212776,3751064,3282196,2723020,3035684,2299240,2988440,5297320,8325080,11222920,15359480,19199440,28

875608,25266172,19406148,26552604,40541052,54202884,72270540,147793668,228408732,348957876,508132204,404465636,303708376,290504024,312058216,294959384,290622016,286081174,151737434,75868720,108199856,101437396,76247552,76099654,42387146,21679318,12752594,7278382,3660794,1855066,927536,932464,1013592,1546008,2425752,5084088,8436192,13709064,20563656,33082104,57142536,99483384,245978376,487384824,745600776,1118401224,1677601896,2538372504,4119772776,8030724504,14097017496,21148436904,40381357656,60572036544,100039354704,179931895322,94685963278,51399021218,28358080762,18046051430,17396081338,8698040672,8426226964,6319670230,5422685354,3217383766,1739126474,996366646,636221402,318217798,195756362,101900794,54202694,49799866,24930374,17971642,11130830,8904682,4913018,3126502,1574810,1473382,736694,541162,312470,249994,127286,69898,34952,34708,26038,13994,7000,11720,14740,19532,16588,18692,14026,7016,6154,3674,2374,1190,1402,704,820,944,916,694,350,394,200,265,59,1

他の数に関しても，n=168 のとき，175 項，n=180 のとき，52 項で 1 に収束することが分かった。

3.7 n=138 の場合のように，100 項以上になるものを探す（探究 7）

アルゴリズムの改良により，200 項くらいまでは迅速に計算できるようになったので，138 や 168 のように，数列が 100 項以上になっても収束しないものについて調べたくなった。

この作業を円滑にするためには，数列が得られてから数列を表示する方が適切なので，数列の値を配列に保存し，必要に応じて表示するように，プログラムを改良した。

200≦n≦300 の範囲について調べると，次の結果が得られた。

・n=220 の場合には，220 と 284 を交互に繰り返す循環数列になる（220, 284 が友愛数）。

・n=222 の場合に，176 項。（n=234 も 222 の系列に含まれ，175 項）

・n=276 の場合に，非常に計算に時間がかかり，計算結果が表示されない。そこで，90 項までを表示してみると，次のようになっている。

276,396,696,1104,1872,3770,3790,3050,2716,2772,5964,10164,19628,19684,22876,26404,30044,33796,38780,54628,54684,111300,263676,465668,465724,465780,1026060,2325540,5335260,11738916,23117724,45956820,121129260,266485716,558454764,1092873236,1470806764,1471882804,1642613196,2737688884,2740114636,2791337780,4946860492,4946860548,9344070652,9344070708,15573451404,27078171764,27284104204,27410152084,27410152140,76787720100,220578719452,254903331620,361672366300,603062136740,921203207260,1381419996068,1395444575644,1395478688996,1395546402460,2069258468900,3065057872156,3277068463844,3429776547484,3597527970596,4028517592540,5641400009252,5641400009308,5641400009364,9709348326636,16331909651988,31948891146732,54770416120644,100509779504316,208751080955844,388416032284476,749365894850244,1414070378301756,2556878765995204,2556878765995260,6726041614128900,15626498692840700,23762659088671300,35168735451235260,78257359358590020,186897487211247036,340813223900632644,592585414385033916,13

26583294186844484

第 90 項は 19 桁もある。n=138 のときの数列の最大値が 179931895322 つまり 12 桁であったのと比較すると，7 桁多いということは，さきほどのアルゴリズムでも，一つあたりの計算量が 5000 倍近くに増えていることを意味する。

3.8 n=276 の場合を解決するためのアルゴリズムの改良（探究 8）

n=276 の場合には，前回よりも，より効率的なアルゴリズムに変えなければいけないようだ。そこで，素因数分解を使って約数の和を求める方法を使うことにしてみた。素因数が少ない場合にはあまり有効ではないが，素因数が多い場合には，かなり計算量を少なくすることができるはずだ。実際，この方法でかなり改善されたのだが，項数が増えるとともに桁数が増えていくので，30 分の計算で到達できたのは，a(95)= 13780400058385352252 の 20 桁の数までであった。

3.9 n=276 の場合を解決するために，Python 用のライブラリ SymPy を利用する（探究 9）

Python という言語には，さまざまなライブラリがある。そこで，このような計算，つまり数式処理に使えるライブラリとして，SymPy を使ってみることにした。具体的には，上述のプログラムの中の素因数分解のところだけを，SymPy の factorint という関数を使うことにした。効果はすばらしく，さきほどは 95 項まで到達するのに 30 分かかったのに，100 項まで到達するのに，0.44 秒しかかからなかった。ちなみに，a(100)= 21374326697892540932 の 20 桁の数であった。

さらに計算させると不思議な現象に遭遇した。

94 項で 20 桁，116 桁で 21 桁，119 項で 22 桁というように，数列の値は大きくなり，桁も増えていき，178 項では 27 桁まで到達するのだが，その後減少し，250 項では 15 桁の極小値になる。しかし，その後また若干増減を繰り返すものの，全体的には桁数は増えていき，400 項までの最大値は a(372) = 57484123452034462154661462024737 6 の 33 桁であった。

その結果，300 項まで調べるのに要する時間は 15.78 秒なのに，400 項まで調べるのに要する時間は 10 分 40.83 秒と急に大きくなった。

3.10 n=276 の場合を解決するために，mathematica を使う（探究 10）

さらに先の項まで調べるために，数式処理ソフトの代表としての mathematica をラズベリィパイで試してみることにした。mathematica を使うと，数列を漸化式で表現することができる。たとえば，今回の数列を n=276 の場合に関して，400 項まで調べ，そこで要した時間を測定するためには，次のようなプログラムを利用した。

```
a[1] = 276;
a[n_] := (a[n] = Sum[Divisors[a[n - 1]][[i]], {i, 1, Length[Divisors[a[n - 1]]]}] - a[n - 1]);
Table[{a[i]}, {i, 1, 600}]
```

一応このプログラムでも動作するのだが，mathematica に詳しい大橋真也先生（千葉県立千葉中学校・千葉高等学校）に，「今，こんな問題に関して mathematica で，こんなプログラムを試しているのだが」と投げかけてみたら，下記のように改善するだけで，約 5 倍の速さになることを教えていただいた。

```
a[1]=276
a[n_]:=a[n]=(Apply[Plus, Divisors[a[n-1]]-a[n-1]];
Flatten[Table[{a[i]}, {i,1,400}]
```

SymPy のライブラリを使った計算でも，400 項まで調べるのに 10 分以上かかったのに，Mathematica での上記のプログラムは 12 秒しかかからなかった。500 項までは 179 秒かかり，たとえば，第 500 項の値は，次のように 53 桁になっている。

a(500)= 219328670563481940017362738533445920387626392265316 96

しかし，この後の項も同じくらいの計算時間で処理できるかというと，そうではなく，たとえば，521 項をもとに 522 項を求めるのに，85 秒かかるなど，多くの時間を要するようになる。

実はそこには，ラズベリィパイというコンピュータの性能(CPU 速度，メモリ量)の問題も関わっている。ちなみに，大橋先生のパソコンでは，600 項まで求めるのに，92 秒で処理できたとのことだった。また，大橋先生の指摘では，さきの二つのプログラムでは消費するメモリ量に違いがあって，速度が大きく異なるとのことで，計算量が大きくなることに伴って，改善のために配慮しなければならない要因は，さまざまに広がっていくことを実感した。

4. コンピュータを使った数学的探究における「サイクル」に関する考察

4.1 ここまでの探究結果から得られた数学的結果

上記の数学的探究の中で注目に値することは何であろうか。出発点である，完全数そのものに関しては，6,28,496,8128 を確認しているが，次に登場するはずの 33550336 には到達していない。友愛数や社交数に関しても同様である。ある意味で，数学的結果に関しては，何も新しいことは見つかっていないと言ってもいいかもしれない。

4.2 ここまでの探究結果から得られた数学的問題や予想

しかし，この探究は，少なくとも私にとっての未解決問題を，暫定的な結果として残している。

未解決の問題

自分以外の約数の和を次の数とするきまりで生成される数列を考える。

この数列の初項を 276 とするとき，この数列は，収束するのか，循環するのか，発散するのかがわからない。(私がこれまでに調べた結果では，680 項が次の値になるところまではわかっている 52486982159651936793206723465369736165223685334399524409082807119383722 52)

このようなスタイルの問題(予想)として有名なものは，コラッツの問題，つまり

整数 n に対して，n が偶数ならば n/2 を，n が奇数ならば，3n+1 を対応させる操作

を考えたとき，有限回の操作で 1 になるか

がある。「1→4→2→1」という周期以外の周期があるかという問題ともいえるし，この手続きで数列をつくったときに，その数列は発散することはないのか，という問題ともいえる。

つまり，そのような側面で考えれば，数学的に有名な(未解決)問題と同じような種類の問題を見つけたということもできる。

4.3 探究の中での「事実との遭遇」

私にとって，今回の数学的探究は興味深いものだったが，特にその中で印象的だったものは，事実との遭遇である。

まず，「138」の場合に出会った。これはその後解決されたのだが，次に 276 が登場してきた。Lakatos の「数学的発見の論理」での記述に即せば，文字通り「モンスター」の登場である。これらのモンスターの正体は何なのか。それを究明していくプロセスは，謎解きのような面白さを有している。

そして，ある意味で，これらの事実との遭遇を可能にしたのは，コンピュータによって，短時間で正確な計算実験が可能であるこそともいえる。

4.4 問題から問題を生む

しかし，いくら計算が速いからといって，当初のプログラムをただ長時間走らせるだけでは，数学的探究としての深まりはほとんど生まれない。このような探究において，それを進めていく上では，最初の問題そのものにずっと固執していてはあまり進展しない。むしろ，そこまでに得られた結果から，関連することや，発展させるべきことなど，「そこから考えるべき問題を探す」ことに，むしろ力点があると言った方がいいのではないだろうか。実際，今回の探究では，完全数かどうかの判定だけにこだわるよりも，数列を考え，その数列の挙動を観察することから，いくつかの興味深い結果や，問題を導いていった。そのように，能動的に問題を生み出すことが，探究の中では大切なのである。

今回の事例でいえば，そのような定式化をしようと思った背景には，単に数を構成する手続きとしてでなく，それを一つの対応あるいは関数として捉えたり，その系列を数列として捉えようとする考え方があったといえる。そこにはもちろん数学的概念もあるだろうが，プログラミング言語を使った場合には，一つの手続きのまとまりはプログラミングの意味での関数として表現したり，配列に記録して処理することが適しているという情報科学的な概念との関わりもあるだろう。数学とコンピュータとの関わりがあるからこそ生まれる STEM 教育的な問題の生成の仕方の一例ということになるのかもしれない。

4.5 改善すべき方法の所在から生まれる問題

さらに，プログラムとして表現されたことによって，そして，そこに生まれる実験(数学的計算)でのボトルネックの明確化やその改善の必要性として，様々な問題が生まれている。今回の探究でいえば，たとえば，約数の和を求めるアルゴリズムを考える上で，約数の集合の特徴を反映して，プログラミングを改善したり，素因数分解と約数との関わりを反映するプロセスは，中高生でも考察可能な数学的探究といえると思う。一方，同じ素因数分解でも，素朴な方法で実現したものと，SymPy のようなライブラリを使ったものとで大きな速さの違いがあることを実感すると，そういう仕組みを実現するために，どこかに数学的な理論が反映されていることを実感するのではないだろうか。それは mathematica を使ったらさらに高速化されることにも関連する。単にコンピュータを使えばいいというだけでなく，高速に処理するために，さまざまな数学的研究

が存在していることを実感できるのではないだろうか。生徒によっては，将来，ユーザーとして，あるいは開発者として接することになるだろうが，そういう現場を肌で感じられる経験として生かすことができれば，そこに一つの教育的価値が生まれると言っていいのではないだろうか。

また，最後の mathematica での処理のように，ソフトによっては，論理的なアルゴリズムだけでなく，メモリの扱い等に関する挙動や，コンピュータの構成や挙動などを分析する必要も生まれてくる。それら全体を配慮しなければならないケースに関しては，まさしく，STEM 教育の中で位置づけていくべき事例ということになるのではないかと考える。

4.6 数学の中にも「未解決」はありそうだし，開拓できる余地もありそうだという実感

276 から始まる，今回の数列は無限に続くのだろうか。無限に続くという予想を立てたくなる一方，138 は 1 に収束したように，どこかで収束したり，循環してもおかしくないとも思える。実際，和が素数になった瞬間に，次は 1 になるのだから。どちらにしても，きちんと証明することは難しそうに思える。普段接する数学の中に，そういうものの存在を実感することは少ないが，このような例に触れていると，数学の中にも「未解決」というものはまだまだありそうな気がする。そしてまた，完全数に関わっていろいろな数学的活動が広がったように，まだまだ開拓の余地も残されているような実感もしてくる。探究することの一つの大きな副産物は，そういう実感をということなのかもしれない。

4.7 このような探究の教材開発を進めていく上で必要なこと

今回，大学院の授業の中でも部分的に扱ったのだが，大学院生がさまざまな点で，「問題を探究する」ことには慣れていないことを実感した。あるいは，「模範解答が用意されているような問題に取り組む」ことは慣れていても，漠然とした問題やその結果から，「次に考えるに値する問題を見つける」ことに慣れていないというべきなのかもしれない。与えられた問題に対して，一定の解決を得るところまでは到達してくれているのだが，「その中でどういうところが面白いと思ったの?」とか「で，次に何をしたいと思っているの?」というような私の問いに対して，窮することが多かった。ある意味で，これまでの普通の算数・数学の授業の中では，あまり求められてこなかった力なのかもしれない。また，探究が長く続く大きな原動力は，その問題の中に面白さを見いだし，熱中することでもある。彼らにとって，熱中するに値するものが感じられなかった程度の素材なのかもしれないが，「面白いと思う問題に熱中する」ことは，このような探究にとっては不可欠なことでもある。

これらのことに関連して，大切ではないかと思うことは，言語活動である。特に，探究そのものを記述する活動である。どういう問題に対して，どういう取り組みを行ない，その結果としてどういうことが得られ，それに対してどのような解釈あるいは価値づけをして，次に何をすべきと感じたかなどである。もちろん，だらだらと記述してもいけないのだが，数学の大学院生や学部生の場合，そのような言語活動が苦手な人が少なくない。やはり，これも通常の授業の中では，たとえば，与えられた問題に対する模範的な解答の書き方までは指導するが，それ以外のことに関する言語活動はあまりしていないことに起因するのかもしれない。

備考 1: 本稿は, 2017/8/30 に, 京都大学・数理解析研究所で開催された研究集会「数学ソフトウェアとその効果的教育利用に関する研究」(代表 中村泰之)において発表した「数学的探究のサイクルを回す原動力としての数学ソフトウェアの役割 - ケーススタディを中心に -」の内容の一部を拡充してまとめたものである。また, 本稿は, 科学研究費補助金 基盤 C(17K00967)「テクノロジーを利用した数理的現象の探究の教材開発と授業実践」(研究代表者 飯島康之)の支援を受けている。

備考 2: 本稿の中では, あまり詳しく記述していないが, 数学的方法をプログラムとして実装するプロセスの中で, プログラミング言語の諸概念の理解やそれを踏まえた実装が必要になってくる場面も少なくない。たとえば, 約数の個数を求める場合に, 与えられた n に対して, 1 から n まで確認するだけなら簡単だが, √n までで対処しようとすると, n が小さい場合には例外的な対処をする必要が生まれる。さらに, 素因数分解を使おうと思うと, 整数の素因数分解の表示に適したクラスを設計する方がいい。素因数の個数は不定なので, 配列に対する要素の追加・削除などを使うことも必要になる。また, 他のプログラミングが短く記述できるのと比べると, 長くなることと複数の関数で記述することが必要になるため, きちんと機能しているかどうかを確かめる作業やデバッグの作業も必要になってくる。高校生や大学生が, プログラミングとの関わりでこれらの方法を実装するような場合には, プログラミングに関する知識・スキルの学習との関わりへの考慮も必要になってくる。

また, 本稿の中では記述しなかったが, 素因数分解を使った結果は, ある数よりも大きくなった場合には, 当初, 正しい結果になっていなかった。どの場合に正しくないかを明確化する中で, 78393390086384856 の約数の和が, 正しくないこと, そしてその原因として, int(n/i)の計算の部分があり, 「18014398509481986 以上の偶数に関して, /2 の計算結果にバグがある」のではないかと感じ, 上記の研究集会のときに言及してみた。すると, さすがに数値計算に慣れている方々。この数は 2^64 によく似た数であること(実際には, 2^54), Python2 ではこの結果は生じないこと, Python2 と 3 では文法的な違いがいくつかあり悩みの種で, Python2 のまま使っている方が多いこと, 整数除法なら, Python3 では, n//i が妥当であることなど, さまざまなことを教えていただいた。Python2 と 3 で言語仕様に違いがあるとはいえ, int(n/i)の計算結果にズレが生じることなど想像していなかった私にとっては, 驚き以外の何物でもなかったが, 数値計算の「現場」というものを肌で実感した瞬間であったことには間違いない。発表の場でご助言をいただけた方々に, この場を借りてお礼申しあげる次第である。