# 大学一般プログラミング教育の実践事例
## －プログラミング的思考力の育成を中心に－

倪　　永茂

### はじめに

2020年度に日本の小学校ではプログラミング教育が必修化された。中学校では2011年度に必修化がすでにスタートしたが､2021年度に「技術･家庭」でプログラミングに関する内容が拡充されることになった。2022年度に高校ではプログラミングを含む「情報Ⅰ」の必修化が開始されようとしている。プログラミング教育が学校教育においてますます重要視する傾向である。それに合わせて、2025年度から大学入学共通テストでも従来の5教科7科目に「情報」を加えた6教科8科目とし、プログラミングに関する問題が出題される予定となっている[1]。

大学入学共通テストに「情報」を追加した方針について、国立大学協会の永田恭介会長は「国立大学においてもすでに多くの大学で、『数理･データサイエンス・AI教育』が文理を問わずすべての学生が身に付けるべき教養科目として履修されている。このような中において『情報』に関する知識については、国立大学の教育を受けるうえで必要な基礎的な能力の1つとして位置づけられていくと考えている」との談話を発表した[2]。

筆者が勤務している国立大学でも確かに､「データサイエンス入門」という名の科目が全学必修科目となっている。そのシラバスは従来の情報処理基礎、つまり、タッチタイピングの修得、ネット・学術データベースの検索、Officeソフトの使い方等に、新に数理・データサイエンス関連の内容を追加したものである。

しかし、筆者の大学においてプログラミング教育という類の授業は開講されてはいるが、全学必修にほど遠い状況にあり、履修した学生数が入学者数の半分を遥かに下回っている。現状では大学入学後に、コンピュータ・プログラミングについてさらに学習する環境がまだ十分に整備されていないといわざるをえない。

本稿は大学における一般プログラミング教育の全学必須の是非を問うものではなく、筆者の実践している一般プログラミング教育の事例を紹介し、それがプログラミング的思考の育成や情報科学の理解につながることを示す目的とする。

### Ⅰ 数学的思考からプログラミング的思考へ

数学的思考とは何か。明確な定義はないが、少なくとも抽象化（モデル化）すること、厳密な証明が必要不可欠であることが数学的思考力に含まれると考えられている。すなわち、与えられた課題を数学的モデル化により抽象化し、モデルから導き出された数式は実験や経験等といったものではなく、厳密な数学的証明によって導き出すことが数学的思考だと思われる。

一方、プログラミング的思考とは何か。日本を代表する専門家たちの意見を集約して、文部科学省はそれをつぎのように述べている[3]。

「自分が意図する一連の活動を実現するために、どのような動きの組み合わせが必要であり、一つ一つの動きに対応した記号を、どのように組み合わせたらいいのか、記号の組み合わせをどのように改善していけば、より意図した活動に近づくのか、といったことを論理的に考えていく力」

つまり、符号化された動きの組み合わせをどう行えばいいか、さらに一旦決めた組み合わせを自分の意図に近づかせるためにどう組みなおして、改善していけばいいか、それを論理的に考えることをプログラミング的思考ということであろう。

従来の数学的思考を学校教育において強化する考え方は無論あるが、それでもさらにプログラム的思考を必要とする理由は何であろう。

これに答えるために、以下では両者の違いについて述べていく。

### 1「時間の有限性」という概念の有無

ここでいう「時間の有限性」とは、テスト等で制限された時間内に数学問題を解くということではなく、一般的に、数学問題を解くためにかかる時間を無視してよい、つまり、無限の時間を使ってよい、という考え方である。

たとえば、整数の因数分解は数学では簡単に解ける問題としてよく知られている。素因数分解の一意性という算術の基本定理として、高校までの数学教育で習ったはずである。言い換えると、数学ではどんな整数でもその因数分解は何ら問題なく解決できるとされている。

しかし、プログラミング教育では、「時間の有限性」をつねに意識しないといけない。つまり、与えられた問題を限られた時間内に解けないといけないという制約である。たとえば、素因数分解は限られた時間内ではできないと広く信じられている。この因数分解の困難さはRSA暗号のような公開鍵暗号における信頼性の基礎になっている。

そう考えると、多くの問題は数学では解決したとしても、プログラミングの世界では未解決問題として知られている。

プログラミング的思考力に関する文部科学省の定義において、改善することの前提となる効率性の追求がここでいう時間の有限性と関連する。

プログラム教育を含め、情報を科学的体系に整理しまとめたのは情報科学という分野であるが、情報科学の基礎をなす「計算量」という概念は計算時間を論じ、プログラムの時間的効率性を学問的に究明するものである。

### 2 処理手順の重視

順次処理、分岐処理、そして繰り返し処理がプログラム教育において避けて通れない処理手順であるが、繰り返し処理は数学的思考ではそれほど強調されない。たとえば、等差数列や等比数列の各項の合計問題では、与えられた計算式に代入するだけで合計問題が解決されることになっている。また、個々人にとって、同じ数式や同じ論理的思考で解く問題を大量に練習することが、数学的思考の定着に必要とされている。

対して、プログラミング的思考では、合計の計算式を考えないで、各項を繰り返し処理によってひたすら足していくだけになる。様々な数学的性質を議論することに適していないかもしれないが、等差数列であろうが、等比数列であろうが、その他の数列であろうが、どんな数列でもその各項の合計を計算することができる。

### 3 実用性の重視

命題の証明に数学では多くの手法が考案されてきた。背理法や帰納法等がその例である。厳密な証明とは何か、論理的な思考とは何か、命題の証明を通して確認していくのである。

対して、一般プログラミング教育では、命題の証明をしない代わりに、あらゆるケースから最適解を見つけたり、正解を出したりすることができる。つまり、実用性がより重視されている。

一般的に、プログラミング教育において、大学で習う高等数学（高等代数、微分、積分、解析幾何、抽象代数、関数論等）に関する知識がなくても、実用性のある応用問題を解くことに問題になることはないといわれている。

### 4 計算効率の追求

実用性を重視すれば、計算効率の追求がおのずと求められる。現に、実行時間のかかるプログラムが多く､数分～数時間以内に実行が終了しない。それを改善するためには、効率のよいデータ構造やアルゴリズムを採用したりする。

## Ⅱ 一般プログラミング教育における実践授業

では、プログラミング的思考力を育成するにはどういうプログラミング教育が有効か。筆者が行った実践授業について、その教育内容の一部を紹介し、考察してみることにしよう。

### 1 繰り返し処理に慣れさせる

あらゆるプログラムは順次処理、分岐処理、繰り返し処理の３つだけで作成できることが知られている。しかし、順次処理や分岐処理と違い、繰り返し処理が従来の数学教育ではその手順を厳密に示すことが少なく、しかも繰り返し処理自体がわかりにくく、多くの学生は戸惑いを感じる。

同じ変数を繰り返し処理で使うことも、繰り返しの終了条件を明示することも、またいくつかの繰り返し処理のパターンも、合わせて学生に理解してもらい、覚えてもらうことが重要である。

たとえば、2つの整数の最大公約数を求める方法は複数あるが、いずれも繰り返し処理が必要である。しかし、習ってきたはずの学生は繰り返し処理に意識したことがなく、その処理手順を厳密に説明できることもできない。

繰り返し処理によって、プログラムの処理が遅くなってしまうことがよくある。言い換えると、繰り返し処理の存在によって、計算量が情報科学の学問分野として研究対象になった。

また、多くの組み合わせ問題において、それほど大きくない問題サイズでも、プログラムの処理時間が何万年、何百億万年もかかることを示すことにより、アルゴリズムの重要性を気づかせる最初のステップとする。

## 2 約数の和を計算させる

整数 $a$ の約数の和とは、$a$ のすべての約数を足し合わせたものである。たとえば、6の約数の和は $1+2+3+6=12$ である。

約数の和について、数学的思考によって、以下の数式が得られることがよく知られている。

整数 $a$ が $a=p_1^{n1}\,p_2^{n2}\cdots p_k^{nk}$ と素因数分解されたとき，$a$ の約数の和は、

$(1+p_1+p_1{}^\wedge 2+\cdots+p_1^{n1})(1+p_2+p_2^{2}+\cdots+p_2^{n2})$
$\cdots(1+p_k+p_k^{2}+\cdots+p_k^{nk})$

となる。

しかし、この数式がそのままでは約数の和の計算に使えない。素因数分解が前提になっているからである。素因数分解を数学ではなく、プログラムで行うには、また繰り返し処理が必要不可欠である。

一方、プログラミング的思考では、約数の定義から、素因数分解をまったく意識することなく、$a$ の約数を1から $a$ までひとつずつ確かめていって､約数を足し合わせるだけで約数の和が求まる。数式ではなく、処理手順によって約数の和が求まることがいかにもプログラミング的思考といえよう。

＜約数の和＞（オーソドックスな計算法）

```
int s = 0;  // 変数sは約数の和の値
for (int i = 1; i <= a; ++i) {
  if (a % i == 0)
    s = s + i;
}
```

約数の和が計算できるようになれば､完全数や、友愛数、婚約数、社交数等、多くの整数問題を考えることにつなげられる。無限個の存在可能性、奇数ペア・偶数ペア・奇偶数ペアの有無等、多くの不思議な性質を学生に話し、好奇心を掻き立てることも可能になる。

## 3 計算効率を改善する

オーソドックスな計算法で約数の和を求め、それを元に友愛数ペアを小さい順から25ペアを課題として学生に提示し、解答プログラムの作成と実行時間（計算時間）の計測をやってもらった。

それぞれの学生の所有するパソコンの機種や実行時の環境（多くのプログラムを同時実行しているか否か等）の違いにもよるが、概ねプログラムの計算時間が数分程度になっていた。

プログラムというものはあっという間に実行が終わると思い込んでいる学生が多く、数分間の計算時間にびっくりしていた。

そこで、計算時間を遅くした理由は繰り返し処理にあり、繰り返し処理の高速化、あるいは、繰り返し回数の削減が計算時間の短縮につながることを解説し、グループワークで計算効率の改善を考えてもらうことにした。

学生がすぐに思いつくのは次に紹介する半減法である。つまり、整数 $a$（$a>1$）の約数に1と $a$ が含まれることは自明であり、また $a/2$ と $a$ との間のどの整数も約数になりえない、という性質を活用した計算法である。

＜約数の和＞（半減法）

```
int s = a+1;   // 変数sは約数の和の値
for (int i = 2; i <= a/2; ++i) {
  if (a % i == 0)
    s = s + i;
}
```

半減法は筆者がつけた名称ではあるが、$a$ までではなく、$a/2$ まで調べることでそう命名した。

半減法による友愛数 25 ペアをもう一度求めてもらったところ、計算時間がたしかに半分以下に短縮されたことが確認できた。

## 4 計算時間と計算量

オーソドックスな計算法も、半減法も、学生それぞれが計測した計算時間はバラバラであった。そのことは、同じプログラムであってもそれの実行時間がパソコン（ハードウェア）に依存していることを意味する。勿論、10 年後に製造されるパソコンを使うと実行時間が半分以下になったりする。

では、プログラムそのものの計算時間を論じることはできないか。

グループワークで討論してもらい、出てきたアイデアのひとつは 2 つのプログラムの計算時間を比較するというものであった。

実際に比較してもらったところ、友愛数ペアごとの計算時間は図 1 のようになる。曲線が滑らかに変化していないのは、友愛数ペアの分布は一様でないことを意味する。なお、それぞれの計算時間を示す曲線がみな同じ傾向であることに学生が納得した。

しかし、比較対象となるプログラムがそもそも存在しないケース、あるいは、プログラムのコードが公開されないケース等において、単独のプログラムに対して、計算時間をどう科学的に考えるべきなのか。

それに応えるために、計算量という概念を授業で紹介することにした。計算時間が入力データサイズの変化に応じて変化するが、それらの変化が計算量という概念で捉えられている。計算量にはさらに時間計算量、空間計算量の 2 種類に分けることができるが、空間計算量のほうは一般プログラミング教育で議論する必要はほとんどないと考えられる。

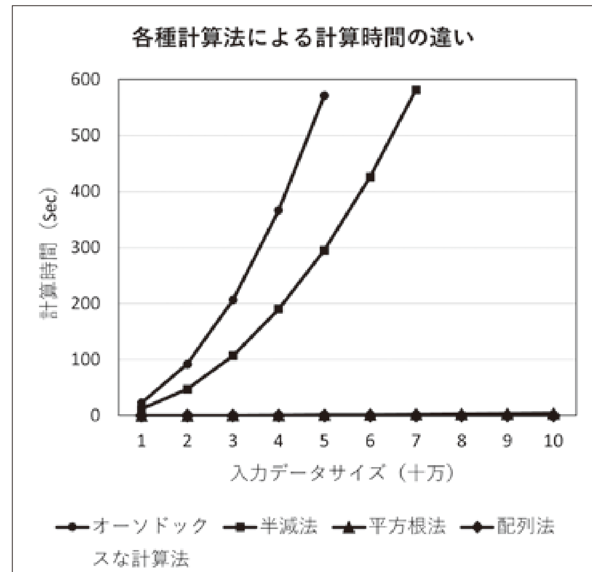


図 1　友愛数ペア数ごとの計算時間

## 5 アルゴリズムと計算量

プログラミング言語と関係なく、すべてのプログラムのベースとなるものは、一般的にアルゴリズムという。

『原論（Elements)』は紀元前 3 世紀に書かれた書物といわれるが、2 つの整数に関する最大公約数の求め方が書物のなかで述べられていた。その求め方を今日では一般的に、ユークリッドの互除法と呼ばれ、最古のアルゴリズムとして知られている。2000 年以上経ってもそれ以上に高速な計算方法が見つかっていないことは、アルゴリズムの生命力の強さを示す好例であり、人類が共有する知的財産といわれるゆえんでもある。

さて、計算量の違いによってアルゴリズムの一部をつぎのとおり、グループ分けすることができる。ただし、$n$ は入力データサイズを表す。

1. 入力データサイズと無関係なアルゴリズム
   ソート済の入力データから、最小値や最大値を求めるアルゴリズム。ひとつの数式で問題解決できるアルゴリズムも本グループに入る。
2. 計算量が $\log n$ であるアルゴリズム
   バイナリサーチ（二分探索。ソート済の入力データから、特定のデータを求めるアルゴリズム)。
3. 計算量が $n$ であるアルゴリズム
   線形探索（入力データから、特定のデータを求めるアルゴリズム)。

4. 計算量が $n \log n$ であるアルゴリズム
   クイックソート。
5. 計算量が $n^2$ であるアルゴリズム
   挿入ソート、バブルソート、選択ソート等。
6. 上記以外のアルゴリズム

グループ6に分類されるアルゴリズムに、グループ1〜5の計算量に近いものもあるが、計算量が指数以上であるものもある。

計算量が指数以上であるアルゴリズムは計算時間がとてもかかることから、実用性がないといわれている。たとえば、巡回セールスマン問題がその一例である。しかし、最適解を求めなければ、実用範囲内では工夫次第、グループ1〜5に改善されることもよくある。

プログラミング的思考で求められる計算効率の改善は同じ計算量グループ内の改良ではなく、より上位グループへの移動、つまり飛躍的な改善を目指すことである。そのことを再三学生に強調すべきであろう。たとえば、約数の和に関するオーソドックスな計算法と半減法は、計算時間に関して半分以下の改善になっているが、それはグループ5内での改良であり、上位グループへの移動にならないことを強調する。

## 6 計算効率をさらに大幅に改善する

約数の和について、さらにその計算量を大幅に改善するアルゴリズムはあるか。

実は数学的思考力によって、2以上の整数 $a$ は2つの約数の積で表せること、さらに片方の約数が $\sqrt{a}$ 以下であることを証明することができる。

その性質を利用すれば、約数の和の計算に関する3つ目の方法が得られる。

＜約数の和＞（平方根法）

```
int s = a+1; // 変数sは約数の和の値
int t = sqrt(a);
for (int i = 2; i <= t; i++) {
  if (a % i == 0)
    s += (i + a/i);  // 約数 i, a/i
}
if (t*t == a) s -= t;
```

平方根法という名称も筆者がつけたものである。2から平方根 $\sqrt{a}$ までしか調べないことを根拠にしてある。

証明は省略するが、平方根法による約数の和の計算量は $n^{1.5}$ である。グループ5から大きく改善したアルゴリズムになった。

平方根法をさらに大幅に計算量を改善する方法もあり、それを配列法と命名した。アルゴリズムのヒントは素数の一覧表を求める方法、いわゆる、エラトステネスの篩（Sieve of Eratosthenes）にある。

＜約数の和＞（配列法）

```
#define MAX 1000000
 int i, j;
 int s[MAX];  // 配列sは約数の和
 for (i = 2; i < MAX; ++i)
   s[i] = i+1;
 for (i = 2; i < MAX/2; ++i)
   for (j = 2*i; j < MAX; j += i)
    s[j] += i;
```

証明は省略するが、配列法による約数の和の計算量は $n \log n$ である。平方根法の計算量をさらに改善し、グループ4に入ることができた。

ただし、配列法は問題がないわけではない。プログラミング言語によって取れる配列の大きさに上限があり、それを超えた整数については約数の和を求めることができない。対して、平方根法は計算時間がかかるが、整数に対する上限はない。

計算効率の追求はプログラミン的思考力の育成に欠かせないプロセスである。

また、計算効率の追求には、トレードオフという判断が求められることもよくある。課題に応じて、使うアルゴリズムを変える柔軟性が必要であろう。

約数の和に関する4つの計算法の計算時間はそれぞれ表1に示し、グラフ化したものをまとめて図2に示す。ただし、平方根法と配列法は計算時間が速く、図2だけではその計算量を観察することは不可能となってしまうので、平方根法と配列法の計算時間をさらに図3に表示させる。

計算の効率がアルゴリズムの違いによっていか

表１　各種計算法の計算時間（Sec）

| 入力データサイズ（万） | オーソドックスな計算法 | 半減法 | 平方根法 | 配列法 |
|---|---|---|---|---|
| 10 | 22.82 | 11.83 | 0.13 | 0.004 |
| 20 | 92.15 | 47.60 | 0.37 | 0.008 |
| 30 | 206.70 | 107.07 | 0.67 | 0.012 |
| 40 | 366.82 | 190.64 | 1.03 | 0.015 |
| 50 | 571.56 | 295.50 | 1.43 | 0.020 |
| 60 | | 426.03 | 1.91 | 0.027 |
| 70 | | 581.50 | 2.40 | 0.031 |
| 80 | | | 2.93 | 0.038 |
| 90 | | | 3.49 | 0.044 |
| 100 | | | 4.11 | 0.051 |

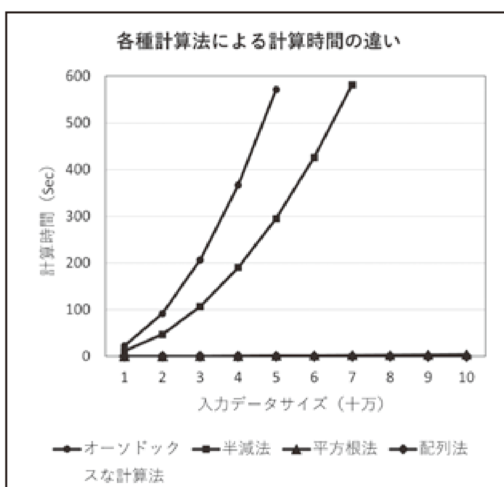


図２　各種計算法による計算時間の違い

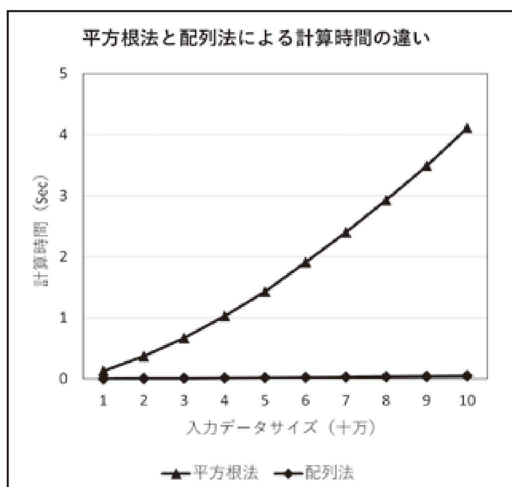


図３　平方根法と配列法による計算時間の違い

に変わるか、図 2 から学生に十分にわかってもらえた。計算の効率を上げることがプログラミング的思考力を高めることにつながり、大勢のプログラマーや専門家がその改善に日々努力している。

## 終わりに

本文は従来の学校教育における数学的思考に、さらにプログラミング的思考を加えるために、両者の違いを、時間的有限性、処理手順、実用性、計算効率に対する考え方の違いという視点から論じたうえで、筆者の実践してきた大学における一般プログラミング教育における授業を事例として、計算効率に深く関わる繰り返し処理から、計算時間・計算量への理解、異なるアルゴリズムによる計算時間の違いまで、そういう一連のプロセスがプログラミング的思考の育成につながることを示した。

---

[1] 国立大学協会が 2022 年 1 月 28 日に発表した「2024 年度以降の国立大学の入学者選抜制度―国立大学協会の基本方針―」。

[2] 読売新聞 2022 月 1 月 31 日。

[3] 文部科学省「小学校段階におけるプログラミング教育の在り方について（議論の取りまとめ）」小学校段階における論理的思考力や創造性、問題解決能力等の育成とプログラミング教育に関する有識者会議（2016 年 6 月 16 日）。

## 参考文献

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; and Stein, Clifford (2009) *Introduction to Algorithms* (3rd ed.), MIT Press and McGraw-Hill.

倪永茂（2019）「素数、約数、合成数に関するプログラミング教育」宇都宮大学国際学部研究論集 第 48 号 95-102。

# A Practical Case Study of General Programming Education:

## Focusing on the Development of Programming Thinking Skills

NI Yongmao

**Abstract**

In order to add programming thinking to mathematical thinking in conventional general education, this article discusses the differences between mathematical thinking and programming thinking from the viewpoints of finiteness in time, processing procedures, practicality, and differences in thinking about computational efficiency, and uses the author's classes in general programming education at a university as a case study. As an example, the author showed that a series of processes, from repetitive processing, which is deeply related to computational efficiency, to an understanding of computation time and computation amount by different algorithms, can lead to the cultivation of programming thinking.

（2022 年 5 月 30 日受理）