# 楽して計算するには　－アルゴリズムの設計と解析

牧野　和久
京都大学　数理解析研究所
makino@kurims.kyoto-u.ac.jp

**概 要**

近年の情報化社会において，高速なアルゴリズムを設計することは極めて重要である．しかしながら，P vs NP 問題に代表されるように，与えられた問題が効率的に解けるか，あるいは，解けないか容易には分からない現状にある．本講義では，計算可能性, P，NP などの計算量理論の基礎的な概念を説明すると同時に，高速アルゴリズム設計の意義や重要性を応用などを交えて議論する．その後，分割統治法や動的計画法などの高速なアルゴリズム設計のための手法およびその解析法を具体的な問題を用いて紹介する．それ以外にも，ＮＰ困難な問題に対する最適化の手法を用いた近似アルゴリズムの設計法も議論する．

## 1　序論

本講義では,「楽して」計算する方法について議論する.「楽をして」というと，ややもすれば「怠けて」,「手を抜いて」,「いいかげんに」と否定的な印象をもちがちであるが，そうではなく,「無駄を省き」,「効率的に」,「高速に」という意味で用いる．本稿では，その意味での計算法，アルゴリズム設計法，あるいは，その逆に計算の限界について議論する．まずはじめに，簡単な例をいくつか挙げてその意義や重要性を考えよう．

**例 1．** $1+2+3+\cdots+1000$ を計算せよ．

みなさんは当然公式を用いて，$\{(1+1000)\times 1000)\}\div 2$ と計算するかと思う．すなわち，足し算，掛け算，割り算をそれぞれ一回づつ，計 3 回の演算を用いることで計算可能である．しかしながら，もし公式を用いないとすると（私にはそのような根性はないが），まず初めに第一項と第二項について $1+2=3$ と計算する．次に，計算結果の 3 に第三項の 3 を加えて $3+3=6$ を得る．さらにこの 6 と第四項 4 を足し合わせて $6+4=10$ を得る．このように順に計算すると，

$$\underbrace{\underbrace{1+2}_{3}+3}_{6}+4+\cdots+1000$$
$$\underbrace{\qquad\qquad}_{10}$$
$$\vdots$$

999 回足し算を行うことで解を得る．このように計算の方法が異なると，演算数も 3 と 999 と大きな差が出ることが分かる．もちろん掛け算や割り算は，足し算に比べて能力を必要とする．しかし例えば，現在のコンピュータにおいては，その差はほとんどないと言って過言ではない．今，例題 1 の 1000 を $n$ とすると，公式を用いた方法では，どんな $n$ に対しても定数回（正確には, 3 回）の演算で計算可能であるが，後者の単純な方法では，線形回（正確には，$n-1$ 回) 必要なことがわかる．このように「定数」と「線形」という大きな違いがあることが分かる．もう一つ例題をみてみよう．

**例 2.** 正整数 $a$ と $n$ に対して，$a^n$ を計算せよ．

もちろん関数電卓などが手元にあれば，すなわち,「べき乗」を演算として許せば，1 回の演算で計算可能であるが，ここでは，四則演算のみが可能であるとする．このとき，例題 1 と同様に，単純な方法

$$\underbrace{\underbrace{\underbrace{\underbrace{a \times a}_{a^2} \times a}_{a^3} \times a}_{a^4} \times \cdots \times a}$$

を用いると，$n-1$ 回の掛け算を用いることで求められる．では，この例題 2 に対しても，例題 1 のようにもっと速く計算する方法はないのであろうか？ 残念なことに公式はないのであるが，上記の単純な方法は無駄があることが分かる．それは，例えば，はじめに折角 $a^2$ を計算したのに，それを一度しか利用していない点である．すなわち，はじめに $a \times a = a^2$ を計算したら，その $a^2$ を用いて，$a^2 \times a^2 = a^4$ を計算する．次に，得られた $a^4$ を用いて，$a^4 \times a^4 = a^8$ を計算する．このように順次計算すると，

$$\begin{aligned}
&(1) \quad a \times a = a^2 \\
&(2) \quad a^2 \times a^2 = a^4 \\
&(3) \quad a^4 \times a^4 = a^8 \\
&\qquad \cdots \\
&(k) \quad a^{2^{k-1}} \times a^{2^{k-1}} = a^{2^k}
\end{aligned}$$

となり，$n = 2^k$ のときは，$k = \log n$ 回の演算で $a^n$ を求めることができる．$n \neq 2^k$ のときは，$n$ の 2 進表現を利用することで，$\log n$ に比例する時間で計算可能である．この例でも，例えば，$n = 2^{10}$ のとき，上記の 2 つの方法では，演算数（すなわち，計算時間）に関して $n-1 = 1023$ 回 $\gg \log n = 10$ 回と大きな差があることが分かる．

上記の例からわかるように，簡単な問題であっても，計算の仕方，すなわち，アルゴリズムが如何に大切であるか分かる．現在の情報化社会においては，宅配便の配送計画，タンパク質の構造解析，大規模集積回路の設計，工場での生産スケジューリングなど様々な分野で日々問題が解かれており，それらに対する効率的なアルゴリズムの開発は極めて重要な課題である．

このような効率的なアルゴリズム設計とは逆に，暗号の世界では,「どう頑張っても解けない」ということが重要になる．たとえば，自転車などに用いられる 4 桁の数字からなるダイヤルロック錠

を考えてみましょう．もし，ロック錠の暗証番号を知っているならば，各桁で（上下どちらかの方向に）高々5回，全体で $5 \times 4$ 回廻せば開錠できる．しかし，暗証番号を知らないとすると，最悪 0000 から 9999 まで一万回廻らなければ，開錠することができない．もちろん運がよければ1回で開けることができるが，期待値としても五千回必要になる．この必要計算量の差が暗号としての根本原理になる．この差は，桁数を $n$ とすると，暗証番号が既知な場合は，$5n$ 回であり，未知な場合は $10^n$ となり，指数ギャップであり，$n$ の増加とともに巨大になる．

もう一つの例として，1977 年に Rivest, Shamir, Adleman により発明された RSA 暗号がある.[1] この暗号は現在インターネットで広く使われている実用的な公開鍵暗号である．この暗号は，実は，素因数分解が高速に解ければ，解読されてしまう．例えば，323 の素因数分解を考えてみよう．みなさんすぐに解を見つけられただろうか？ 少なくとも私は一瞬素因数分解ができるかどうか戸惑ってしまう．答えは $323 = 17 \times 19$ である．では，

$$
\begin{aligned}
&1230186684530117755130494958384962720772853569595334792197\\
&3224521517264005072636575187452021997864693899564749427740\\
&6384592519255732630345373154826850791702612214291346167042\\
&9214311602221240479274737794080665351419597459856902143413
\end{aligned}
$$

は素因数分解できるだろうか？ ここで, 上記は 4 つの整数があるのではなく，一段目の左端から右端，二段目の左端から右端，三段目の左端から右端，四段目の左端から右端と続く一つの整数である．これは，RSA-768 とよばれる 2 進で７６８桁（10 進で２３２桁）の整数であり，この素因数分解は，RSA 社が RSA 暗号の難しさ，解読のためのアルゴリズム考察，また，最新の計算機環境でどの程度の桁数の整数が素因数分解可能であるかなどを調べるために 2007 年まで行われた RSA Factoring Challenge の中で出題された [7]．この RSA-768 は，2009 年に, 数百台のパソコンによる並列計算処理で約 3 年かけて,

$$
\begin{aligned}
&33478071698956898786044169848212690817704794983713768568 91\\
&24313889828837938780022876147116525317430877378144679994 89\\
\times\ &36746043666799590428244633799627952632279158164343087642 67\\
&60322838157396665112792333734171433968102700927987363089 17
\end{aligned}
$$

と素因数分解された．これらのことからわかるように，素因数分解に対する高速なアルゴリズムは現在知られておらず，計算量的には難しい問題，より正確には，桁数の多項式時間で素因数分解するアルゴリズムは存在しないと信じている研究者が多い．この素因数分解は，それ自身の計算は難しいが，その逆に，与えられた 2 つの整数の積を求めることは簡単である．公開鍵暗号系は，このような一方向性という性質に基づいて考案された安全な暗号系である．

$$
17 \times 19 \underset{\text{困難}}{\overset{\text{容易}}{\rightleftharpoons}} 323
$$

[1] RSA 暗号に関連する功績によって 2002 年にチューリング賞を受賞した．歴史的には，RSA 暗号を最初に発見したのは、実は英国政府通信本部 (GCHQ) に勤めていた Cock であるが，英国政府が極秘扱いとされ，1997 年に初めて世間に公表された.

このように効率的に解けること，あるいは，その逆に解けないことを示す計算量解析は非常に重要であるが，P vs. NP などに代表されるように，未解決な問題は多く残されている現状にある．

本稿の構成は以下の通りである．まず，第 2 節で，計算の可能性の概念を説明し，計算不可能問題が存在することを示す。次に，時間，領域という 2 つの計算量の概念を与えると同時に，階層定理を示す．第 3 節では，効率的なアルゴリズム技法である分割統治法と動的計画法を例題を用いて説明する．最後に，第 4 節では，計算クラス NP，NP 困難性，完全性などの定義を与えると同時に，代表的な NP 完全問題として代表的な充足可能性問題を紹介する．また，制度保証付き近似アルゴリズムについても議論する．

# 2　計算可能性と計算量

## 2.1　計算可能性

本節では，序論で説明した内容をすこし定式化して議論する．そもそも「計算」とは何だろうか? 現代社会ではコンピュータは身近な存在であり，自然にその概念をイメージすることができるかもしれない．しかし，これらの議論が盛んであった 1930 年代頃には容易ではなかったのかも知れない．「計算」を議論するためには，まず「計算機」，および，その下での「計算可能性」を定義する必要がある．1930 年代頃，Church, Godel, Kleene, Post, Markov, Turing らにより，有限状態機械，プッシュダウン・オートマトン，チューリング機械，ランダムアクセル機械 (RAM)，$\lambda$ - 計算，ポストシステム，帰納的関数など様々な妥当な計算 (機) モデルとそれに基づく計算可能性が提案された．例えば，チューリング機械は，計算量理論において中心的な役割を果たしている．また，ランダムアクセル機械は，現在の計算機を抽象化した仮想的な計算機である．現在のコンピュータから抽象化された主なポイントは，記憶領域が無限個あり，そのどこにも単位時間でアクセスできる点である．これら以外は，すべてみんなのイメージ通りだと思ってよい．特に，ランダムアクセル機械で許される命令は，四則演算，数字の大小比較，if 文，goto 文に相当するものがあり，みなさんが知っているプログラム言語が表現できると仮定してよい．ただ，細かな点は各計算モデルにはたくさんの変種があり，本稿では扱わない．詳しくは最後に載せた文献を参考にされたい．

さて，この計算モデルを用いて問題を解くことを考えるのであるが,「問題」とは何だろうか? 計算機科学分野においては，入力と出力の対の二項関係を問題とよぶ．すなわち，入力を決めたとき，その出力がはっきりするものを問題という．ここで，入力を決めたものを問題例とよぶ．通常，問題は，無限個の問題例からなる．すこし直観的でないので，例を用いると，**例 2** は入力が正整数 $a$ と $n$，出力が $a^n$ である問題であり，$a = 2$，$b = 3$ と入力を決めたものを問題例と呼ぶ．また，**例 1** は入力が正整数 $n$，出力が $1 + 2 + \cdots + n$ である問題の $n = 1000$ という問題例である．計算モデル $M$ で許された命令からなるアルゴリズムを用いて，問題 $A$ が ($M$ で) 有限ステップで解けるとき，問題 $A$ が計算モデル $M$ の下で計算可能であるという．すなわち，どんな入力に対しも有限ステップ（時間で）正確に答えを出すアルゴリズムが存在するときに，計算可能であるといわれる．したがって計算モデル毎に計算可能性が定義できることになる．たとえば，有限状態機械で計算できる問題は，プッシュダウン・オートマトンでも計算可能であり，さらに，プッシュダウン・オートマトンで計算できる問題は，チューリング機械でも計算可能であること，すなわち，プッシュダウン・オートマト

ンの能力は有限状態機械の能力より高く，チューリング機械の能力はプッシュダウン・オートマトンの能力より高いことが知られている．1930 年代当初は,「妥当な」モデルの中で最も計算能力の高いものは何か盛んに議論された．ここで「妥当な」とは，もちろん数学として定義しなくてはならないため，抽象化はしなくてはならないのであるが，その抽象化した点以外では，現実のコンピュータに即したものでなくてはならないということである．したがって,「妥当な」とは数学的ではない．1930 年代，同時期に，妥当かつ高能力のモデルが様々提案されたが，それらの能力は実は等価であることが判明した．上記の中でいうと，有限状態機械，プッシュダウン・オートマトンを除いたすべて計算モデルの能力が等しい．これにより，このような計算のモデル，たとえば，チューリング機械で計算できるものを単に,「計算可能」と呼ぶようになった．これは，Turing-Church の提唱とよばれている．ただ，上記に述べたように，何を以て妥当とするかは，数学的でなく，単なる定義だとみなしてよい．本稿の残りでは，先ほども述べたように，計算モデルを意識せずに，みなさんが普段用いているコンピュータとプログラム言語をイメージして読んでいただきたい．

このように計算可能性を定義したのであるが，多くのひとは，どんな問題でも十分な時間をかけさえすれば解ける，すなわち，コンピュータは無限の計算能力があると思っていないだろうか？ 残念なことにこの直観は正しくない．その代表的な問題はチューリング機械の停止性問題である．この問題はチューリング機械モデルの下で，入力 $x$ に対してアルゴリズム $A$ を実行したときに,（有限時間で）停止するかどうかを問う問題であり，入力は $A$ と $x$，出力は Yes (停止する)，No (停止しない) の 2 種類である．ここで，アルゴリズム $A$ 自身も入力の一部の文字列であるとみなしていることに注意されたい．

**定理 2.1** チューリング機械の停止性問題は計算不可能である．

さてチューリング機械の定義をしていないので，正確には上記の定理は証明できないのであるが，みなさんが普段用いているコンピュータとプログラム言語をイメージして証明もどきをしてみよう．いま，チューリング機械の停止性問題が計算可能であると仮定して，矛盾を導く．チューリング機械の停止性問題を有限時間で解くアルゴリズム $H$ が存在すると仮定する．ここで，入力 $x$ に対してアルゴリズム $A$ を実行し，停止するとき，$H(A,x) = \text{Yes}$，そうでないとき，$H(A,x) = \text{No}$ と定義する．このアルゴリズム $H$ を用いて，以下のアルゴリズム $M$ を考えよう．

入力 $x$ に対して，$H(x,x) = \text{Yes}$ ならば停止せず，$H(x,x) = \text{No}$ ならば停止する

チューリング機械においては，無限ループも実現できるので，アルゴリズム $H$ からアルゴリズム $M$ を構成可能である．このとき，$M(M)$ が停止するかどうかを考えよう．もし，$M(M)$ が停止するならば，$M$ の定義より $H(M,M) = \text{No}$ となる．$H$ の定義より，$M(M)$ が停止しないことになり，矛盾を得る．一方，$M(M)$ が停止しないならば，$M$ の定義より $H(M,M) = \text{Yes}$ となる．$H$ の定義より，$M(M)$ が停止することを意味し，矛盾する．よって停止性問題を有限時間で解くアルゴリズム $H$ は存在しない．

この定理は，停止するかどうかを，必ず停止するアルゴリズムを使って判定することができない，ということを主張している．また，上記の証明は本質的に，対角線論法を用いたものとみなすことができる．

さて，この停止性問題はすこし人工的で，自然でないという印象をもつかもしれない．では次の問題を考えよう．

**ポストの対応問題**

**入力**：0-1 文字列集合 $A = \{a_1, a_2, \ldots, a_k\}$ と $B = \{b_1, b_2, \ldots, b_k\}$.

**出力**：$a_{i_1}a_{i_2}\ldots a_{i_m} = b_{i_1}b_{i_2}\ldots b_{i_m}$ となる添え字列 $i_1, i_2, \ldots, i_m$ が存在すれば，Yes.
そうでなければ，No.

たとえば，$A = \{a_1 = 1, a_2 = 10111, a_3 = 10\}$，$B = \{b_1 = 111, b_2 = 10, b_3 = 0\}$ である問題例を考えよう．この問題例において，

$$a_2a_1a_1a_3 = 10111|1|1|10$$
$$b_2b_1b_1b_3 = 10|111|111|0$$

となり，ともに文字列 101111110 を表す．ここで，分かり易くするために | で仕切りをいれていることに注意されたい．したがって，$i_1 = 2, i_2 = i_3 = 1, i_4 = 3$ を得るので，この問題例の出力は Yes である．ポストの対応問題は，一見すると簡単に解けそうに思えるのであるが，以下の結果が知られている．

**定理 2.2** ポストの対応問題は計算不可能である．

本稿ではこの証明は行わない．しかし直観的になぜこの問題が難しいのかを説明する．みなさんならばこの問題をどのように解くだろうか？ 最も自然な解法（アルゴリズム）は，$a_{i_1}a_{i_2}\ldots a_{i_m} = b_{i_1}b_{i_2}\ldots b_{i_m}$ となる文字列の長さ $m$ について順次確認する方法である．

まずは，$m = 1$ のときを調べる： $a_1 = b_1$? $a_2 = b_2$? ... $a_k = b_k$? もしどこかの $i$ で $a_i = b_i$ となれば，Yes を出力する．そうでなければ，$m = 2$ のときを調べる： $a_1a_2 = b_1b_2$? $a_1a_3 = b_1b_3$? ... $a_{k-1}a_k = b_{k-1}b_k$? もしどこかの $i, j$ で $a_ia_j = b_ib_j$ となれば，Yes を出力する．さもなければ，$m = 3$ のときを考える，というように順次 $m$ を大きくすることにより問題を解く

このアルゴリズムは一見すると正しいように思え，それゆえポストの対応問題は計算可能であると主張したくなるかもしれない．しかし上記では，いつ No と出力するかまったく述べていない．もちろん No である問題例はいくつもあり，いつかは No と結論付けなければならないのであるが，どの段階で No だと言い切れるのであろうか．$m = 1$ 億 まで反復すれば No と出力してよいだろうか．そうではない．ひょっとして 1 億 1 回目で文字列が発見できるかもしれない．では，1 京回反復すればいいのだろうか．ポストの対応問題は，このように何回反復しても No だと結論付けられず，それゆえ計算不可能になる．

このように計算不可能な問題は存在し，計算機は万能でないことが分かった．では計算可能であればよいのであろうか？ たとえ計算可能，すなわち，有限時間で正しい答えを必ず出力したとしても，その有限が莫大であれば，例えば，1 億年かければ問題が解けると言われても，このことは解けないことと同等である．次節ではこのようなことを議論するため計算量の概念を導入し，計算可能な問題クラスを細分化することを行う．

## 2.2 計算量

序論の例題にあるように，問題を解くアルゴリズムは一意でなく複数存在する．では，どんなアルゴリズムが望まれるのであろうか？ 設計者あるいはユーザなどさまざまな立場があると思われる

が，当然高速で省メモリなアルゴリズムが望まれる．計算量理論分野では，この２つの対応する時間量と領域量を合わせて，計算量とよび，それらの解析が中心的な研究課題となっている．時間量は，アルゴリズムを実行したときに必要となる実行時間に対応し，領域量は，アルゴリズム実行時に使用するメモリ数に対応する．もちろん，アルゴリズムの良さはそれだけでなく，たとえば，見易さ，デバッグのし易さなどもアルゴリズム設計の観点からは重要である．時間量と領域量を吟味する際，入力のサイズを基準に議論することに注意しよう．たとえば，皆さんが職場で部下の勤務評価を行う場面を考えてみよう．部下の人数が十人の場合と百人の場合では，当然百人の勤務評価を作成する方が時間を要することは明らかである．すなわち，この場合，入力のサイズは $n = 10$ と $n = 100$ であり，この $n$ に対して，計算時間やメモリ数がどのように変化するかを議論する．たとえば，線形時間アルゴリズムとは，入力サイズ $n$ であるどんな問題例に対しても，$n$ に比例する基本演算を行うことでアルゴリズムが正しい答えを出力し停止することを意味する．また，計算科学分野では，入力 $n$ に対して線形，あるいは二乗が上界となるとき，$O(n)$，$O(n^2)$ などとオーダ表記を用いる．

講演で説明するが, 多項式 $n^k$ ($k$ : 正定数) と指数 (たとえば，$2^n$) では $n$ の増加に伴う挙動に大きな違いがある．たとえば，$2^{100}$ は巨大な数であり，みなさんがお持ちのコンピュータで，$2^{100}$ 回の演算を行おうとすると $10^{12}$ 年以上必要となる．このようなことから計算量理論の分野では，$n$ の多項式を効率的，逆に，そうでないもの，例えば，指数などを，手に負えない，あるいは，効率的でない，とよび，多項式時間アルゴリズムの設計，あるいはその逆に，不可能性などを議論している．

以下の節では，時間量に限って議論をすすめる．

# 3　高速なアルゴリズム設計法

本節では，効率的なアルゴリズム設計法として代表的な分割統治法と動的計画法を紹介する．

## 3.1　分割統治法

2 つの $n \times n$ 行列 $A$ と $B$ が与えられたとき，その積 $C = AB$ を求める行列積を例にとり，分割統治法を説明する．定義より，積の第 $(i, j)$ 成分

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

であり，$n$ 回の掛け算と $n-1$ 回の足し算，すなわち，$O(n)$ 回の演算で計算できる．従って，$n^2$ 個の要素をすべて計算するには，$O(n) \times n^2 = O(n^3)$ 時間が必要となる．この定義に従った方法より高速に行列積が求められるのであろうか？　下記のような分割統治法を用いると簡単に，$O(n^c)$ 時間 $(c < 3)$ で解けることが分かる．説明を簡潔に行うために，$n = 2^k$ ($k$ は正整数) と仮定する（それ以外の場合も同様に示すことができる).

行列 $A$，$B$ とその積 $C$ をそれぞれ 4 つの $n/2 \times n/2$ 行列に以下のように分解する．

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\ B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},\ C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

定義より

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned} \tag{1}$$

である．これらの $C_{ij}$ を以下に示す 7 個の $n/2 \times n/2$ 行列 $D_1, \ldots, D_7$ を用いて表現する．

$$\begin{aligned} D_1 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ D_2 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ D_3 &= (A_{11} - A_{21})(B_{11} + B_{12}) \\ D_4 &= (A_{11} + A_{12})B_{22} \\ D_5 &= A_{11}(B_{12} - B_{22}) \\ D_6 &= A_{22}(B_{21} - B_{11}) \\ D_7 &= (A_{21} + A_{22})B_{11} \end{aligned} \tag{2}$$

$$\begin{aligned} C_{11} &= D_1 + D_2 - D_4 + D_6 \\ C_{12} &= D_4 + D_5 \\ C_{21} &= D_6 + D_7 \\ C_{22} &= D_2 - D_3 + D_5 - D_7 \end{aligned} \tag{3}$$

以下では，(3) に基づくアルゴリズム，より正確には，

**Strassen アルゴリズム**

**ステップ 1 (分割)**．(2) の右辺にある 14 個の $n/2 \times n/2$ 行列 $A_{12} - A_{22}$，$B_{21} + B_{22}$，$A_{11} + A_{22}$，$B_{11} + B_{22}$，$A_{11} - A_{21}$，$B_{11} + B_{12}$，$A_{11} + A_{12}$，$B_{22}$，$A_{11}$，$B_{12} - B_{22}$，$A_{22}$，$B_{21} - B_{11}$，$A_{21} + A_{22}$，$B_{11}$ を作る．

**ステップ 2 (再帰)**．(2) を用いて，**ステップ 1** で用意した行列から $D_1, \ldots, D_7$ を計算する．

**ステップ 3 (統治)**．(3) を用いて，$D_1, \ldots, D_7$ から $C_{11}, C_{12}, C_{21}, C_{22}$ を計算する．

このアルゴリズムは開発者の名に因んで Strassen アルゴリズムとよばれる．(1)，(2)，(3) から Strassen アルゴリズムが正しく行列積を計算することは容易にわかるかと思う．

分割統治法では，まず前処理として，元問題をいくつかの（子問題とよばれる）問題に分割する．この際，子問題の入力サイズは元問題のサイズより小さくする．Strassen アルゴリズムでは，**ステップ 1** で前処理を行うことで，7 つの $n/2 \times n/2$ 行列の積問題に分解している．

その後，子問題を再帰的に解く．ここで，再帰的とは，上記の例では，子問題として現れた $n/2 \times n/2$ 行列の行列積問題のそれぞれも同様のアルゴリズム，すなわち，7 つの $n/4 \times n/4$ 行列の積問題に分解し，さらに，その $n/4 \times n/4$ 行列の積問題も $n/8 \times n/8$ 行列の積問題へと分解していく方法であ

る，なお，再帰的に解かれる問題のサイズが十分に小さいとき，例えば，行列積の例では，$1 \times 1$ 行列の積問題は素直に解くものとする．

最後に後処理として，再帰計算で得られた子問題の解から元問題の解を構成する．上記のアルゴリズムでは，**ステップ 2** の再帰計算で得られた $D_1, \ldots, D_7$ から $C_{11}, C_{12}, C_{21}, C_{22}$，すなわち，$C = AB$ を計算する．

単純なアイデアであるが，この分割統治法はアルゴリズム高速化に大きく貢献している．Strassen アルゴリズムの時間量を解析してみよう．まず，

$$T(n) : \text{Strassen アルゴリズムを用いて, } n \times n \text{ 行列の積問題を解く際に必要な計算時間}$$

と定義すると，

$$\begin{aligned} T(n) &= 7T(n/2) + c_1 n^2 && (4) \\ T(1) &= c_2 && (5) \end{aligned}$$

となる．ただし，$c_1$，$c_2$ は正定数である．(4) の $7T(n/2)$ は再帰部分（7 つの $n/2 \times n/2$ 行列の積問題の計算時間），$c_1 n^2$ は，前処理と後処理の時間である．また，(5) は，$1 \times 1$ 行列の積問題は単純に解くことを意味する．この再帰式 (4) と (5) より，

$$\begin{aligned} T(n) &= 7^{\ell} T\left(\frac{n}{2^k}\right) + c_1 n^2 \left(1 + \frac{7}{4} + \cdots + \left(\frac{7}{4}\right)^{k-1}\right) \\ &= \left(c_2 + \frac{4}{3} c_1\right) n^{\log_2 7} \end{aligned}$$

を得る．ここで，$n = 2^k$ であることに注意されたい．従って，行列積問題は，Strassen アルゴリズムを用いると $O(n^{\log_2 7})$ 時間で解くことができる．なお，指数部の $\log_2 7 = 2.807\cdots$ となり，単純な $O(n^3)$ 時間アルゴリズムより高速になる．

実用的な観点からは，やはり再帰は時間がかかるので[2]，実際に現れる行列 $A$，$B$ のサイズが小さい場合や疎な（非零要素数が少ない）場合は，遅くなる．もちろん，サイズが大きく，密な行列には適している．したがって，ハイブリッド的にアルゴリズムを使い分けることが必要になる．

なお，現在最速な行列積アルゴリズムは，Le Gall [5] により提案された $O(n^{\omega})$　$(\omega < 2.3728639)$ 時間アルゴリズムである．また，行列積問題の時間量の明らかな下界は，積に現れる $n^2$ 個の要素をすべて出力しないといけないので $\Omega(n^2)$ となる．ただし，$\Omega(f)$ とは，$f$ に比例する時間以上であることを意味する．

本節の残りでは，どのような分割統治アルゴリズムをつくればよいかを示すために，再帰式の解の構造を示す．

いま，

$$\begin{aligned} T(n) &= aT(n/b) + f(n) \\ T(1) &= c \end{aligned}$$

[2]すなわち，オーダ記法に隠された定数が大きい．

という分割統治アルゴリズムを構成したとしよう．ここで，$a$，$b$，$c$ は正定数とする．すなわち，前処理，後処理両方合わせて，$f(n)$ 時間を要し，サイズ $n$ の問題例を $a$ 個のサイズ $n/b$ の問題例に分解する分割統治アルゴリズムである．この再帰式について以下の結果がある．

1．ある正定数 $\epsilon$ に対して $f(n) = O(n^{\log_b a - \epsilon})$ ならば，$T(n) = \Theta(n^{\log_b a})$.

2．$f(n) = O(n^{\log_b a})$ ならば，$T(n) = \Theta(n^{\log_b a} \log n)$.

3．ある正定数 $\epsilon$ に対して $f(n) = \Omega(n^{\log_b a + \epsilon})$ であり，かつ，ある定数 $d < 1$ と十分大きな $n$ に対して $af(n/b) \leq df(n)$ ならば， $T(n) = \Theta(f(n))$.

ここで，$\Theta(g)$ とは，上下界がともに $g$ に比例することを意味する．直観的には，1．は前後処理に要する時間が再帰部に比べて小さいときを，2．は前後処理に要する時間が再帰部と同等であるときを，3．は前後処理に要する時間が再帰にかかる時間に比べて大きいが，それほど大きくない（再帰によって膨れ上がらない）ことを意味する．先ほどの Strassen アルゴリズムは，$a = 7$，$b = 2$，$f(n) = O(n^2)$ なので， 1．の場合に対応する．また，(1) を用いる分割統治アルゴリズムを考えると，$a = 8$，$b = 2$，$f(n) = O(n^2)$ となり，1．より， $T(n) = \Theta(n^3)$ を得る．これは，単純な手法と全く同じ計算時間を必要とすることを意味する．このように，よりよい分割統治アルゴリズムを作る際には，できるだけ小さな個数 $a$ で，なおかつ，できるだけ小さなサイズ $b$ の問題に分解することが必要であり，また，前後処理に要する時間 $f$ もできるだけ小さいことが望まれる．

## 3.2 動的計画法

本節では，効率的なアルゴリズム設計としての動的計画法を紹介する．

さきほどの分割統治法では，元の問題例を小さな問題例に分割し，単に再帰を用いて計算していた．すなわち，場合によっては，まったく同じ問題例を何度も解いている場合はあるかもしれない．動的計画法では，折角計算した小さな問題例に対する計算結果をうまく記憶などして再利用することにより，アルゴリズムを高速化させる方法である．下記には，行列の連続積の計算順を求める問題を例にとり解説する．

$\ell \times m$ 行列 $A$ と $m \times n$ 行列 $B$ の積 $AB$ を考えよう．単純な方法を用いると，$\ell mn$ 解の掛け算と $\ell n(m-1)$ 回の足し算の計 $2\ell mn - \ell n$ 回の演算で計算できる．前節で説明したアイデアを使うと高速に解けるのであるが，この節では，話を簡単にするため，行列積 $AB$ を $f(\ell, m, n) = 2\ell mn - \ell n$ 回の演算で計算するとして話を進める．

いま，$30 \times 1$ 行列 $A$，$1 \times 40$ 行列 $B$，$40 \times 10$ 行列 $C$，$10 \times 20$ 行列 $D$ が与えられたとき，$A \times B \times C \times D$ を考える．このとき，$((AB)C)D$，$(AB)(CD)$，$(A(BC))D$，$A((BC)D)$，$A(B(CD))$ のどの順で計算したほうが楽であろうか?

$$
\begin{aligned}
((AB)C)D:&\ f(30,1,40) + f(30,40,10) + f(30,10,20) = 1,200 + 23,700 + 11,400 = 36,300\\
(AB)(CD):&\ f(30,1,40) + f(40,10,20) + f(30,40,20) = 1,200 + 15,200 + 47,400 = 63,800\\
(A(BC))D:&\ f(1,40,10) + f(30,1,10) + f(30,10,20) = 790 + 300 + 11,400 = 12,490\\
A((BC)D):&\ f(1,40,10) + f(1,10,20) + f(30,1,20) = 790 + 380 + 600 = 1,770\\
A(B(CD)):&\ f(40,10,20) + f(1,40,20) + f(30,1,20) = 15,200 + 1,580 + 600 = 17,380
\end{aligned}
$$

この例でみるように，4 つの行列の連続積においても，最小 $1,770$，最大 $63,800$ と大きな差がでる．従って，多くの行列の連続積を考える際，どのような順序で計算したほうがよいかを，予め計算することは，全体としても計算量削減となる．本節では，この行列の連続積問題を考えよう．

**行列の連続積**

**入力**: $n$ 個の行列 $A_1, A_2 \ldots, A_n$．ただし，$A_i$ は $d_i \times d_{i+1}$ 行列である．

**出力**: 最小の計算順

さて，この問題も分割統治法のときと同様に，小さなサイズの部分問題に分割することを考える．

$$M(i,j):\ A_i \times \ldots A_j \text{を計算するのに必要な演算数の最少数} \quad (1 \le i \le j \le n)$$

と定義する．この定義においては，先ほども述べたように，２つの行列の積は単純な方法を用いるという仮定の下で，計算する順序をうまく選ぶことで最少化することを考えている．定義より，

$$M(i,i) = 0 \quad (i = 1, \ldots, n)$$

であり，我々の目的は，$M(1,n)$ の値とそれを満たす計算順を求めることである．いま，

$$A_i \times \cdots \times A_j = (A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_j)$$

と計算したとすると，すなわち，第 $i$ 番目から第 $j$ 番目まで計算するときに，第 $i$ 番目から第 $k$ 番目までを計算して得られた行列と第 $k+1$ 番目から第 $j$ 番目まで計算して得られた行列を掛けたとすると，$A_i \times \cdots \times A_k$ が $d_{i-1} \times d_k$ 行列，$A_{k+1} \times \cdots \times A_j$ $d_k \times d_j$ 行列であることに注意すると，途中の計算が最適であった場合，その全体の演算数は $M(i,k) + M(k+1,j) + f(d_{i-1}, d_k, d_j)$ となる．すべての $k$ を考えることにより，

$$M(i,j) = \min_{k:\, i \le k \le j-1} \big( M(i,k) + M(k+1,j) + f(d_{i-1}, d_k, d_j) \big) \tag{6}$$

を得る．あとは，この最適性の原理を用いるだけである．ただ，単純に再帰的に計算してしまうと，指数時間必要になる．そこで，(6) をよくみると，$M(i,j)$ を計算するときには，$|\ell - k| < |j - i|$ である $M(k,\ell)$ が予め計算してあれば，(6) の右辺の min 中に現れるそれぞれの式の値は単位時間で計算できる．すなわち，$M(i,j)$ は，(6) にあるように $j - i - 1$ 個の中から最小値を求めることにより，$O(j-i)$ 時間で計算可能である．これらのことから，$j - i$ が小さい順に順次 $M(i,j)$ を計算し，記憶していけば．$\sum_{i,j:1 \le i \le j \le n} O(j-i) = O(n^3)$ 時間で目的である $M(1,n)$ が計算できる．また，このアルゴリズムの計算手順を逆に辿ることにより，最適な行列積順も計算可能である．

なお，この行列の連続積の問題は，さらに工夫を加えることで，$O(n^2)$ 時間で解けることが知られている．

## 4　NP 困難性と近似アルゴリズム

最後に，本節では計算クラス NP，NP 困難性，NP 完全性などの概念を紹介するとともに，NP 困難な最適化問題に対する近似アルゴリズムの話題を扱う．

### 4.1 計算クラス NP

本節で扱う問題は，Yes か No かを出力する決定問題に限定する．まず，計算クラス P とは，入力サイズの多項式時間で計算可能な問題の集合をいう．また，計算クラス NP とは，どんな Yes を出力する問題例に対しても，その入力長の多項式サイズの証拠（ヒント）が存在し，その証拠を用いることにより，与えられた問題例が Yes であるかどうかを（入力サイズの）多項式時間で確認できる問題の集合である．ここで NP とは Nondeterministic Polyonomial の略であり，非決定性計算において多項式時間で解ける問題集合を意味する．直感的には，簡単に解ける問題が P であり，ヒントを教えてもらえば簡単に解けるのが NP である．NP を考える場合，テレビ番組で出題されるクイズや新聞に載せられているクイズを想像してもらえれば，イメージがつきやすいかと思う．これらのクイズはなかなか解けないのであるが，ヒントや解答を示されると納得する．すなわち，ヒントや解答の長さ自体は短く，また，それを用いた説明も短い．ただし，NP は Yes-No 問題の集合であることに注意されたい．

この定義から明らかに，クラス P は，証拠がなくても多項式時間で Yes か No か判定できる問題の集合なので，NP に含まれる．ここで，証拠なしとは，長さ 0 の証拠をもつと理解してほしい．また，これらの定義から，クラス P は Yes, No について対称的であるのに対して，クラス NP は対称的でないことに注意されたい．NP の定義において，Yes を No に置き換えた計算クラスは coNP と呼ばれる．これは，問題を言語とみなしたとき，補集合になることによる．さきほどの議論からわかるように，P は coNP の部分集合にもなる:

$$\mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}. \tag{7}$$

さて，上記の定義だけではイメージし難いので，具体的な問題を例にとり説明しよう．

**オイラー閉路問題**
**入力**：連結な無向グラフ $G = (V, E)$.
**出力**：もし $G$ 中にオイラー閉路があれば，Yes．そうでなければ，No.

**ハミルトン閉路問題**
**入力**：連結な無向グラフ $G = (V, E)$.
**出力**：もし $G$ 中にハミルトン閉路があれば，Yes．そうでなければ，No.

ここで，無向グラフとは，有限の頂点集合 $V$ と頂点の対である枝の集合 $E \subseteq \binom{V}{2}$ からなる．任意の 2 頂点 $v, w \in V$ 間にそれをつなぐパス $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \ldots, e_{k-1} = (v_{k-2}, v_{k-1}), e_k = (v_{k-1}, v_k)$ （ただし，$v = v_0$，$w = v_k$) が存在するとき，連結グラフとよばれる．なお，枝は，頂点の対であり，$e = \{v, w\}$ のように記述する本もあるが，本稿では，$e = (v, w)$ と記述し，$(v, w)$ と $(w, v)$ とを区別しない．また，長さ $k \geq 1$ 以上，かつ，$v_k = v_0$ となるパスを閉路という．オイラー閉路とは，すべての枝を丁度一度用いる閉路であり，ハミルトン閉路とは，すべての頂点を丁度一度通る閉路である．オイラー閉路とハミルトン閉路は定義から似ているのであるが，計算量的にはかなり違う問題と考えられている．

さて，このオイラー閉路問題とハミルトン閉路問題ともに NP に属する．なぜならば，例えば，オイラー閉路問題の場合を考えよう．もし，与えられた問題例であるグラフがオイラー閉路をもてば，すなわち，Yes である問題例であるならば，その存在するオイラー閉路自体を証拠としよう．明らか

にオイラー閉路は入力であるグラフのサイズに比例するので，多項式サイズの証拠である．また，その証拠が本当に入力のグラフのオイラー閉路であるかは多項式時間で簡単に確認できる．このことからオイラー閉路問題はNPに属する．同様の議論により，ハミルトン閉路問題では，ハミルトン閉路自体を証拠とすれば，NPに含まれることを容易に示せる．このようにNPに属する多くの問題においては,「解」自身を証拠とすることでNPに含まれることがわかる．

では，次にこれらの問題がcoNPに含まれるかどうか考えてみよう．すなわち，Noである問題例，オイラー閉路やハミルトン閉路をもたないグラフにおいて，本当に存在しないことをどのように説明すればよいのであろうか? オイラー閉路においては，次の定理が知られている．

**定理 4.1** 連結な無向グラフ $G = (V, E)$ 中にオイラー閉路が存在するための必要十分条件は，すべての頂点 $v \in V$ の次数が偶数となることである．

ここで，頂点 $v$ の次数とは，それに接続する枝の本数のことをいう．

この定理から，オイラー閉路をもたない連結グラフにおいては，次数が奇数である頂点が存在する．したがって，このような頂点を証拠とすればよい．この証拠は明らかに多項式サイズであり，本当にその頂点の次数が奇数であるか多項式時間で確認できる．よって，オイラー閉路問題はcoNPに含まれる．実際，定理を用いれば，オイラー閉路問題が多項式時間で示すことができる．なぜならば，coNPの証拠となる頂点を多項式時間で見つけることができるからである．このことから，

$$\text{オイラー閉路問題} \in \mathrm{P} \subseteq \mathrm{NP} \cap \mathrm{coNP}$$

と示すこともできる．

では，ハミルトン閉路問題はどうであろうか? あとで述べるが，ハミルトン閉路問題は，NP完全問題であり，多項式時間で解けるかどうか，もっというと，coNPに属するかどうかも知られていない．

次にNP困難性，完全性を議論しよう．そのために，まず，帰着（還元）を定義する．

いま問題Aを解くプログラム（アルゴリズム）を作りたい状況を考える．みなさんどのようにするだろうか? もちろんそのプログラムを書けば良いのであるが，多くの人は，たぶんインターネットにフリー（無料）のプログラムがないかを探すだろう．もちろん，そのようなプログラムが信頼できるかどうかという問題はあるが，もしインターネットで見つけられれば，それで終わりである．もし，見つけられない場合，問題Aを解くためにサブルーティンとして使えそうなプログラムを探すのではないか．すなわち，皆さんは帰着を利用したプログラムを制作しようとしている．より正確には，問題Bを解くアルゴリズムをサブルーティンとして利用する問題Aのアルゴリズムが存在するとき，問題Aを問題Bに帰着可能であるという．この帰着を利用したアルゴリズムにおいて，問題Bを解く部分を多項式時間と仮定したとき，全体が多項式時間になるとき，その帰着は多項式時間帰着とよばれる．この帰着はチューリング帰着と呼ばれるものであり，決定問題ばかりでなく，探索問題や最適化問題などにも定義可能である．決定問題に限定した帰着としては，次の多対一帰着が有名である．問題Aの任意の問題例 $I$ を問題Bの問題例 $J$ に移す関数 $f(I) = J$ が存在して，以下の条件を満たすとき，問題Aは問題Bに多対一帰着可能であるという：

$$I:\ \text{Yes 問題例} \iff f(I):\ \text{Yes 問題例}$$

定義から，問題 A を問題 B に多対一帰着した場合，対応するアルゴリズムは，問題 B を解くサブルーティンは丁度一度呼ばれ，そのサブルーティンの出力がそのまま全体のアルゴリズムの出力となる．従って、多対一帰着は，チューリング帰着の特殊な場合とみなすことができる．この多対一帰着においては，関数 $f$ が多項式時間で計算可能であるとき，多項式時間帰着と呼ばれる．

NP に属する任意の問題が問題 A に多項式時間帰着可能であるとき，問題 A は NP 困難であると呼ばれる．問題 A が NP 困難であり，かつ，NP に属するとき，$A$ は NP 完全と呼ばれる．定義からすぐに NP 困難な問題や完全な問題が存在するかどうか明らかではないが，次の充足可能性問題 (SAT) がそのような問題であることが知られている．

**充足可能性問題 (SAT)**
**入力**：論理積形 (CNF) $\varphi$.
**出力**：$\varphi(x) = 1$ となる $x$ が存在すれば，Yes．そうでなければ，No.

例えば，入力の論理積形として $\varphi(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \overline{x}_3)(\overline{x}_2 \vee \overline{x}_3)(\overline{x}_1 \vee x_3)$ を考えると，$\varphi(1, 0, 1) = 1$ より，Yes となる．

**定理 4.2 (Cook-Levin 定理)** 充足可能性問題は NP 完全である．

この証明は，本稿では省略する．概略を述べると，本稿では，NP の定義に「証拠」を用いたが，非決定性チューリング機械を用いても定義が可能である．すなわち，NP に属する問題は，非決定性チューリング機械により多項式時間で受理される言語と同一視できる．この非決定性チューリング機械の動作そのものを論理積形と表現し，受理することが論理積形を満たすことと等価であることを示し，証明している．

この SAT のように NP 完全な問題が 1 つ分かれば，あとは帰着を用いて様々な問題が NP 完全あるいは困難であることがわかる．たとえば，上記のハミルトン閉路問題の NP 完全である．このように非常に単純な問題が NP 完全な問題であることがわかる．

NP 困難性の定義より，NP 完全の問題がどれか 1 つ，たとえば，充足可能性問題が多項式時間で計算可能であれば，NP に属するすべての問題が多項式時間で解けることを意味する．しかしながら，現在それらの問題が多項式時間で解けるかどうか分かっていない．さきほども述べたが，coNP に属するかどうかでさえ知られていない現状にある．

## 4.2 近似アルゴリズム

NP 困難な問題は多項式時間で解けないと信じられている．実際証明された訳でもないので，ひょっとして多項式時間アルゴリズムが存在するかもしれないが，少なくとも現時点では，そのようなアルゴリズムは発見されていない．しかし，現代社会の様々な場面で，NP 困難な問題を解くことが要請されている．このような現状でいったいどのようなアルゴリズムを開発すればよいのであろうか．アルゴリズム論の分野では 2 つの軸でアルゴリズムの設計が行われている．

第一番目の軸は，正確に解くか，近似的に解くかという軸である．世の中には正確に解くことを要求されている問題も多い．このような問題においては，計算時間を犠牲にしても，すなわち，最悪指数時間かかることは覚悟しなくてはならないが，色々工夫することで，より速いアルゴリズムの設計

を目指すものである．それとは逆に，少々解の質は悪くなってもいいが，高速に解かなくてはならない問題もある．こうした問題においては，如何に高精度な近似解を多項式時間で解くかを議論する．

第二番目の軸は，精度保証をするかしないかという軸である．すなわち，計算量理論，アルゴリズム論を背景に，出力する解の精度や計算時間を保証しようという試みと，理由はともかく，実際に解く問題例において，高速に解くアルゴリズムを求めようとするものである．前者は，近似精度保証アルゴリズムや高速な指数時間アルゴリズムなどの分野として盛んである．後者は，実用的に重要であり，メタヒューリスティックアルゴリズムとして広く知られている．

本講演では，最大充足可能性問題を例にとり，近似アルゴリズム設計法や精度保証のアイデアなどを議論する．

## 参考文献

[1] Sara Baase, *Computer Algorithms : Introduction to Design and Analysis*. Addison-Wesley, 1988.

[2] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein, *Introduction to Algorithms*, MIT Press, 1990.

[3] 茨木俊秀, アルゴリズムとデータ構造, 昭晃堂, 1989.

[4] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thome, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann, Factorization of a 768-Bit RSA Modulus, *CRYPTO 2010*, Lecture Notes in Computer Science 6223, 333-350, 2010.

[5] Francois Le Gall, Powers of Tensors and Fast Matrix Multiplication, http://arxiv.org/pdf/1401.7714v1.pdf.

[6] Ron Rivest, Adi Shamir and Leonard Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, *Communications of the ACM* 21, 120-126, 1978.

[7] http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm.