プログラミング応用 第1回: アルゴリズムの基礎

清水 伸高 (塩浦研 助教)

June 5th, 2025

アルゴリズムの計算量とオーダー記法

アルゴリズムと計算量

- **アルゴリズム**: 問題を解くための計算の手続き
  - 同じ問題を解くにも様々なアルゴリズムが考えられますが, その中で最も効率的なものが望ましい
  - そこでアルゴリズムの効率性を定量的に測る尺度として**計算量** の概念を定義
  - 計算量にも様々なものがあり, 例えば時間計算量や空間計算量などがある
- **時間計算量** : サイズnの任意の入力に対し, 計算が終わるまでにかかるステップ数
- **空間計算量** : サイズnの任意の入力に対し, 計算が終わるまでに使用したメモリのサイズ
  - サイズnの全ての入力について**最悪の**計算量を考える (最悪時計算量)

本講義では特に断りのない限り, 計算量と言えば最悪時時間計算量を指すものとし, 基本的な演算(加算や掛け算など)の回数によって測ることとする.

例: 1からnまでの総和を求める関数

問題

与えられた $n \in \mathbb{N}$ に対し, $1 + 2 + \cdots + n$ を出力せよ.

愚直に計算すると, 以下のコードになる:

```
1   def sum_to_n(n):
2       S=0
3       for i in range(1,n+1):
4           S=S+i
5       return S
```

この関数は `S=0` と `S=S+i` のそれぞれで変数 `S` の値を更新しています. 計算回数は `S=0` の箇所では1回, `S=S+i` はn回実行されるので, この関数の計算量は$n + 1$となる.

例: 1からnまでの総和を求める関数

問題

与えられた $n \in \mathbb{N}$ に対し, $1 + 2 + \cdots + n$ を出力せよ.

等差数列の和の公式を使うと, 以下のように書ける:

```
1   def sum_to_n(n):
2     return n*(n+1)//2
```

`n*(n+1)//2` では, 1回の足し算, 1回の掛け算, 1回の割り算で計算するので, 計算回数は3となる. 例1と同じ値を計算しているが, n が大きくなった時はこちらの方が計算回数が少ないので高速である. つまり, このアルゴリズムの方が前のものより効率的であるといえる.

オーダー記法

アルゴリズムの効率性を議論するために不可欠な**関数のオーダー**の概念を説明する.

- 関数 $f\colon \mathbb{N} \to \mathbb{N}$ を考える.
  - $f(n)$は, サイズnの入力を受け取ったときのアルゴリズムの計算回数と思えばよい.
- アルゴリズムの効率性を議論する際は $n \to \infty$ における$f(n)$の**増加のスピード**を考える.
  - 例えば, $f(n) = n$よりも, $f'(n) = n^2$の方が速く増加する.
- 本質的にはアイデアの同じアルゴリズムであっても, 実際の計算回数は実装の細かい部分にも左右される.
  - このような細かい違いは無視したい (同じアイデアのアルゴリズムは同じ計算量として評価したい)
  - そこで, $f(n)$の増加スピードにおいて, **定数倍は無視**する.

オーダー記法

定義

関数の支配項 (最も大きい部分) を抜き出し, その係数を1としたものをその関数の**オーダー**と呼び, 関数 $f(n)$のオーダーが$g(n)$であるとき, $f(n) = O(g(n))$ と書く.

- 例1: $f(n) = 3n^2 + 5n + 2$ の支配項は $3n^2$ であり, 係数を1とすると n^2 になる.
  - よって$3n^2 + 5n + 2$ は n^2 のオーダーであるといい, $3n^2 + 5n + 2 = O(n^2)$と書く.
- 例2: $f(n) = \sqrt{n} + \log n$ の支配項は $\sqrt{n}$ なので, $f(n) = O(\sqrt{n})$ である.

注意

「支配項」の定義が曖昧なので, この定義は厳密なものではない. 厳密なオーダーの定義は次ページで紹介

オーダー記法

厳密には, $f(n) = O(g(n))$の定義は以下のようになる:

定義

関数 $f\colon \mathbb{N} \to \mathbb{N}$ と $g\colon \mathbb{N} \to \mathbb{N}$ に対し, $f(n) = O(g(n))$ であるとは, ある定数$C > 0$と$N \in \mathbb{N}$が存在し, 任意の$n \geq N$に対して $f(n) \leq C \cdot g(n)$ が成り立つことをいう.

- 直感: どんなに大きな n を考えても, $f(n)$ は $g(n)$ の**定数倍** で上から抑えられる
  - **定数**: nに依存しない固**定**された**数**
  - $f(n)$の増加スピードが$g(n)$の増加スピードよりも**速くならない**ことを意味する
  - 小さい n ($n < N$) での大小関係は無視する (十分大きなnでの振る舞いを考える)

例. 1からnまでの和を求める関数

```
1   def sum_to_n(n):
2       S=0
3       for i in range(n):
4           S=S+i
5       return S
```

- この関数の計算量は $n+1$ である
  - 2行目の代入操作は1回
  - 4行目の `S=S+i` はn回実行される
- よって, 計算量のオーダーは $O(n)$ である.

例. 1からnまでの和を求める関数

```
1   def sum_to_n(n):
2     return n*(n+1)//2
```

- この関数の計算量は 3 である
- オーダーで表すと $O(1)$ である ($g(n) = 1$という関数も考える)
- つまり, 上記のコードの計算量は n に依存しない定数で上から抑えられることになる. 本当にそのようなことがありえるのだろうか?

注意

ここでは, 計算量を「演算の回数」によって定義していた. しかし, 実際にはコンピュータの1回の演算は**一桁同士の数字の加算や乗算**を実行できるに過ぎない. 従って, **加算や乗算などの演算は少なくともその桁数に比例した時間がかかる** (足し算の筆算を考えてると分かりやすい). nの桁数は $\lceil \log_{10} n \rceil$ であるから, 実際には上記のプログラムの計算時間は$O(\log n)$となる.

例. 行列積の計算

二つの$n \times n$行列A, Bが与えられたとき, $A \cdot B$を返す以下の関数を考える:

```
def matrix_mult(A, B):
  n = len(A)
  C = [[0]*n for _ in range(n)]
  for i in range(n):
    for j in range(n):
      for k in range(n):
        C[i][j] += A[i][k] * B[k][j]
  return C
```

上のコードでは行列積を定義通りに計算している:

$$(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j}$$

再帰関数

再帰関数

プログラミングにおいて, 関数の中で関数を呼び出すという操作を**再帰**と呼び, 再帰を行う関数を**再帰関数**という.

- 元来「再帰」という用語は様々な分野(言語学, 論理学, 数学など)で登場する用語である
  - 端的に言えば, ある概念Xを定義する際に, その記述がX自身を含むことを指す
- 例: フィボナッチ数列$(a_n)_{n\in\mathbb{N}}$の定義は

$$a_n = \begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ a_{n-1}+a_{n-2} & (n\geq 2) \end{cases}$$

これはa_nの定義の中にa_{n-1}やa_{n-2}が含まれているので, 再帰である

再帰関数の例

- 再帰関数の例として, フィボナッチ数列を求める関数を考える

```
1  def fib(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      else:
7          return fib(n - 1) + fib(n - 2)
```

- この関数は, nが0または1のときは直接値を返し, それ以外のときは再帰的に`fib(n-1)`と`fib(n-2)`を計算してその和を返す (7行目)
  - このように, 自分自身の呼び出しのことを**再帰呼び出し**という
- どんな$n > 1$であっても再帰呼び出しを繰り返すといずれ4行目の条件が満たされるので, 再帰呼び出しは必ず終了する
  - しかし, 例えば`fib(-1)`を実行しようとすると, 再帰呼び出しは`fib(-2)`, `fib(-3)`と続き, 終了しない
  - 再帰関数を実装する際は必ず終了するように気をつけなければならない

再帰関数の例

- 再帰関数の別の例として, 階乗を求める関数を考える

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      else:
5          return n * factorial(n - 1)
```

- この関数は, nが0のときは1を返し, それ以外のときは再帰的に `factorial(n-1)` を計算してその結果にnを掛けた値を返す (7行目)
- この関数も再帰呼び出しを繰り返すことで, いずれ2行目の条件 `n==0` が満たされるので, 再帰呼び出しは必ず終了する

再帰関数の応用: べき乗法

問題

与えられた$a, b \in \mathbb{N}$に対し, a^bを計算せよ (計算量は演算回数で測る).

普通に計算すると, $a, a^2, a^3, \ldots, a^b$ と順に計算していくことになるので, b回の掛け算が必要となる

```
1   def power(a, b):
2       result = 1
3       for _ in range(b):
4           result *= a
5       return result
```

定理

計算量$O(\log b)$でa^bを計算できる.

- 上のアルゴリズムの計算量$O(b)$よりはるかに高速
- これを可能にするアルゴリズムがべき乗法

べき乗法のアルゴリズム

再帰を使ってa^bを計算することを考える. 関数名を`fast_power(a, b)`とする.

- $b \in \{0, 1\}$のときは, a^bを出力すればよい (出力は1またはaなので, $O(1)$時間)

```
if b==0:
  return 1
if b==1:
  return a
```

- $b \geq 2$が偶数のとき, $a^b = a^{(b/2)} \cdot a^{(b/2)}$である. $a^{(b/2)}$を再帰的に計算すればよい.

```
if b >= 2 and b % 2 == 0:
  half = fast_power(a, b // 2) # b//2はbを2で割った商（少数部分は切り捨て）
  return half * half
```

- それ以外 ($b \geq 2$が奇数) のとき, $a^b = a^{\frac{b-1}{2}} \cdot a^{\frac{b-1}{2}} \cdot a$である. $a^{(b-1)}$を再帰的に計算すればよい.

べき乗法の実装

```python
def fast_power(a, b):
    if b == 0:
        return 1
    if b == 1:
        return a

    if b >= 2 and b % 2 == 0:
        half = fast_power(a, b // 2)
        return half * half
    else:
        half = fast_power(a, (b - 1) // 2)
        return half * half * a
```

- この関数の計算量(演算回数)はどうなるだろうか?
  - $b \geq 2$が偶数でも奇数であっても, 次の呼び出しにおいてbは高々半分になる
  - 従って, 呼び出しの回数は高々$\log_2 b$回で抑えられる
  - 各呼び出しでの演算回数は高々4回である

べき乗法の応用例: 行列のべき乗とフィボナッチ数列

- べき乗法ではa^bを計算したが, ここでaは数である必要はない. 例えば, 行列Aに対して, $O(\log b)$回の行列乗算でA^bを計算することもできる.
- この性質を使ってフィボナッチ数列を高速に計算することができる.

定義

フィボナッチ数列$(a_n)_{n\in\mathbb{N}}$は以下の漸化式で定まる数列$(a_n)_{n\geq 0}$である:

$$a_n = \begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ a_{n-1}+a_{n-2} & (n\geq 2) \end{cases}$$

愚直に$a_0, a_1, \ldots,$の順に計算すると, a_nの計算に$O(n)$の時間がかかる.

フィボナッチ数列の行列表示

- フィボナッチ数列は行列を使って次のように表現できる (ただし$n \geq 1$)

$$\begin{pmatrix} a_n \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

演習

この事実を証明せよ.

- a_nは, 行列$A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$に対し, A^nを計算することによって求められる.
- べき乗法を使ってA^nを計算すると, フィボナッチ数列のn番目の項を$O(\log n)$で計算できる.

再帰関数の応用: 掛け算

問題

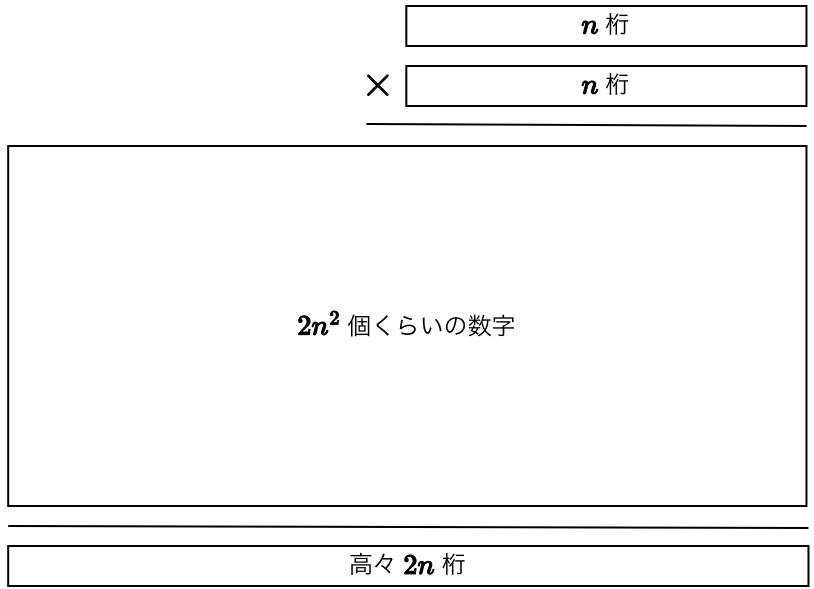
与えられた2つのn桁の自然数$a, b \in \mathbb{N}$に対し, $a \times b$ を計算せよ. 計算量は一桁同士の掛け算と足し算の回数で測る.

小学校で習うアルゴリズム: **筆算**

$$
\begin{array}{rrrrrrr}
 & & & 1 & 2 & 3 & 4 \\
 & & \times & 6 & 7 & 8 & 9 \\
\hline
 & & & 1 & 1 & 1 & 0 & 6 & \leftarrow 1234 \times 9 \\
 & & & 9 & 8 & 7 & 2 & 0 & \leftarrow 1234 \times 80 \\
 & & 8 & 6 & 3 & 8 & 0 & 0 & \leftarrow 1234 \times 700 \\
 & 7 & 4 & 0 & 4 & 0 & 0 & 0 & \leftarrow 1234 \times 6000 \\
\hline
 & 8 & 3 & 7 & 7 & 6 & 2 & 6
\end{array}
$$

再帰関数の応用: 掛け算

一般に, n桁同士の積を筆算すると $O(n^2)$ 時間かかる.



再帰に基づく掛け算

- 数字aの各桁の数字を一の位から順に$a_0, a_1, \ldots, a_{n-1} \in \{0, 1, \ldots, 9\}$とする. このとき

$$a = a_0 + 10a_1 + 10^2 a_2 + \cdots + 10^{n-1} a_{n-1} = \sum_{i=0}^{n-1} a_i 10^i.$$

- これを $a = [a_0, a_1, \ldots, a_{n-1}]$ と表す.
  - 例えば, $1234 = [1, 2, 3, 4]$, $9876 = [9, 8, 7, 6]$である.
- 簡単のため, nは2ベキ, すなわち$n = 2^k$の形であるとする
  - そうでない場合は, 最後の桁に0を挿入して桁数を2べきにする. これによって桁数は高々2倍となる.
  - 例えば $a = 12345$ ならば, $a = [1, 2, 3, 4, 5, 0, 0, 0]$ とする

再帰に基づく掛け算

- $c = [c_0, \ldots, c_{n-1}]$に対し
  - 左半分の桁からなる数字を$c_L = [c_0, \ldots, c_{n/2-1}]$
  - 右半分の桁からなる数字を$c_R = [c_{n/2}, \ldots, c_{n-1}]$とする
  - 例えば$c = 1234$ならば, $c_L = [1, 2]$, $c_R = [3, 4]$である.
- 特に, $c = c_L + c_R 10^{n/2}$と表せる.
- 求めたい積$a \times b$は

$$
\begin{aligned}
a \times b &= (a_L + a_R 10^{n/2})(b_L + b_R 10^{n/2}) \\
&= a_L b_L + (a_L b_R + a_R b_L) \times 10^{n/2} + a_R b_R \times 10^n
\end{aligned}
$$

再帰に基づく掛け算

- 求めたい積$a \times b$は

$$a \times b = (a_L + a_R 10^{n/2})(b_L + b_R 10^{n/2}) = a_L b_L + (a_L b_R + a_R b_L) \times 10^{n/2} + a_R b_R \times 10^n$$

  - $\times 10^{n/2}$ や $\times 10^n$ の計算は簡単 (後ろに0を追加するだけ)
  - 足し算は$O(n)$時間で計算できる
- したがって, $a \times b$を計算するためには四つの積 $a_L b_L$, $a_L b_R$, $a_R b_L$, $a_R b_R$ を計算すればよい

問題

四つの積 $a_L b_L, a_L b_R, a_R b_L, a_R b_R$ を再帰的に計算すると?

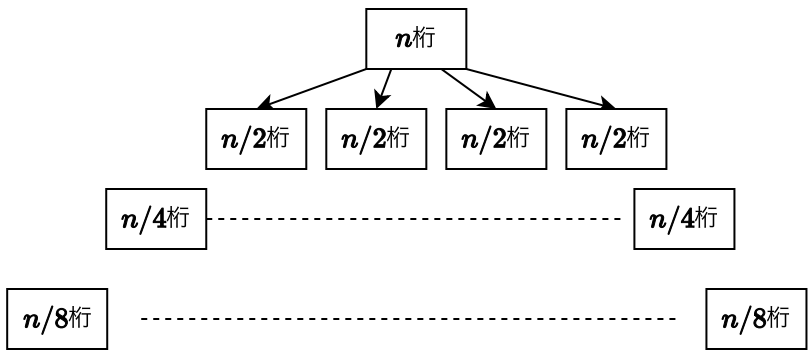
一度の再帰呼び出しで桁数は半分になるが, 再帰呼び出しは4回行われる.

再帰に基づく掛け算

問題

四つの積 $a_L b_L, a_L b_R, a_R b_L, a_R b_R$ を再帰的に計算すると?

- 一度の再帰呼び出しで桁数は半分になるが, 再帰呼び出しは4回行われる.



- 各四角形(=呼び出し)内部では, 足し算が行われているので, $O($桁数$)$の時間がかかる.

再帰に基づく掛け算

全体の計算量はいくつになるだろうか?

- n桁の呼び出し : 1回
- $n/2$桁の呼び出し : 4回
- $n/4$桁の呼び出し : 16回
- $n/8$桁の呼び出し : 64回
- $\cdots$
- 1桁の呼び出し : $4^{\log_2 n} = n^2$回

全体の計算量は $1 + 4 + 16 + \cdots + 4^{\log_2 n} = O(n^2)$. つまり筆算と同じオーダー.

カラツバ法

定理

n桁同士の掛け算を$O(n^{\log_2 3}) \approx O(n^{1.585})$時間で計算できる.

アイデア:

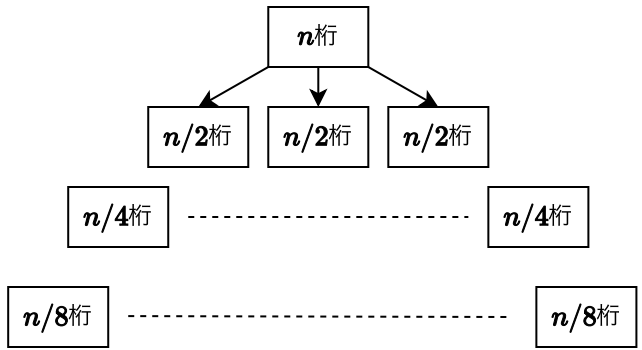
$$a \times b = a_L b_L + (a_L b_R + a_R b_L) \times 10^{n/2} + a_R b_R \times 10^n$$

三つの値 $a_L b_L, a_L b_R + a_R b_L, a_R b_R$ を**3回**の再帰呼び出しで計算する

1. $a_L b_L$と$a_R b_R$を計算する (**2回**の再帰呼び出し)
2. $(a_L + a_R)(b_L + b_R)$を計算する (**1回**の再帰呼び出し)
3. $a_L b_R + a_R b_L = (a_L + a_R)(b_L + b_R) - a_L b_L - a_R b_R$ を計算する (再帰呼び出しは不要)

カラツバ法

- $(a_L + a_R)(b_L + b_R)$の計算は高々$n/2 + 1$桁同士の掛け算



- 各四角形(=呼び出し)内部では, 足し算が行われているので, $O(\text{桁数})$の時間がかかる.

全体の計算量 = 全ての呼び出し内部の計算量の総和

再帰に基づく掛け算

全体の計算量はいくつになるだろうか?

- n桁の呼び出し : 1回
- $n/2$桁の呼び出し : 3回
- $n/4$桁の呼び出し : 9回
- $n/8$桁の呼び出し : 27回
- $\cdots$
- 1桁の呼び出し : $3^{\log_2 n} = n^{\log_2 3}$回

全体の計算量は $1 + 3 + 3^2 + \cdots + 3^{\log_2 n} = O(n^{\log_2 3})$.

まとめ

- **計算量**の概念の紹介
  - アルゴリズムの効率性を**オーダー記法**で表現
- **再帰**の概念
  - 定義の中で自分自身を参照する関数を**再帰関数**と呼ぶ
  - 漸化式をプログラミングで表現したようなもの
  - 再帰関数の例: フィボナッチ数列, 階乗
- 再帰関数の応用
  - **べき乗法**: a^bを$O(\log b)$時間で計算するアルゴリズム
  - 行列のべき乗を使ったフィボナッチ数列の計算
  - **カラツバ法**: n桁同士の掛け算を$O(n^{\log_2 3})$時間で計算するアルゴリズム