■6 群（基礎理論とハードウェア）- 3 編（アルゴリズムとデータ構造）

# 1 章 アルゴリズムとアルゴリズム解析

（執筆者：久野　靖）[2012 年 7 月 受領]

## ■概要■

アルゴリズムとは，プログラムの本質部分をなす「手順」を意味する．ただし，ある「手順」がアルゴリズムであるためには，有限性，非曖昧性，停止性という条件がつけられることが通例である．

アルゴリズムは何らかの計算モデルに基づく動作を規定するので，計算モデルとして何を取り上げるかも，アルゴリズムの記述に際しては重要な点となる．ただし，多くの計算モデルはチューリング完全という点で同等であることもまた，示されている．代表的な計算モデルとして，チューリング機械，ラムダ計算，レジスタマシンなどが挙げられる．

アルゴリズムの表記方法としては，計算モデルそのものによる記述は繁雑なため，実コード，疑似コード，フローチャートに代表される各種の図法などが主に使われる．

アルゴリズムの良し悪しに関する指標としては，時間計算量，領域計算量などの考え方が主に使われる．アルゴリズムを分析してこれらの指標やその他の性質を調べる作業をアルゴリズム解析と呼ぶ.

## 【本章の構成】

1-1 節ではアルゴリズムとは何かについて，基本的な考え方を紹介する．続いて 1-2 節では，アルゴリズムの土台となる計算モデルについて，ラムダ計算とレジスタマシンを挙げて説明する．ほかの主要な計算モデルであるチューリング機械については，6 群 2 編 4 章に解説されている．更に，疑似コードや図法などのアルゴリズムを表記する方式についても説明する．1-3 節では，アルゴリズムの複雑さを測るという問題を扱う手法であるアルゴリズム解析と計算量の考え方を紹介する．

 © 電子情報通信学会 *2012* 

■6 群 - 3 編 - 1 章

## 1-1 アルゴリズム

(執筆者：久野　靖)[2012 年 7 月 受領]

**アルゴリズム（algorithm）** [1)]という言葉は，今日ではコンピュータと深く結び付いて使われているが，これはコンピュータを用いた計算（ないし情報処理）をアルゴリズムと考えて定式化することで，多くの有用なことがらが明らかになるからである．しかし，アルゴリズムという言葉はコンピュータ以前から存在している．例えば，高校生は数学で「ユークリッドの互除法」によって**最大公約数（GCD, greatest common divisor）**を求める方法を学ぶことになっているが，この計算手順はしばしば，単に**ユークリッドのアルゴリズム（Euclid's algorithm）**と記される．ユークリッドのアルゴリズム（もともとユークリッドが示したとされる，引き算を使うバージョン）を Ruby 言語で記したものを図 1･1 に示しておく．

```
def gcd(x, y)
  while x != y
    if x > y
      x = x - y
    else
      y = y - x
    end
  end
  x
end
```

**図 1･1　Ruby による GCD プログラム**

アルゴリズムとは何かという定義については，多くの先人が様々に表現しているので簡単ではないが，ここでは次の条件を満たすようなものであるとする．

(1) **有限性（finiteness）** — 有限個のステップの集まりとして表現されている．

(2) **厳密性（definiteness）** — それぞれのステップについて，動作が厳密に定まっている．

(3) **有効性（effectiveness）** — 実行させると常に有限のステップで停止し，求める答えを返す．

今日のコンピュータのプログラムは（特に手続き型言語の場合），ステップを順番に実行するというかたちの定式化になじみやすい（もともと今日のコンピュータは，その内部で命令語をステップとして順番に実行していくというモデルで自然に理解される）．そして，コンピュータに読み込ませる以上，その表現は有限でなければならない．また，プログラミング言語による記述は，それぞれの記述がどのように動作するか（そしてステップがどのような順番で実行されて行くか），厳密に定義されている．つまり (1) と (2) については，手順をプログラムとして書き表したときに自然に達成される性質だといえる．

一方，(3) についてはどのようなプログラムでも成り立つということはなく，誰にでも「止まらない」プログラムを書いてしまった経験があるはずである．(3) の条件を除外し，(1)，(2)

を満たすもののことを**計算手順（computational procedure）**，ないし単に手順と呼ぶことにする．

(2) と関連のある概念として，**決定性アルゴリズム（deterministic algorithm）**と**非決定性アルゴリズム（nondeterministic algorithm）**の区分がある．決定性アルゴリズムとは，各ステップにおいて次のステップが**一意（unique）**に定まるものをいい，非決定性アルゴリズムとは次のステップが複数である場合があるものをいう．

非決定性であることは，曖昧さを意味しない．すなわち,「次に実行すべきステップとそこで扱っている値（状態）の集合」を保持し，各ステップの動作を「状態の集合に基づいて次の状態の集合を定める」ものとすることで，非決定性アルゴリズムを決定性アルゴリズムに変換できる．これは**非決定性オートマトン（nondeterministic automata）**の扱いと類似している（非決定性オートマトン→ 6 群 2 編 2 章）．

一方で，上記の方法で非決定性アルゴリズムを実装し動かすことは処理時間の点で現実的でないことが多い．そこで，乱数を用いてランダムに選択を行い，いずれかの選択のみを実行することも行われる．例えば，クイックソートでピボットとなる要素をランダムに選択することもその一例である．このような，乱数を用いて選択を行うアルゴリズムのことを**乱択アルゴリズム（randomized algorithm）**[2)] と呼ぶ（乱択計算→ 6 群 2 編 6 章）．

「ある手順を与えられて，その手順が停止するか否かを定める問題」のことを**停止問題（halting prolem）**と呼ぶが，この問題は**決定不能（unsolvable）**であることが知られている（停止問題→ 6 群 2 編 4 章）．そこでそれに代わる方法として，与えられた手順が求める答えを返すか否かの**検証（validation）**を行う手法についても，多くの研究がなされている（ソフトウェア検証→ 7 群 1 編 2 章）．

**■参考文献**

1) Donald E. Knuth, “The Art of Computer Programming 2nd ed.,” vol.1/Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1975.
2) 渡辺 治, “確率的アルゴリズム,” 情報処理学会誌, vol.24, no.4, pp.372-377, 1983.

■6 群 - 3 編 - 1 章

# 1-2 計算モデルとアルゴリズムの表記

(執筆者：久野　靖) [2012 年 7 月 受領]

## 1-2-1 計算モデルの位置づけ

具体的にアルゴリズム(ないし計算手順)を書き表すことを考えた場合，その記述には何が含まれてよく，何は含まれるべきでないのかが明確に定まっている必要がある．それには，アルゴリズムが書き表す計算そのものがどのようなものであるか，すなわち**計算モデル (computational model)** を定義したうえで，それを書き表す方法を定めることになる．

最も基本的な計算モデルの一つに，**チューリング機械 (Turing machine)** (チューリング機械→ 6 群 2 編 4 章) がある．すべてのチューリング機械は**万能チューリング機械 (universal Turing machine)** によって模倣可能であるなど，多くの面から，チューリング機械のもつ計算能力は今日のコンピュータがもつ能力と原理的には同等だとされている．

このため，ある計算手順の表記法 (特にプログラミング言語) がチューリング機械と同等の動作を記述可能であることを**チューリング完全 (Turing complete)** と呼び，その表記法が一定水準の (ほかのプログラミング言語全般と同等の) 能力をもつことの基準として使われる．よく使われる表記や言語のなかでも，**正規表現 (regular expression)** (正規表現→ 6 群 4 編 3 章) や **SQL** (SQL → 7 群 5 編) など，任意回数の繰り返しまたは再帰を書く能力をもたないものはチューリング完全ではない．

このように，チューリング機械は計算モデルとしては重要な位置を占めるが，その命令の記述方法が繁雑であることから，それに直接基づいて様々なアルゴリズム記述を行うことは (計算モデル自身の研究以外では) 通常ない．

## 1-2-2 ラムダ計算

チューリング完全であることが示されている計算モデルの一つに，**ラムダ計算 (lambda calculus)** [3, 6]がある．ラムダ計算はラムダ抽象によって表現される**関数 (function)** とその適用に基づく手順の表記法であり，**関数型プログラミング言語 (functional programming languages)** となぞらえて理解しやすい．また**型付きラムダ計算 (typed lambda calculus)** (型付きラムダ計算→ 6 群 4 編 2 章) など型理論の土台としても多く参照される．

基本的なラムダ計算では，**ラムダ式 (lambda term)** を次のように**変数 (variable)**，**ラムダ抽象 (lambda abstraction)**，**適用 (applicaiton)** の三つの記法を組み合わせて定義する．

$$
\begin{aligned}
t ::= &\ x && (\text{変数}) \\
|&\ \lambda x\,.\,t && (\text{ラムダ抽象}) \\
|&\ t\ t && (\text{適用})
\end{aligned}
$$

ラムダ抽象はいわゆる関数を表し，適用は関数に引数を与えて実行することを表す (このほかに，順序を表すかっこを適宜用いる)．その意味するところは，$\lambda$ の次に記された変数が引数であり，適用時には，．の後ろに書かれた項の中の引数を適用時に渡された式で置き換えることを意味する．

ラムダ計算の興味深いところは，上で定義した項だけで計算と計算対象となるデータの両

方を表せる点である．これは，プログラムもまたデータであり，プログラムによって加工できるという，現代のコンピュータがもつ特性にそのまま対応していると言える．

例えば，自然数の集合やその加算をラムダ式だけで表現し定義する様子を見てみることにする．まず，**自然数（natural number）**を次のようなかたちの項によって表すものとする．これを**チャーチ数（Church numerals）**と呼ぶ．

$$
\begin{aligned}
c_0 &= \lambda s\,.\,\lambda z\,.\,z \\
c_1 &= \lambda s\,.\,\lambda z\,.\,s\;z \\
c_2 &= \lambda s\,.\,\lambda z\,.\,s\;(s\;z) \\
c_3 &= \lambda s\,.\,\lambda z\,.\,s\;(s\;(s\;z)) \\
&\ldots
\end{aligned}
$$

このとき，加算を行うラムダ式 plus は次のように定義すればよい．

$$\mathrm{plus} = \lambda m\,.\,\lambda n\,.\,\lambda s\,.\,\lambda z\,.\,m\;s\;(n\;s\;z)$$

これに対してチャーチ数の 1 と 2 を与えて適用した様子を次に示す．

$$
\begin{aligned}
&\mathrm{plus}\;c_1\;c_2 \\
&\quad = \lambda m\,.\,\lambda n\,.\,\lambda s\,.\,\lambda z\,.\,m\;s\;(n\;s\;z)\;(\lambda s\,.\,\lambda z\,.\,s\;z)\;(\lambda s\,.\,\lambda z\,.\,s\;(s\;z)) \\
&\quad = \lambda s\,.\,\lambda z\,.\,(\lambda s\,.\,\lambda z\,.\,s\;z)\;s\;((\lambda s\,.\,\lambda z\,.\,s\;(s\;z))\;s\;z) \\
&\quad = \lambda s\,.\,\lambda z\,.\,(\lambda s\,.\,\lambda z\,.\,s\;z)\;s\;(s\;(s\;z)) \\
&\quad = \lambda s\,.\,\lambda z\,.\,s\;(s\;(s\;z)) \\
&\quad = c_3
\end{aligned}
$$

確かに，1 と 2 を足した結果として 3 が得られている．

また，論理値とそれに基づく枝分かれも同様にラムダ式で定義できる．これを**チャーチ論理値（Church Booleans）**と呼ぶ．まず，真と偽を次の項によって表現する．

$$
\begin{aligned}
\mathrm{true} &= \lambda t\,.\,\lambda f\,.\,t \\
\mathrm{false} &= \lambda t\,.\,\lambda f\,.\,f
\end{aligned}
$$

次に，真偽値と更に二つの引数を受け取り，真偽値が真なら二つの引数の前者，偽なら後者を値とするラムダ式 test（if-then-else 演算子に相当）は次のように定義できる．

$$\mathrm{test} = \lambda b\,.\,\lambda y\,.\,\lambda n\,.\,b\;y\;n$$

実際に真ないし偽と二つの引数 $p$，$q$ を与えた様子を示す．

$$
\begin{aligned}
&\mathrm{test\;true\;p\;q} \\
&\quad = (\lambda b\,.\,\lambda y\,.\,\lambda n\,.\,b\;y\;n)\;(\lambda t\,.\,\lambda f\,.\,t)\;\mathrm{p\;q} \\
&\quad = (\lambda t\,.\,\lambda f\,.\,t)\;\mathrm{p\;q} \\
&\quad = \mathrm{p} \\
&\mathrm{test\;false\;p\;q}
\end{aligned}
$$

$$= (\lambda b\,.\,\lambda y\,.\,\lambda n\,.\,b\;y\;n)\;(\lambda t\,.\,\lambda f\,.\,f)\;\mathrm{p}\;\mathrm{q}$$
$$= (\lambda t\,.\,\lambda f\,.\,f)\;\mathrm{p}\;\mathrm{q}$$
$$= \mathrm{q}$$

このように，数と枝分かれまではラムダ式の項によって素直に表現できることが分かる．次に繰り返しについては，**再帰（recursion）**を用いることで表現する．無限の再帰は，例えば次のラムダ式で表現できる．

$$\mathrm{Y} \;=\; \lambda f\,.\,(\lambda x\,.\,f\,(x\;x))\;(\lambda x\,.\,f\,(x\;x))$$

これは**名前渡し（call-by-name）**の **Y コンビネータ（Y-combinator）**ないし**不動点コンビネータ（fixed point combinator）**と呼ばれ，次のように任意の関数を繰り返し適用させられる．

$$\mathrm{Y}\;\mathrm{g}$$
$$= (\lambda f\,.\,(\lambda x\,.\,f\,(x\;x))\;(\lambda x\,.\,f\,(x\;x)))\;\mathrm{g}$$
$$= (\lambda x\,.\,\mathrm{g}\,(x\;x))\;(\lambda x\,.\,\mathrm{g}\,(x\;x)))$$
$$= \mathrm{g}\;(\lambda x\,.\,\mathrm{g}\,(x\;x))\;(\lambda x\,.\,\mathrm{g}\,(x\;x)))$$
$$= \mathrm{g}\;(\mathrm{Y}\;\mathrm{g})$$

ただしこれが使えるのは，評価の順序が名前渡しの場合に限られ，値渡しだと（Y g）の部分が展開されてしまうので発散する．このため「名前渡しの」という限定がついて呼ばれる．**値渡し（call-by-value）**の Y コンビネータ／不動点コンビネータは次のようにもう少し複雑になる．

$$\mathrm{Y'} \;=\; \lambda f\,.\,(\lambda x\,.\,f\,(\lambda y\,.\,x\;x\;y))\;(\lambda x\,.\,f\,(\lambda y\,.\,x\;x\;y))$$

ラムダ計算そのものは理論的な検討のためのものであるが，これに基づいて **Lisp** 系の言語やその他の関数型のプログラミング言語が多数作られている．

プログラミング言語ではもちろん，数値やその演算，論理値やそれに基づく枝分かれなどはあらかじめ組込みの機能として備わっている．また，再帰についても，名前をつけた関数を定義し，その本体で名前を用いて自分自身を参照することで，Y コンビネータのようなものを使用しなくても記述できる．更に，Lisp など副作用を忌避しない言語では，再帰を使用しなくても済むように直接的に繰り返しを実現する機能を備えていることが普通である．

### 1-2-3 レジスタマシン

もう一つの主要な計算モデルとして，**レジスタマシン（register machine）**[2)]と呼ばれる一群のものがある．ここでレジスタと呼んでいるものは，有限個または無限個の，数値（上限のない非負整数）を格納できる場所であり，一般のコンピュータでいえばメモリに相当するイメージである．そして，ラベルのついた順番に実行される命令の列があり，それぞれの命令はレジスタに対する操作を行ったり，レジスタの値に応じて次の命令を定める（条件ジャンプ）などの機能をもつ（図 1･2）．すなわち，このモデルは実際のコンピュータの命令や動作にかなり近いものとなっている．

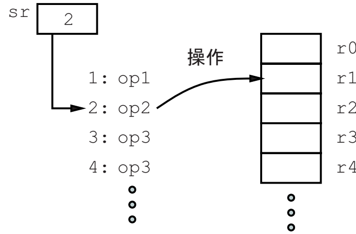


**図 1･2　レジスタマシンのモデル**

上で一群と記したのは，命令やその機能としてどのようなものを用意するかに応じて，様々な能力をもつ複数のモデルが提唱されてきたことを表している．

例えば，**カウンタマシン（counter machine）**と呼ばれるモデルの一つでは，指定したレジスタ $r$ の値を 1 増やす $INC(r)$，1 減らす $DEC(r)$，指定したレジスタ $r$ の値がゼロである場合に次の命令をラベル $z$ のものとする $JZ(r, z)$ の三つの命令のみをもつ．このほか，上記の命令の代わりにレジスタ $r$ をゼロにする $CLR(r)$，二つのレジスタ間で値をコピーする $CPY(r, s)$，二つのレジスタ値の比較による条件ジャンプ $JE(r, s, z)$ などを用いるものもある．カウンタマシンでは，レジスタの値を一つ増減する命令しかもたないため，加減算にはループが必要となる．また，配列のような機能も提供されない．

**RAM（random access machine）**と呼ばれるモデルでは，間接指定（あるレジスタ $r$ に別のレジスタの番号を入れておき，その番号のレジスタを読み書きする機能）を加えており，これによって有限の命令列でも使用できるレジスタ（メモリ）数に上限をなくすことができる．また，四則演算の命令ももつものとしており，我々の考えるコンピュータにかなり近いものとなっている．

更に，ここまでは命令列はレジスタとは別に格納されているものとしてきたが，**RASP（random access stored program machine）**と呼ばれるモデルでは命令列もレジスタに格納されることとしており，プログラムの動作によって命令を書き換えることができる．これはフォンノイマン型のモデルに相当する．

このように，RAM 以上のクラスのレジスタマシンの動作は通常の CPU の動作に近いものとなっており，したがってアルゴリズムをステップで記述し，各ステップで値の計算や変数の書き換えを行うような記法や，通常のプログラミング言語を用いてアルゴリズムを記述する場合は，この計算モデルにおおむね相当しているといえる．

### 1-2-4 実コードと擬似コード

ここまででは計算モデルについて述べてきたが，実際に説明や検討のためにアルゴリズムを表記する場合は，プログラミング言語による実コードを示したり，それに近いかたちの擬似コードで表現することが多い．これは，アルゴリズムは最終的にはプログラムとしてコンピュータ上で動作させて利用しようとする場合が多いことを考えれば自然である．

アルゴリズムを表記する目的では，**Algol 60**[1)]やその後継である **Pascal**[5)]など，いわゆる

Algol 系の言語が多く用いられてきた．これは，これらの言語では C 系統の言語と比べて，制御構造を表すのに記号ではなく **begin**，**end** などの予約語を使用し，また（Algol の場合）これらの予約語や演算記号などは「単一の固有な記号」であり，それをどのように表記するかは処理系の都合としていたため，予約語や記号を印刷物に見栄え良く表現して掲載しやすいことが主な理由だと想像される．図 1･3 に Algol 60 による最大公約数のコードを示す．

```
procedure gcd(x, y);
  integer x, y;
  begin
    while x ≠ y do
      if x > y then
        x := x - y
      else
        y := y - x
  end
```

**図 1･3**　Algol 60 **による** GCD **プログラム**

しかし，実際に動作するプログラムを掲載しようとすると，内容は自明だが言語や標準ライブラリに備わっていない部分を動くように用意するなどのため，極めて長く繁雑になることがある．このため，手順の構造はプログラミング言語と同様に示すが，そのなかの具体的な操作については自明部分を言葉で説明するだけで済ませたり，集合操作など数学で使われる記法を自由に使って記すことも多く行われる．これを**擬似コード（pseudo code）**と呼ぶ．

擬似コードは実際に動くコードを構築する前段階での検討などのための記法として使われることもあり，その場合はある程度の規格化が行われることもあるが，アルゴリズムを表記するという目的の場合には「分かれば良い」ので比較的自由に様々な書き方がなされる (図 1･4).

```
手順 gcd(x, y):
  while x ≠ y do
    x と y のうち大きい方から小さい方を引く
```

**図 1･4**　**擬似コードによる** GCD **プログラム**

### 1-2-5 フローチャート（流れ図）

**図式（diagram）**は，様々な概念を人間にとって直観的に表現する機能を提供する．このことは手順についても当てはまる．コンピュータが実用化される以前から，手順を**流れ図**ないし**フローチャート（flowchart）**として表現することは行われていた（1920 年代に技術者の間で普及したとの説がある）.

**フォン・ノイマン（John Von Neuman）**らがコンピュータの発明に従事していたとき，彼らは流れ図（彼らの用語では flow diagram）によって手順を表記し，それ以後ずっと，流れ図は手順を表記する代表的な手法の一つであり続けている[2].

流れ図では，図 1･5 に示すような，入出力・計算・分岐の箱を矢線でつなげることによって

手順を表現する．流れ図により最大公約数のアルゴリズムを記述したものを図 1･6 左に示す．

流れ図はコンピュータの入出力命令，計算命令，分岐命令（条件分岐を含む）にそのまま対応しているため，特に初期のアセンブリ言語や FORTRAN など分岐を多用するプログラミング言語には適していた．

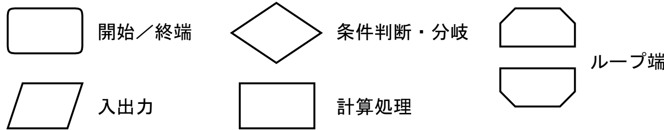


**図 1･5　フローチャートの部品**

しかし，1970 年代以降に**構造化プログラミング（structured programming）**が普及すると，構造化言語による制御構造の記述と流れ図の矢線と分岐による構造の表現は必ずしも対応しなくなり，流れ図の使用は下火になった．ループをループ端により表現することで構造化言語との対応を高めることも行われたが（図 1･6 右），これによって流れ図がもっていた簡潔さや直観的な分かりやすさが低下する面もあった．

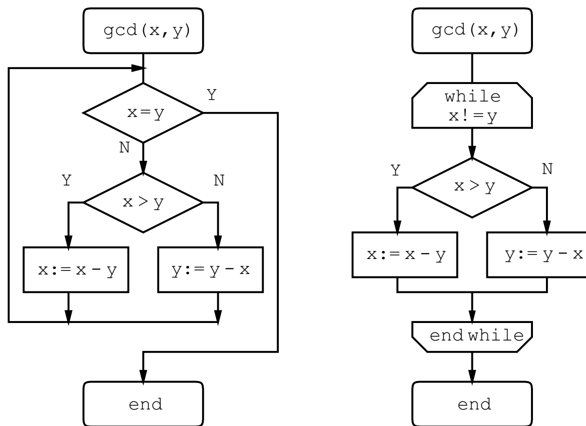


**図 1･6　フローチャート**

このような状況に対処するため，**PAD**[4] に代表される**構造化流れ図（structured flowchart）**の図法も複数考案され，それなりに使われたが，ベンダーにより異なるものが採用されるなどして，共通に合意されたものにまとまることはなかった．図 1･7 に，PAD による最大公約数アルゴリズムの記述例を示す．

やがて，ソフトウェアが大規模化し，オブジェクト指向などの上位レベルの構造が重要になるにつれ，流れ図が表現するような制御の流れの複雑さは重要な課題ではなくなってきた．また，場所を取る流れ図を書くことが好まれなくなった面もある．今日では流れ図は，コンピュータの動作原理やプログラミングの初歩的な原理を説明するような教育の場面で使用さ

れることが多い.

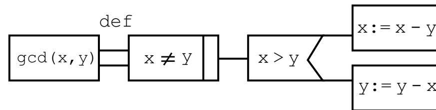


**図 1･7** PAD

**■参考文献**

1) John W. Backus et.al., “Revised report on the algorithmic language Algol 60,” Numerische Mathematik, vol.4, no.1, pp.420-453, 1962. http://www.masswerk.at/algol60/report.htm
2) Richard Bird 著, 土居範久 訳, “プログラム理論入門,” 倍風館, 1981.
3) Alonzo Church, “A Set of Postulates for the Foundation of Logic, Annals of Mathematics,” Series 2, vol.33, pp.346-366, 1932.
4) 日立ソフトウェアエンジニアリング人材開発本部教育センタ部 編, “やさしい PAD 入門,” オーム社, 1991.
5) Kathleen Jensen, Niklaus Wirth, “Pascal User Manual and Report 2nd ed.,” Springer-Verlag, 1975.
6) Benjamin C. Pierce, “Types and Programming Languages,” MIT Press, Cambridge, Mass., 2002.

■6 群 - 3 編 - 1 章

# 1-3 アルゴリズム解析と計算量

（執筆者：神林　靖，久野　靖）[2012 年 7 月 受領]

## 1-3-1 アルゴリズム解析から計算の複雑さへ

アルゴリズムの性能は，実際のコンピュータ上でそれを実行するときに動作する演算の個数や使用する記憶領域の個数を数えることで，量的に扱うことができる．このような分析のことを，**アルゴリズム解析（algorythm analysis）**という[2]．アルゴリズムが実行する演算の個数は通常，入力の**大きさ（size）**と構造に依存して定まる．例えば整列問題であれば，大きさは整列させる項目の個数である．場合によっては，同じ大きさの入力であっても，それらが異なる構造をとるために，実行される演算の個数に影響を与えることがある．例えば整列アルゴリズムのなかには，均等に乱雑な入力データに対しては効率が良いが，整列済みの入力データに対しては効率が悪くなるものもある．

そこで，アルゴリズム解析に際しては，最悪の場合を考えることが多い．アルゴリズム $A$ へのサイズ $n$ の入力が**最悪の場合の入力（worst-case input）**となるのは，サイズ $n$ のほかのすべての入力と比較したとき，$A$ が最大の演算数を実行する場合である．任意の入力 $I$ について，そのサイズを $\mathrm{size}(I)$ と記し，$I$ 上の $A$ によって実行される演算数を $\mathrm{time}(I)$ と記す．そのとき，$A$ の**最悪の場合の関数（worst-case function）**とは，以下のように定義される．

$$W_A(n) = \max\{\mathrm{time}(I) \mid I \text{ は } \mathrm{size}(I) = n \text{ なる任意の入力 }\}$$

次に，複数のアルゴリズム間の比較を議論する．問題 $P$ を解決する二つのアルゴリズム $A$, $B$ において，$n > 0$ について $W_A(n) \leq W_B(n)$ であれば，「アルゴリズム $A$ はアルゴリズム $B$ よりすぐれているか等しい最悪の場合の実行効率をもつ」という．

このようなかたちで複数のアルゴリズムを比較することで，それらのアルゴリズムが共通に解く問題そのものの難しさについて論じることができる．これを**計算の複雑さ**ないし**計算量（computational complexity）**の理論と呼ぶ．ここで計算量として，演算の数を考える場合，これを**時間計算量（time complexity）**と呼び，使用する記憶領域の数を考える場合，**空間計算量（space complexity）**と呼ぶ．

## 1-3-2 計算量解析の方法

**計算可能（computable）**であるということは，**列挙可能（enumerable）**であるということに等しい（計算可能性→ 6 群 2 編 4 章）．そして列挙問題の多くは，再帰的に定義される関数によって表現される解にまとめることができる．再帰的に定義される手続きや関数を含むプログラムに対しては，その計算量を解析するためには，**閉形式（closed form）**を発見しなければならない．どのようにして閉形式を発見できるかについて，議論しよう．

定義域が $\mathbb{N}$ で，数を計算する任意の再帰的に定義される関数 $f$ は，**漸化式（recurrence）**と呼ばれる．例えば次の漸化式の定義を考えてみる．

$$f_0 = 1$$
$$f_n = 2f_{n-1} + n$$

漸化式 $f$ を解くためには，一般項 $f_n$ の再帰的でない表現を見つけなければならない．一つの方法は，次のようにして，代入により解くことができる．この解法では，積のかたちを残すことで，一般的なパターンの発見を試みる．発見したパターンを強調するために括弧で囲んで指数で表している．

$$
\begin{aligned}
f_n &= 2f_{n-1} + n \\
&= 2^2 f_{n-2} + 2(n-1) + n \\
&= 2^3 f_{n-3} + 2^2(n-2) + 2(n-1) + n \\
&\quad\vdots \\
&= 2^{n-1} f_1 + 2^{n-2}(2) + 2^{n-2}(2) + \cdots + 2^2(n-2) + 2^1(n-1) + 2^0(n) \\
&= 2^n f_0 + 2^{n-1}(1) + 2^{n-2}(2) + \cdots + 2^2(n-2) + 2^1(n-1) + 2^0(n) \\
&= 2^n(1) + 2^{n-1}(1) + 2^{n-2}(2) + \cdots + 2^2(n-2) + 2^1(n-1) + 2^0(n)
\end{aligned}
$$

第 1 項をそのまま残し，和の残りの部分を逆にすることで次の閉形式を得ることができる．

$$
\begin{aligned}
f_n &= 2^n(1) + n + 2(n-1) + 2^2(n-2) + \cdots + 2^{n-2}(2) + 2^{n-1}(1) \\
&= 2^n + \left[2^0(n) + 2^1(n-1) + 2^2(n-2) + \cdots + 2^{n-2}(2) + 2^{n-1}(1)\right] \\
&= 2^n + \sum_{i=0}^{n-1} 2^i(n-i) \\
&= 2^n + n\sum_{i=0}^{n-1} 2^i - \sum_{i=0}^{n-1} i2^i \\
&= 2^n + n(2^n - 1) - \left(2 - n2^n + (n-1)2^{n+1}\right) \\
&= 2^n(1 + n + n - 2n + 2) - n - 2 \\
&= 3(2^n) - n - 2
\end{aligned}
$$

計算量解析において，式の閉形式が常に発見できるとはかぎらない．しかし実用的には，アルゴリズムを実行するのに必要なステップ数の近似が求まれば十分なことも多い．ある関数を用いてほかの関数の近似を求めるためには，それらを比較する手段が必要である．これから**成長率（rate of growth）**というアイディアが導かれる．

### 1-3-3 ビッグ・オー記法とその他の漸近記法

計算機科学では，定義域と余域として実数の部分集合をとる関数について考えることが多い．そこで，大きな正の数 $n$（$n$ は無限大に近づく）について，$f(n)$ と $g(n)$ を比較することによって，二つの関数 $f$ と $g$ の漸近的な振る舞いを調べる．以下に，代表的な漸近的記法を紹介する．

#### (1) ビッグ・オー記法

正の数 $c$ と $m$ が存在し，任意の $n \geq m$ について $|f(n)| \leq c|g(n)|$ であるとき，「$f$ の成長率

は，$g$ の成長率により上に有界である」という．これを $f(n) = O(g(n))$ と記し，「$f(n)$ は $g(n)$ の**ビッグ・オー（big oh）**である」という．ビッグ・オーには次の性質がある．

- $f(n) = O(f(n))$ である．
- $f(n) = O(g(n))$ かつ $g(n) = O(h(n))$ であれば，$f(n) = O(h(n))$ である．
- すべての $n \geq m$ について $0 \leq f(n) \leq g(n)$ であれば，$f(n) = O(g(n))$ である．
- $f(n) = O(g(n))$ かつ $a$ が任意の実数であれば，$af(n) = O(g(n))$ である．
- $f_1(n) = O(g(n))$ かつ $f_2(n) = O(g(n))$ であれば，$f_1(n) + f_2(n) = O(g(n))$ である．
- $f_1$ と $f_2$ が非負な値をもち，そして $f_1(n) = O(g_1(n))$ かつ $f_2(n) = O(g_2(n))$ であれば，$f_1(n) + f_2(n) = O(g_1(n)) + g_2(n))$ である．

ビッグ・オーは，同じ問題を解く異なるアルゴリズムについて議論する際の大まかな尺度として使用できる．例えば問題 $P$ を解くアルゴリズムがあるとしよう．このアルゴリズムの最悪の場合に実行時間は，次数 $m$ の多項式で表せる．そのとき $P$ を解く最悪の場合の最適なアルゴリズムがあれば，それは，最悪の場合で $O(n^m)$ の実行時間をもつはずだといえる．つまり，$P$ を解く最適なアルゴリズムの最悪の場合の実行時間の成長率は，$n^m$ で上に有界でなければならない．

**（2）ビッグ・オメガ記法**

次に，ビッグ・オーの鏡像について議論する．正の数 $c$ と $m$ が存在して，任意の $n \geq m$ について $|f(n)| \geq c|g(n)|$ であるとき，「$f$ の成長率が $g$ の成長率により下に有界である」という．これを $f(n) = \Omega(g(n))$ と記し，「$f(n)$ は $g(n)$ の**ビッグ・オメガ（big omega）**である」という．ビッグ・オメガの性質は，ビッグ・オーの性質と類似している．

ビッグ・オーほどではないものの，ビッグ・オメガも最適なアルゴリズムのおおよその下界を記述するのに使用できる．例えばある特定の問題を解くすべてのアルゴリズムの下界が，次数 $m$ の多項式で与えられていることが分かっているとき，$P$ を解く最悪の場合の最適なアルゴリズムがあるならば，$\Omega(n^m)$ の最悪の場合の実行時間をもつはずであるということができる．つまり，$P$ を解く最適なアルゴリズムは，最悪の場合の実行時間の成長率が $n^m$ で下に有界でなければならない．

**（3）ビッグ・シータ記法**

次に，$f$ と $g$ についてビッグ・オーとビッグ・オメガの両方が成立する場合について考える．数 $m$ と二つの正の整数 $c$，$d$ が存在し，任意の $n \geq m$ について $c|g(n)| \leq |f(n)| \leq d|g(n)|$ であるとき，「関数 $f$ が $g$ と同じ成長率をもつ」，ないし「$f$ が $g$ と**同じオーダー（same order）**である」という．これを $f(n) = \Theta(g(n))$ と記し，「$f(n)$ は $g(n)$ の**ビッグ・シータ（big theta）**である」という．

$f(n) = \Theta(g(n))$ であり，かつすべての $n \geq m$ について $g(n) \neq 0$ であれば，定義中の不等式を $g(n)$ で除算することで，以下が得られる．

$$\text{すべての } n \geq m \text{ について } c \leq \left|\frac{f(n)}{g(n)}\right| \leq d$$

ビッグ・シータについては，次の性質がすべての関数で成立する．

- $f(n) = \Theta(f(n))$ である．
- $f(n) = \Theta(g(n))$ であれば， $g(n) = \Theta(f(n))$ である．
- $f(n) = \Theta(g(n))$ かつ $g(n) = \Theta(h(n))$ であれば，$f(n) = \Theta(h(n))$ である．

ビッグ・シータの定義から，$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ であれば $f(n)=\Theta(g(n))$ であるといえる（ただし $c \neq 0$ かつ $c \neq \infty$）．例えば，$p(n)$ が次数 $m$ の多項式であれば，$p(n) = \Theta(n^m)$ である．なぜなら，$p(n) = a_0 + a_1 n + \cdots + a_m n^m$ として，次の極限を得るからである．

$$\begin{aligned}\lim_{n\to\infty} \frac{p(n)}{n^m} &= \lim_{n\to\infty} \frac{a_0 + a_1 n + a_2 n^2 + \cdots + a_{m-1} n^{m-1} + a_m n^m}{n^m} \\ &= \lim_{n\to\infty} \left( \frac{a_0}{n^m} + \frac{a_1}{n^{m-1}} + \frac{a_2}{n^{m-2}} + \cdots + \frac{a_{m-1}}{n} + \frac{a_m}{1} \right) = a_m\end{aligned}$$

ここで $p(n)$ は次数 $m$ なので，$a_m \neq 0$ であるから，$p(n) = \Theta(n^m)$ となる．

アルゴリズムの効率の近似を議論するために，どのようにビッグ・シータが用いられるかの例を見てみる．

例えば二分探索アルゴリズムの最悪の場合の効率は $\Theta(\log n)$ である．なぜなら実際の値は $1 + \lfloor \log_2 n \rfloor$ だからである．線型探索の平均と最悪の効率は，ともに $\Theta(n)$ である．なぜなら平均の比較回数は $(n + 1)/2$ であり，最悪の場合の比較回数は $n$ だからである．

整列アルゴリズムの最悪の場合の下界は，$\lceil \log_2 n! \rceil = \Theta(n \log n)$ である．バブルソートのような単純な整列アルゴリズムの最悪の実行効率は $\Theta(n^2)$ である．与えられたリストの順列を構成して，整列しているかどうかを検査するアルゴリズムは，正しく整列されたものを得る前にすべての可能な順列を構成しなければならないので， $\Theta(n!)$ である．“ヒープソート” と呼ばれる整列アルゴリズムは，$n$ 個の任意のリストを最大 $2n \log_2 n$ の比較回数で整列することができる．つまりヒープソートは，最悪の場合で $\Theta(n \log n)$ である．

### (4) リトル・オー記法

前項の極限が 0 の場合を考える．

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

または同等の定義を極限を使わずに述べるなら，任意の $\epsilon > 0$ についてある $m$ が存在し，$n \geq m$ のとき $|f(n)| \leq \epsilon$ であるなら,「$f$ は，$g$ よりも低い成長率である」という．これを $f(n) = o(g(n))$ と書き,「$f$ は $g$ の**リトル・オー（little oh）**である」という．

成長率の大小を $\prec$ で表すことがある．この場合，$f(n) \prec g(n)$ は，$f(n) = o(g(n))$ を意味する．代表的な関数をこれにより並べたものを次に示す．

$$1 \prec \log n \prec n \prec n \log n \prec n^2 \prec n^3 \prec 2^n \prec 3^n \prec n! \prec n^n$$

## 1-3-4 時間計算量と $P$，$NP$

**判定問題（decision problem）**とは，入力に対して Yes か No かで回答できるような問題

をいう．アルゴリズムが判定問題を**解く（solve）**とは，問題に対してアルゴリズムが停止し，Yes か No で正しく解答することをいう．

ここで，判定問題を解く決定性アルゴリズムが必要とする時間に注目し，その時間計算量にしたがって，判定問題をクラス $P$ とクラス $NP$ に分類する．クラス $P$ は，最悪の場合でも多項式に比例する実行時間で，決定性アルゴリズムが解くことができる判定問題から構成される．つまり判定問題は，その問題を解く決定性アルゴリズム $A$ が存在し，その問題について $W_A(n) \leq p(n)$ となる多項式 $p$ が存在するとき，クラス $P$ に属する（ここで $n$ は，問題の大きさである）．

ある問題が $P$ に属するならば，それは**扱いやすい（tractable）**といい，$P$ に属さなければ，**扱いにくい（intractable）**という．換言すれば，ある問題の最悪の場合の計算量の下界がいかなる多項式よりも大きければ，その問題は扱いにくいといわれる．

クラス $NP$ とは，解を多項式時間で検査できるすべての問題から構成される．換言すれば $NP$ の問題は，解を非決定的に探索するアルゴリズムをもつ．ここで判定問題の非決定性アルゴリズムは，次の二つの段階をもつ．

1. ある問題 $I$ について，解 $S$ を推測する．
2. 推測段階で得られた推測 $S$ がその問題 $I$ の解であるかどうか検査する決定性アルゴリズムを開始する．

この検査アルゴリズムは，$S$ が $I$ の解であり，かつそのときに限り Yes と回答して停止する．しかし $S$ が $I$ の解でなければ，停止するかもしれないが，停止しないかもしれない．可能な解 $S$ の推測は，天からの啓示としてまったく何もないところから得る．$S$ は無限長かもしれないので，推測段階が停止しないこともあり得る．そして検査段階は，$S$ を考慮しなくてもよい．

ある判定問題 $I$ に対して，検査段階で Yes を出力して停止するような解 $S$ が存在して，検査段階に必要な時間が $p(n)$ 以下になるような多項式 $p$ が存在するとき，「非決定性アルゴリズムが多項式時間で判定問題を解く」という（ここで $n$ は，$I$ の大きさである）．つまりクラス $NP$ は，非決定性アルゴリズムによって多項式時間で解くことのできる判定問題から構成される．

$P$ と $NP$ の関係を考えてみる．$P$ は $NP$ の部分集合なので，次の関係が成立する．

$$P \subseteq NP$$

したがって $P$ に属するすべての問題を含む多くの問題が，$NP$ に含まれることになる．$P$ と $NP$ が等しいかどうかは知られていない．つまり，$P$ に含まれない $NP$ 問題を誰も発見していない（$P$ に属さない $NP$ 問題があれば，$P \neq NP$ を証明できる）．一方で，誰もすべての $NP$ 問題が $P$ に含まれるとも証明していない．すべての $NP$ 問題が $P$ に属するのであれば，$P = NP$ が証明できる．この問題は，計算機科学の最前線にある未解決問題の一つである（クラス P と NP → 6 群 2 編 5 章）．

### 1-3-5 領域計算量と *PSPACE*

次に問題を解くアルゴリズムが必要とする記憶領域に注目しよう．クラス *PSPACE* とは，

問題実体の長さの多項式より大きなメモリセルを使用せずに決定性アルゴリズムが解くことのできる判定問題の集合である．つまり，問題を解く決定性アルゴリズムが存在して，そのアルゴリズムが $p(n)$ より大きな記憶領域を使用しないような多項式 $p$ が存在するとき，その問題は $PSPACE$ に属する．ここで $n$ は，問題の大きさである．

クラス $NP$ と $PSPACE$ の関係を考える．$NP$ 中の任意の問題 $\pi$ について，たかだか $p(n)$ ステップで検査する非決定性アルゴリズム $A$ と多項式 $p$ が存在する．ここで $n$ は，問題の大きさである．$A$ の任意のステップは，たかだか固定の数 $k$ 個のメモリセルにアクセスするにすぎない．したがって $I$ の解を検査するのに，たかだか $kp(n)$ 個のメモリセルを使用する．$p$ は多項式なので，$kp$ も多項式である．したがって検査段階は，多項式領域を使用する．$S$ が問題 $I$ の解であれば，検査段階で使用する $S$ の部分は，$kp(n)$ 個のメモリセルに収まる．したがって $S$ は，記号の有限なアルファベット上の，長さはたかだか $kp(n)$ の文字列と仮定することができる．

$\pi$ を解く決定性アルゴリズム $B$ を定義することにしよう．長さ $n$ の実体について $B$ は，長さがたかだか $kp(n)$ であるすべての可能な文字列を一度に一つずつ生成しては検査する．検査段階は，$A$ の検査段階を変更して，$p(n)$ ステップの後に，それまでに停止していなければ停止するようにして行うことができる．解が見つかれば，$B$ は Yes を出力して停止する．そうでなければ，長さがたかだか $kp(n)$ のすべての可能な文字列を生成して検査した後に，No を出力して停止する．

生成段階は多項式領域を使用する．なぜなら長さがたかだか $kp(n)$ の文字列を生成するからである．検査段階も多項式領域を使用する．なぜならそれは，制限時間を設ける変更を加えた $A$ の検査段階だからである．有限のアルファベットと局所変数は，一定の領域を使用する．つまり $B$ も多項式領域を使用する．したがって $NP \subseteq PSPACE$ である．このことと前の節の結果を併せると，次の関係が得られる．

$$P \subseteq NP \subseteq PSPACE$$

$P$ と $NP$ が等しいか否かが未知であるのと同様, $NP$ と $PSPACE$ が等しいか否かも未知である．

### 1-3-6 NP 完全性と NP 困難性

問題の集合 $A$ の要素を問題 $B$ の要素へと写像する多項式時間計算可能な関数 $f$ が存在するとき，問題 $A$ は問題 $B$ に**多項式時間帰着可能（polynomially reducible）** であるという．

この考え方を用いて $B$ の効率的アルゴリズムから $A$ の効率的アルゴリズムを求める方法を説明する．$B$ を解く多項式時間アルゴリズム $M$ を発見したとする．$M$ と $f$ を用いれば，問題 $A$ の任意の要素 $I$ について，これを多項式時間で $f$ により問題 $B$ へと写像し，続いて $M$ により多項式時間で解くことができる．

多項式帰着可能性は，クラス $NP$ の議論に深くかかわっている．$NP$ の判定問題のうちで，$NP$ のほかのすべての判定問題が多項式時間でその問題に変換することができるとき，その判定問題を **NP 完全（NP-complete）**であるという．

例えばある $NP$ 問題 $B$ があり，$NP$ のほかのすべての問題が $B$ に多項式時間で帰着できることが判明したとする．そのとき $B$ を解く決定性多項式時間アルゴリズムを求めさえしたな

らば，*NP* のすべての問題が決定性多項式時間アルゴリズムをもつことになる．これにより，多くのよく知られた問題を解く効率的な解法を得るのみならず，*NP* と *P* が等しいことも示せたことになる．

*NP* 完全問題の最初の例は，Cook[1] によって与えられた．それは，連言標準形になっている命題整論理式が充足可能かどうかというものである．Cook は，任意の *NP* 完全問題が CNF 充足可能性問題へと多項式時間で帰着可能であることを示すことで， CNF 充足可能性問題が *NP* 完全であることを証明した．

いったん *NP* 完全問題（例えば CNF 充足可能性問題）を得たならば，ほかの問題 *A* が *NP* 完全であることを示すには， *A* が *NP* であることを示したうえで，既知の *NP* 完全問題（例えば CNF 充足可能性問題）を，*A* に多項式時間で帰着できることを示せばよい．つまり任意の *NP* 問題を， *A* に多項式時間で帰着可能である既知の *NP* 完全問題へと多項式時間で帰着することで *A* に帰着することができる．つまり *A* が *NP* 問題であり，*B* が *A* に多項式時間帰着可能な *NP* 完全問題であれば，*A* は *NP* 完全である．

NP 完全と類似した概念に **NP 困難（NP-hard）**がある．ある問題 *B* があり，任意の *NP* 問題 *A* を多項式時間で *B* に帰着できるならば，*B* は *NP* 困難な問題である，という．*B* は必ずしも *NP* であるとは限らないので，*NP* 完全問題の集合は *NP* 困難問題の集合に含まれるといえる．

### 1-3-7 償却解析

ここまでは計算量解析において，アルゴリズムの 1 回の実行に要する時間や領域に着目してきた．しかし実用に供されているシステムやアルゴリズムにおいては，1 回の実行に要する最悪の時間は劣っているとしても，繰り返し呼び出された場合の平均について言えば好ましい性能を示すようなものが存在する．**償却解析（amortized analysis）**とは，このような多数回の実行を通じたアルゴリズムの振る舞いを解析することをいい，これにより求められる計算量を**償却計算量（amortized complexity）**と呼ぶ[3]．

現実によくある例として，ベクトルのような複数の値の並びを保管する領域の管理を考える（図 1･8）．最初にまとまった大きさの領域を割り当てておけば，値の追加があっても空いている部分がある限り，その空き部分に収納できるので，追加操作の時間計算量は $O(1)$ である．しかし，空きがなくなった場合には，1 要素を追加する前に新しい領域を用意し，そこに既存の要素をすべてコピーしてから追加しなければならない．現在格納されている要素の個数が $m$ であるなら，この操作は $O(m)$ を要する．この操作は平均でどれだけの手間を要するだろうか．

最初に 10 要素分の空き領域を用意して，一杯になるごとに新たに 10 要素分大きな領域を用意してコピーするものとして，$n$ 個の要素を順に追加する場合，合計と平均の時間計算量はどうなるだろうか．$\frac{9}{10}$ の場合においては，1 回のコピーで操作は終了するが，残る $\frac{1}{10}$ の場合においては，現在入っている要素をすべてコピーする必要がある．一般性を失わずに $n$ が 10 の倍数であるものと考えることができるので，このとき全体のコピー数は次のようになる．

$$\frac{9}{10} \times n + \sum_{i=1}^{\frac{n}{10}} 10i = \frac{9n}{10} + \frac{\frac{n}{10}(\frac{n}{10} - 1)}{2} = \frac{n^2}{200} + \frac{17n}{20}$$

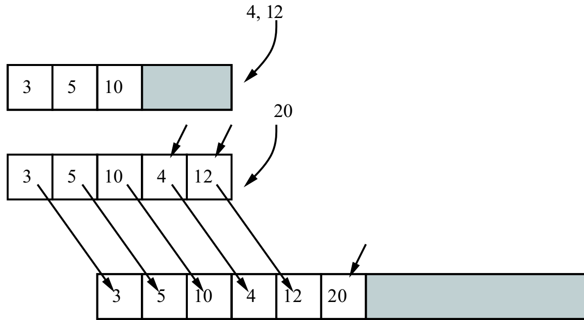


**図 1·8　ベクトルへの値の追加**

これを $n$ で割って，平均で追加 1 回あたりの償却計算量は $O(n)$ となる．ところで，10 ずつ増やす代わりに,「最初は 1 の大きさで，一杯になるごとに領域の大きさを倍にする」方法ではどうだろうか．再び一般性を失わずに $n$ が $2^m$（$m = \log_2 n$）であるものと考えることができるので，このとき全体のコピー数は次のようになる．

$$(n - \log_2 n) + \sum_{i=0}^{\log_2 n-1} 2^i = 2n - 1 - \log_2 n$$

これを $n$ で割ると，今度は平均で追加 1 回当たりの償却計算量は $O(1)$ となる．このように償却解析を行うことで，多数回の操作を通じたコストの小さいアルゴリズムを選択することができる．

**■参考文献**

1) S.A. Cook, “The Complexity of Theorem Proving Procedures,” Proceedings of the Third Annual ACM Symposium on the Theory of Computing, pp.151-158, 1971.

2) James L. Hein 著, 神林　靖 訳, “独習コンピュータ科学基礎 I 離散構造,” 翔泳社, 2011.

3) Clifford A. Shaffer 著, 久野禎子・久野　靖 訳, “Java によるデータ構造とアルゴリズム解析入門,” ピアソンエデュケーション, 2000.