第2章 アルゴリズム解析の基礎

トレーシー　「そこにいるのに気がつかなかったよ。」

ゾーイ　「それがまさに狙ったことさ。ステルスなんだ。聞いたことはあるかもしれないけど。」

トレーシー　「それは基礎編じゃ説明されていなかったわ。」

Fireflyの第14話 "The Message"（遺言）より

数学的な手法、アルゴリズムの設計原則、そして、本書の大部分を占める古典的なアルゴリズムに移る前に、扱っておくべき基本的な原則や手法があります。次章以降を読み始めるときには、「負のサイクルがない重み付き有向グラフ」や「$\Theta(n \log n)$ の実行時間」といった用語の意味が明確に理解できている必要があります。加えて、基本的な構造をPythonで実装する方法についても理解しておく必要があります。

幸い、この基本的な原則や手法を理解するのはそこまで難しくありません。本章の主な2つのトピックは、実行時間に焦点を当てる**漸近記法**と、Pythonを使った**木構造とグラフの表現方法**です。みなさんがプログラムの実行時間を測るとき、陥りやすい「罠」があるので、本章ではその罠に陥らないようにする実践的なアドバイスをします。とは言いながら、まずは、アルゴリズムの動きを説明するときにアルゴリズム設計者がよく使う仮想的なマシンを見ていきましょう。

◆ 2-1 計算機における重要な考え

1930年代半ば、イギリスの数学者、Alan Turing（アラン チューリング）は「計算可能数とその決定問題への応用[*1]」

*1 Entscheidungsproblem（エントシャイドゥングプロブレム）（決定問題）はDavid Hilbertが提起した問題です。基本的には、数学的な式が真か偽かを判断できるアルゴリズムが存在するかを問いています。Turing（と彼の指導教授Alonzo Church）はそのようなアルゴリズムは存在できないことを示しました。

と題した論文を発表しました。これは、さまざまな面で現代の計算機科学の基礎を作り上げました。彼の仮想的な**チューリングマシン**は、直感的に理解し易いため、計算理論において中心的概念になっています。チューリングマシンは、無限に長いテープ上を動きながら、読み書きのみができる単純な仮想の装置です。マシンの実際の振る舞いはさまざまで、それぞれ有限状態マシン★1と呼ばれています。それは、有限個のある状態（そのうちのいくつかは終了状態を示します）と、読み書きの指令や異なる状態へ切り替える指令などが書かれた記号からなっています。このマシンは、（「もし私が状態4になり、Xを見たとき、私は左に一歩進み、Yを書き込み、状態を9に切り替える」といった）ルールの集まりとも考えることができます。このマシンは単純に見えますが、どんなアルゴリズムでも実現できる驚くべき力を備えています。大半のコンピュータサイエンティストは、このマシンが私たちが考える計算の本質を捉えていると考えています▼1。

アルゴリズムとは、与えられた問題を解くために必要な、ある有限個のステップ（繰り返しや条件分岐を含む）からなる手順や計算方法のことです。一方、アルゴリズムでどんな問題を解くのかを定式化して説明しているのがチューリングマシンです*2。その定式化は、どの問題が解けるのかどうかを議論するときによく使われています（本章の後半と第11章で、適度な時間で解けるのかまったく解けないのかを扱います）。しかし、アルゴリズムの効率をより詳細に分析するとき、実際にはチューリングマシンから議論を始めることはほとんどありません。紙のテープを巻き取る代わりに、直接アクセスできる大きなメモリを使います。このマシンは、**ランダムアクセスマシン**として広く知られています。

ランダムアクセスマシンの定式化は、少し複雑になるかもしれませんが、アルゴリズムの解析でごまかさないように、その能力の限界について知っておく必要があります。ランダムアクセスマシンは、以下の特徴を持つ、1つのプロセッサを持った標準的なコンピュータを仮想化・単純化したものです。

- マシンは並列実行できず、指令を1つずつ実行する
- 算術演算、比較、メモリアクセスなど標準的な基本演算は、（おそらくそれぞれ異なるが）すべて定数時間がかかる。それより複雑なソートのような演算はできない

★訳注1 有限オートマトンと同義で、有限状態機械や有限状態マシンとも訳されます。

▼監訳注1 計算とは何か、という本質的な問いに応えるために、計算機科学者はいくつかの計算モデルを提案してきました。チューリングマシンはその1つで、計算を状態の遷移と捉えモデル化したものです。この他に、命令型計算モデルや関数型計算モデルなども体系化されていて、実際に使われているプログラミング言語の設計思想とも関係しています。この話題について詳しく知りたい方には、猪股俊光、山田敬三（共著）「計算モデルとプログラミング」（森北出版、2019）をお勧めします。

＊2 中には問題を一切解かず、ずっと動き続けるだけのチューリングマシンも存在します。こういうものは、アルゴリズムとは呼びませんが、プログラムとは呼びます。

- マシン内の1ワード（定数時間で扱うことができる値の大きさ）は無限大ではないが、変数が使うメモリも含め、解いている問題を表現するのに使われるすべてのメモリの場所を指定できるほど十分大きい

場合によっては、もっと詳細が必要になるかもしれませんが、現時点ではこのくらいの理解で大丈夫でしょう。

さて、アルゴリズムやそのアルゴリズムを走らせる仮想的なハードウェアがどんなものか、少し感覚がつかめてきたでしょうか。パズルの最後のピースは、問題の記法です。私たちにとっては、**問題**とは入力と出力の間の関係です。この表現は実に言い得て妙です。数学における関係は、ペアの集合として表されます。どの出力がどの入力に対応しているのかという関係ですね。この関係性を決めることで、私たちは問題を解決してきました。例えば、ソートの問題は、2つのシーケンスの集合AとBの間の関係として捉えることができます*3。ソートをどのように実行するか（これがアルゴリズムですが）を気にせずに、ある入力シーケンス（Aの要素）に対して、どのシーケンスが出力（Bの要素）として相応しいかだけを決めます。例えば、出力シーケンスは入力シーケンスと同じ要素で構成され、出力シーケンスの要素は昇順（各要素が前の要素以上）であると表すことができます。Aの各要素、つまり各入力シーケンスは**問題インスタンス**★2と呼ばれ、問題ではありません。実際の問題は、入力の集合と出力の集合の間の関係であることに注意してください。

マシンに問題を解かせるために、この入力を0と1に**エンコード**（翻訳または符号化）する必要があります。あまり詳細を気にする必要はありませんが、これは重要な考え方です。（次章で説明する）実行時間の計算量の概念は、この問題のインスタンスの大きさを理解することに基づいています。ここで言う問題の大きさとは、問題をエンコードした結果を記録するのに必要なメモリの量です▼2。これから説明していくことですが、このエンコードには通常、厳密さはあまり必要ありません。

◆ 2-2 漸近記法

第1章のappendとinsertの違いの例を覚えていますでしょうか？ どういうわけか、リスト

＊3 入力と出力が同じ種類なので、実際は、AとAの関係を指定することになります。

★訳注2 問題インスタンスとは、問題（Problem）に対して、具体的な問題例（instance）というニュアンスです。

▼監訳注2 数字を表現するために何桁必要かを考えてみるといいでしょう。10進数を使っていると2も4も1桁で表現できますが、2進数では2は10なので2桁必要ですし、4は100なので3桁必要になります。この違いが後ほど重要な意味を持つことになります。

の末尾に要素を加えることは、リストの先頭に挿入することよりも、スケール[★3]していたのです。次の「ブラックボックス」コラムlistの説明を参照してください。このような組み込み演算はC言語で書かれていますが、list.appendをPythonだけで再実装する場合を考えてみてください。その新しいバージョンの演算が元の演算より50倍遅かったとします。その遅い、Pythonだけで書いたappendを使ったコードをとても遅いマシンで走らせる一方、insertを使った最適化された速いコードを1,000倍速いマシンで走らせるとしましょう。いま、insertを使ったコードは、50,000倍も速いアドバンテージがあるわけです。この2つの実装を100,000の数字を挿入して比較したら、どうなると思いますか？

直感的には「速い」解法が勝って当然と思えるかもしれませんが、この「速さ」とは、単なる定数項なのです。「速い」解法は、「遅い」解法より実行時間が増加していく割合が大きいです。この例では、遅いマシン上で実行されたPythonコードの方が、実際には「速い」解法の半分の時間で終わっています。遅いマシン上のPythonコードは、速いマシン上でのCバージョンの2,000倍も速かったのです。これは、実行時間が約1分と1日半かかるくらいの差です！

問題の規模が大きくなるにつれ、（プログラミング言語全般の性能やハードウェアのスピードなどに関する）定数項と実行時間の増加率を区別することは、アルゴリズムの分野において非常に大事なことです。

ブラックボックス list

Pythonのリストは、伝統的なコンピュータサイエンスの意味でのリストではありません。それを理解すれば、なぜappendが圧倒的にinsertより効率的なのかが分かります。従来のリスト（いわゆる連結リスト）は、（最後のノードを例外として）各ノードが次のノードを参照し続けている、連続したノードとして実装されています。単純な実装は次のようになります。

```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next
```

リストを作るには、次のようにすべてのノードを指定します。

```
>>> L = Node("a", Node("b", Node("c", Node("d"))))
>>> L.next.next.value
'c'
```

これは、いわゆる片方向リストと呼ばれるもので、各ノードが前のノードも参照しているものは双方向リストと呼ばれています。

★訳注3 本書では、「スケールする」という言葉を、問題インスタンスの大きさが増大してもアルゴリズムがそれに適応できる、という意味で用いています。逆に入力に対して指数時間や階乗時間で実行されるアルゴリズムはスケールしないと言えます。

Pythonのlist型の内部の実装は、少し異なります。listは、複数の個別のノードが互いに参照し合うのではなく、基本的には配列として知られている、1つの隣接したメモリの塊です。このため、連結リストとは重要な違いが生まれます。例えば、リストの内部を走査する処理は（連結リストの一部のオーバーヘッドを除いて）どちらも同じくらいの効率ですが、特定のインデックスの要素に直接アクセスするのは、配列の方がはるかに効率的です。配列では、要素の位置を計算して、その要素のメモリの位置に直接アクセスできるためです。一方、連結リストでは、リストを最初からたどる必要があります。

一方、思わぬ違いが現れたのは、insertに関するものでした。連結リストでは、insertしたい場所が分かれば、計算コストは安く済みます。リストに含まれる要素の数に関係なく、ほぼ同じ時間がかかります。配列の場合はそうではありません。配列でinsertを行うには、insertする要素の右側にあるすべての要素を移動する必要があり、必要に応じてすべての要素をより大きな配列に移動する場合もあります。一方、appendするためによく使われるのが、動的配列または動的ベクトルと呼ばれるものです＊4。これは、十分大きな配列を割り当てておき、オーバーフローする★4たびに線形時間でもう一度十分大きな配列を割り当てることです。一見、appendもinsertと変わらないくらい非効率に見えるかもしれません。どちらの場合も、多数の要素を移さなければならないリスクがあります。主な違いは、appendの方がこのような操作を必要とする頻度が少ない点です★5。実際には、ある配列を一定の割合（20パーセントまたは100パーセントでもいいですが）で余裕のある大きい配列に常に移動させ続けられれば、appendで使われる平均コストは一定にできます▼3。

ギリシャ語はちんぷんかんぷん！★6

漸近記法は（いくつかのバリエーションがありながら）19世紀後半から使われてきており、アルゴリズムとデータ構造の分析には必須なツールです。核となる考え方は、入力サイズをパラメータとして、私たちが分析するリソース（時間やメモリなど）を関数で表すことです。例えば、$T(n) = 2.4n + 7$という実行時間で動くプログラムがあるとします。

すぐさまある重要な疑問が浮かびます。この時間と入力サイズの単位は何でしょうか？ 実行時間は秒やミリ秒単位で測り、問題の大きさはビットやメガバイトを使えば済むように見えるかもしれません。ここでちょっと驚きの答えが待っています。実は単位は結果にまったく影響がを与えません。木星の自転周期で時間を測り、問題の大きさをキログラムで計っても（例えば、使っているハードディスクの重さでも）、まったく関係ないのです。ここでは、実装の細かい部分は無視するという考え方が反映されているからです。なので、漸近記法はこうした

＊4 配列の先頭にオブジェクトを挿入するための「すぐ使える」解法は、第5章の「ブラックボックス」コラム dequeをご覧ください。

★訳注4 配列からはみ出ること。

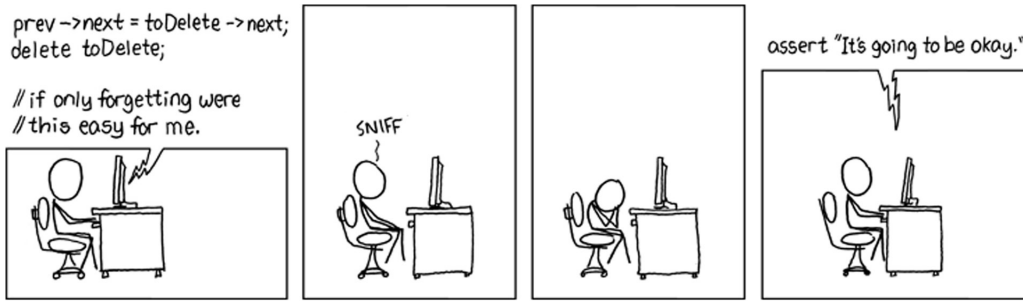
★訳注5 insertは必ず要素を移動する必要がありますが、appendはオーバーフローしたときのみ要素を移動する必要があるので、頻度が少ない、ということです。

▼監訳注3 sys.getsizeof関数を使うとオブジェクトがどれくらいのメモリを消費しているかが分かります。リストを引数にすると、メモリの消費量がサイズとは直接関係しないことが分かります。同じサイズのリストでもメモリの消費量が若干異なるなど、内部ではいろいろな処理がされているので、よほど興味がある方以外はあまり深入りしないほうがいいでしょう。

★訳注6 原文は"It's Greek to me"というタイトルです。まるで外国語のように「意味が分からない！」というのが直訳です。しかし、ここでは、本セクションが漸近記法のギリシャ文字ばかりで難しく見えることにもかけています。

要素をすべて無視します（問題の大きさは通常正の整数としていますが）。

よくやっていることは、ある基本的な操作の実行回数を実行時間にします。一方、問題の大きさは、扱う項目の数（例えば、ソートされる整数の数）や、問題の入力値を適切な方法でエンコードするのに必要なビット数[★7]となります。



// こんなに忘れることが
// 簡単だったらなぁ

（鼻をすする）

assert "きっと大丈夫"

忘れたいこと。もちろん、assertは動かない[★8]（https://xkcd.com/379）

NOTE

たいてい、問題と解をビットパターンとしてエンコードする方法は、極端でない限り、漸近的実行時間にほとんど影響しません。極端な例として、整数を1進数で表すこと（1=1, 2=11, 3=111など）は避けてください。

漸近記法は、ギリシャ文字で書かれたたくさんの演算子で成り立っています。最も重要で、本書で使うのは、O（もともとはオミクロンというギリシャ文字でしたが、通常「ビッグオー」と呼ばれています）、Ω（オメガ）、Θ（シータ）の3つだけです。O 演算子の定義は他の2つの文字の土台となります。ある関数 $g(n)$ に対する $O(g)$ という表現は、関数の集合を示しています。次の条件を満たすのであれば、関数 $f(n)$ がこの集合内にあることになります。

★訳注7 第3章のコラム「擬似多項式時間」にて素数を判定する問題がこの例に当たります。

★訳注8 1コマ目：連結リストを書いている主人公Cueballが、コメント欄に「こんなに忘れることが簡単だったらなぁ」と連結リストの間にあるアイテムが簡単に消せることと比べて、自分の忘れたい記憶が消せないことを嘆いている。
4コマ目：'assert "it's going to be okay"'という必ずassertが通ってしまう、意味のないコマンドを書く。このassertは、通常コードが予想通りに動いているかをチェックするのに使われる。このassert文は確実に通り、True（=大丈夫）が返される一方、タイトルの「忘れること」ができるのかが「きっと大丈夫か」どうかについては、assertできない（=定かではない）。