**解説** *New Paradigms for Designing Memory-Constrained Algorithms*

# 省メモリのための新たなアルゴリズム設計技法

## 制限された作業用メモリでアルゴリズムをいかに設計するか 前編

浅野 哲夫 北陸先端科学技術大学院大学情報科学研究科

IT 技術の進歩は目を見張るばかりである．急速な進歩を遂げる携帯電話やデジタルカメラ等には計算機並みのソフトが組み込まれている．今では顔認識はデジタルカメラでは定番となっているが，そのような高度な処理をカメラのような装置で実現されると誰が予想しただろう．携帯電話も機能的には計算機にどんどん近づいており，iPad の出現を待って，実にさまざまなアプリケーションが導入されるに至っている．そのような高機能端末上のプログラム開発が最近になって脚光を浴びている．

最近までは組込みシステムという名前の下に研究されてきたが，アンドロイドなどの組込み OS の出現により，その守備範囲は飛躍的に拡大している．ここで問題になるのが記憶領域のサイズである．通常の計算機の環境に比べて，利用できるメモリ量はかなり制限されているのが普通である．大量のメモリを使うことに慣れてしまったプログラマにとっては制限されたメモリ環境でプログラムを開発することは苦痛ですらある．ソフトウェア工学の分野では，スモールプログラムと呼ばれる概念の下に省メモリ・プログラミングの開発法も提唱されているが，アルゴリズム設計のレベルでも積極的な取り組みがあるかというと，そうとも言えないのが現状である．経験と勘に基づいてアルゴリズムが設計されてきたと言って過言ではない．

上に述べた状況から，本稿では，作業領域が制限された状況でのアルゴリズム設計技法のいくつかを紹介する．大きな特徴は，作業領域が制限されているので，複雑なデータ構造は使えないことにある．これはアルゴリズムそのものがシンプルになるという利点も兼ね備えている．アルゴリズムがシンプルであれば，解析も容易だし，プログラムとしての実装も簡単である．バグも少ないだろう．コロンブスの卵と同じで，知っていれば何でもないことであるが，知らなければ不可能なのである．

### 44544000 分の 1 を正確に計算しよう

最初の例題として，44544000 分の 1 の計算について考えよう．電卓を利用すると，

0.000000022449712643678160919540229885057

という結果が得られる．44544000 を素因数分解してみると，

$$44544000 = 3 \times 29 \times 2^{12} \times 5^{3}$$

となり，2 と 5 以外の素因数を持つので，割り切れることはない．したがって，必ず循環小数となるが，電卓の結果からは，どこに循環があるのかが分からない．

では，どうすれば循環部分を求めることができるだろう．中学か高校では，毎回，余りを 10 倍しては $n$ で割ったときの商と余りを求めるという操作を同じ余りが出るまで繰り返すということを学んだはずである．実際に計算してみると次の結果を得る．

1/44544000 = 0.000000022449(71264367816091
95402298850574)

学校では循環の最初と最後の数字の上にドットを置くというような表記法を学んだが，ここでは 1 行で表現するために，循環部分を括弧を用いて表現して

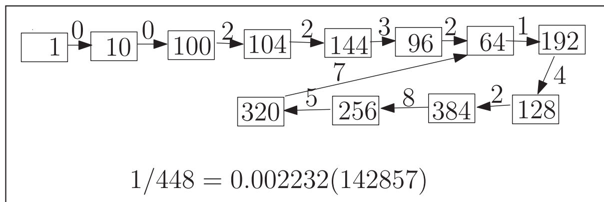


**図 -1　1/448 を循環小数に変換する様子．枠内の数字は毎回の剰余を，矢印の近くに付した数字は毎回の商を表している．**

いる．

上に述べた考え方に従ってプログラムを書こうとすると，毎回の商と余りを順に蓄えていくための配列が必要となる．その配列のサイズはどのように設定すればいいだろう？　整数 $n$ による除算の余りは 0 から $n$−1 までの $n$ 通りしかないから，配列のサイズは $n$ でよい．しかし，1/44544000 の計算のためだけに，4000 万を超えるようなサイズのメモリを使わないといけないのだろうか？

## ➤ アルゴリズムの観点から問題を見直そう

アルゴリズムの観点から問題を見直してみよう．そのために，もう少し小さな整数 $n$ を考えよう．**図 -1** は 1/448 を循環小数に変換する様子を示したものである．毎回計算される商(矢印に付した数字)と剰余(枠内の数字)で模式的に表したものである．

図からも明らかなように，1 という初期値から出発して，ループに陥るか，あるいは余りが 0 となって終了するか，を判定することが問題である．余りは高々 $n$ 通りしかないので，$n$ 回以上計算を繰り返しても終了しなければ，循環小数であることは確かである．しかし，どこから始まるのかを求めるのは別の問題である．先に述べた例では，$n$ の値は 4000 万を超えているが，循環部分の長さは 28 程度でしかない．

## ➤ 問題はループの先頭を発見すること

ここまでの観察で，問題は要するにループの発見にあると言っても過言でない．では，配列を使わずにループの先頭を求めるにはどうすればいいだろうか？

重要なことは，余りの初期値 1 から始めて上に述べた計算(現在の余りを 10 倍したものを $n$ で割って，商と余りを求めるという計算）を繰り返して余りの系列を求めるという一連の計算は，整数 $n$ の値が分かっていれば，いつでも完全に同じ形で実行できるということである．つまり，いつでも先頭からの余りの系列を計算で求めることができるということである．いつでも再計算が可能だから，何も結果を蓄えておくことはないだろう，というのが考え方である．

## ➤ 素朴なアルゴリズム

ここまで分かればアルゴリズムは自明だろう．余りの初期値を 1 から始めて，現在の余りを 10 倍したものを $n$ で割ったときの商と余りを求めるという計算を，余りが 0 になるか，同じ余りが出現するまで繰り返せばよい．$k$ 回目の繰り返しで得られた余り $R_k$ が過去に出現したかどうかは，余りを 1 から始めて同じ計算を $k$−1 回だけ繰り返し，その間に $R_k$ と同じ余りが出るかどうかを調べればよい．したがって，余りを全部記憶しておかなくても計算はできるのである．

基本的には上に述べた通りである．しかし，実際にプログラムを書こうとすると，厄介な問題がある．出力するのは商の列であるが，単純に商を順に出力したのでは，循環部の先頭を示す左括弧 "(" を出力することができない．循環部の先頭に気が付くのは，その計算を終わった後だからである．では，どうする？　簡単なことである．最初の計算では，何番目の繰り返しが循環部分の先頭かを求めるのである．$k$ 番目に得られた余りが循環部の先頭に相当するなら，とにかく上記の計算を $k$−1 回行って，先頭の 0.1 に続けて，毎回の商を順に出力していく．その後で循環部の先頭を表す左括弧 "(" を出力し，その時の商を出力する．後は，このときの余りが再び出現するまで，同じ計算を繰り返し，商を順に出力していく．最後に右括弧 ")" を出力して終了である．

これで配列を使わなくても 1/$n$ の計算を正確に実行できることが分かった．$n$ で割ったときの余り

は高々 $n$ 通りしかないので，繰り返しの回数は高々 $n$ 回である．毎回，過去に同じ余りが出たかどうかを判定しているので，アルゴリズム全体は 2 重ループの構造になっている．したがって，計算時間は最悪の場合には $n$ の 2 乗に比例する程度（これをアルゴリズム理論では $O(n^2)$ という記号で表す）になってしまう．

### ➤ 高速化のための画期的な定理

メモリは削減できたが，非常に遅くなってしまった．次に考えるのは，同じメモリで高速化ができるかどうかである．実は，過去にとんでもない定理を発見した研究者がいる．上にも述べたように，循環部が何桁目から始まるかさえ分かれば，問題は解けたも同然である．上の方法では実際に同じ余りを見つけたが，まったく別の方法でも求めることができるのである．その驚くべき定理 [1] とは，入力の整数 $n$ が何回 2 と 5 で割り切れるかを求めて，その大きい方の値に 1 を加えたものが循環部の先頭の位置を表すというものである．44544000 の素因数分解が

$$44544000 = 3 \times 29 \times 2^{12} \times 5^3$$

であることはすでに見た．つまり，44544000 という整数は，2 では 12 回割ることができ，5 では 3 回割り切ることができる．したがって，max(12, 3) +1 = 13 桁目が循環部の先頭である．これは先に見た循環小数表現と合致する．

$$1/44544000 = 0.000000022449(7126436781609195402298850574)$$

2 と 5 で割る操作を定数時間で実行できると仮定すると，この方法は循環小数表現の長さに比例する程度の時間で終わる．つまり，計算時間は最悪の場合でも $O(n)$ である．

### ➤ 数論の定理を使わない高速化

上に述べた数論の定理を理解するには群論の知識が必要になる．では，難解な定理を使わずに同じ線形時間で $1/n$ を計算する方法はないだろうか？　ここでは「兎と亀法」と呼ばれるアルゴリズムを紹介しよう．この方法では，兎と亀に対応する 2 つの変数

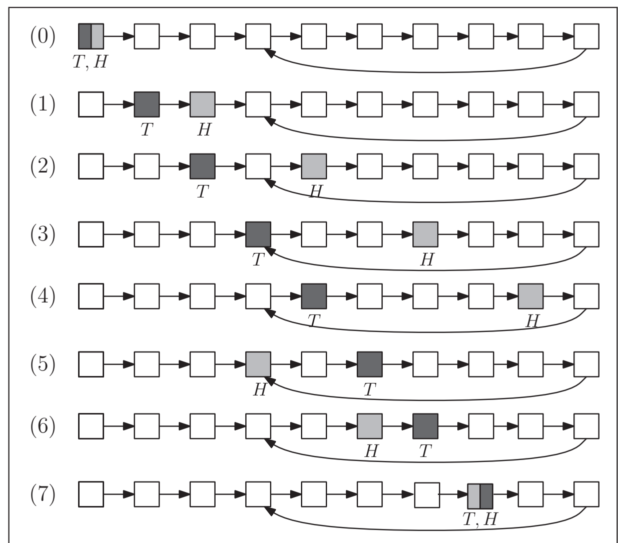


図 -2　速度 1 のポインタ $T$ と速度 2 のポインタ $H$ を順に進めていくと，やがて両者が同じ場所を指すようになる．

を用いる．どちらの変数も，上に述べた基本操作（現在の余りを 10 倍したものを $n$ で割ったときの商と余りを求める操作）を実行した後の余りを管理するものである．違いは，亀に対応する変数は，1 回の基本操作の後の余りを管理するのに対して，兎に対応する変数は，基本操作を 2 回連続して実行した後の余りを持っておくことにする．

$1/n$ が循環小数でないなら，先を行く兎の計算で余り 0 が出現する．そうでない場合にはループに陥る．兎の方が先にループに入り，後で亀がループに入ることになる．いったんループに入ってしまえば，そのループから逃れることはできない．兎と亀の速度差を考慮すると，いつかは必ず兎と亀が同じ場所に到達することになる．

**図 -2** は兎が亀に追いつく様子を図示したものである．亀がループに入り口に到達したとき，兎から亀までの距離（両者の位置を隔てる辺の個数）を $d$ としよう．図 -2 の例では，3 回目の繰り返しで亀がループの入り口に到達するが，このとき兎から亀までの距離は 4 である．そこから 4 回目の繰り返しで兎が亀に追いついている．

この方法では作業用の配列を使うことなく，単純変数を 2 個使うだけでループの有無を判定している．しかし，ここでの目的は何だっただろうか？

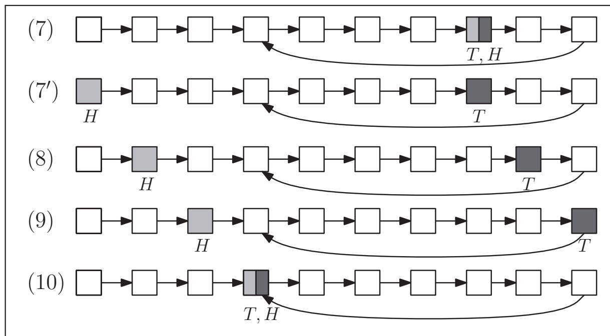

図-3　兎と亀が出会った後は，兎だけ出発点に戻した後で，両者同じ速度で進め，再び出会った所で止まる．

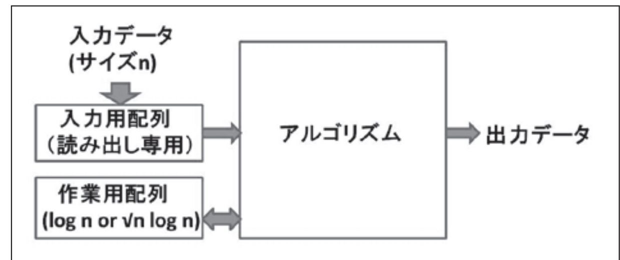

図-4　省メモリアルゴリズムの計算モデル

単にループの有無を判定するだけではなく，ループの場合にはその入り口を発見したかったことを思い出そう．ではどうすればいいだろう？

兎と亀が出会った後，兎だけを出発点に戻す．具体的には，兎で管理している余りを初期値の 1 に戻すのである．その後，兎と亀は基本操作を実行しては結果の余りを管理する．以前と違うのは，今度は速度を同じにするのである．毎回，兎と亀のそれぞれに対して基本操作を 1 回だけ実行して，それらが同じになったかどうかを判定するのである．**図-3** は，兎と亀が出会って以降のアルゴリズムの振る舞いを説明したものである．ちょうどループの入り口で両者が出会っていることが分かるだろう．実は，上記の方法で必ず両者はループの入り口で出会うことを証明することができる．証明は読者の楽しみに残しておこう．

ここでは与えられた整数 $n$ に対して，その逆数 $1/n$ の値を循環小数表現を用いて正確に求める方法をいくつか紹介した．最初のアルゴリズムはサイズ $n$ の作業用配列を使って $O(n)$ 時間で計算するものであった．作業用の配列を使わなくても同じ計算ができることを示したのが「素朴なアルゴリズム」であるが，計算時間は $O(n^2)$ かかった．次に数論の知識を使うと定数領域でしかも $O(n)$ 時間で問題が解けることを示した．最後に示した「兎と亀法」は，数論の知識がなくても，同じ定数領域と $O(n)$ 時間で問題が解ける．兎と亀法は定数領域でループを求めるための一般的な方法であり，さまざまな問題に応用できる．

## 計算のモデル

ここまで簡単な例題に沿って省メモリアルゴリズムについて述べてきたが，無用な誤解を避けるために計算のモデルを定義しておこう．

**図-4** は，本稿で仮定する省メモリアルゴリズムの計算モデルを示したものである．入力データが多数の場合には配列に蓄えるが，それ以外の作業用の配列のサイズを最小限に抑えるのが目的である．入力データのサイズ（入力データの個数）を $n$ としたとき，それらの入力データを蓄えておくのに $n$ に比例するメモリ量が必要になる．配列を使って蓄える場合，0 から $n-1$ の間の整数をインデックスとして用いて配列の任意の要素を指定して，アクセスを行う．連結リストに蓄える場合でも，それぞれのリスト項目を参照するためには，これら $n$ 個の場所を区別するのに十分な長さのアドレスが必要になる．厳密な意味では，上記のアクセスを可能にするのに $\lceil \log_2 n \rceil$ のビット数を持つ語を用いる必要がある．また，何かをカウントする場合にも，$n$ までの数はカウントできないと意味がないので，やはり同じ長さの語が必要になる．したがって，厳密な意味では，$\log_2 n$ ビットの長さの語を何個必要とするかで作業領域のサイズを議論すべきであるが，パソコンで扱えないような巨大な数のデータを扱うのは稀なので，本稿では変数を何個使うのかだけに注目する．

さて，入力データは配列に蓄えるが，その配列を作業領域の一部として使ってよい場合と，そうでな

い場合がある．入力用の配列を作業用に用いるということは，その配列の内容を書き換えてもよいことを意味している．たとえば，$n$ 個のデータを入力して，その中に同じ値のものが含まれているかどうかを調べる問題を考えてみよう．$n$ 個のデータを配列に蓄えた後，データを配列上で昇順に並べ替えると，配列を順に走査するだけで同じ値の要素が含まれていたかどうかが分かる．値が同じなら，必ず連続して現れるからである．ソートのためのアルゴリズムは多数あるが，ここではソートすべき配列以外には作業用の配列を使わない「その場でのソートアルゴリズム」が必要である．たとえば，ヒープソートを用いると，その場でサイズ $n$ の配列を $O(n \log n)$ の時間でソートしてくれる．

### ➤ その場でのアルゴリズムとの相違点

本稿で対象とするアルゴリズムは，その場でのアルゴリズムではない．入力データは配列に蓄えておくが，要素の値を読むことだけはできても，その値を変更することは一切許さない (read-only array) と仮定している．その場でのアルゴリズムに比べて制限が強いが，それでも興味あるアルゴリズムが設計できる．また，入力データは実際にはメモリになくても，いつでも任意の場所のデータを測定できるとか，関数によって値を計算できるというような場合にも対応できるという強みがある．たとえば，カラー画像を入力して，指定した色の範囲に入るかどうかで 2 値の画像に変換するという場合を考えてみよう．もちろん，各画素について，その色が指定範囲に入るかどうかを判定して，その結果を別の配列の対応する場所に格納するという方法で 2 値画像が得られる．このようにして得られた 2 値画像の配列は，任意の要素の値を参照することも，その値を書き替えることもできる普通の配列である．しかし，そのために余分なメモリが消費されていることも事実である．もし値の変更を必要とせず，対応する値が 0 か 1 かを参照したいだけなら，その値を配列に蓄える必要はない．単に指定した色範囲に入っているかを判定するだけで値が分かる．これは，2 値画像が読み出しのみ可能な配列に入っていると考えるのと同じである．

### ➤ 定数領域アルゴリズム

上では簡単な例を考えた．RGB の 3 原色で指定された色が入力画像に格納されているが，別の表色系，たとえば L*a*b* で色の範囲が指定されるとき，変換の作業が必要になる．そのような変換は画像のサイズに依存しない定数時間でできるので，事情は同じであると考えることにする．

さて，本稿では，入力データが読み出し専用の配列に入っていると仮定した上で，できるだけ少ない作業領域で問題を解くためのアルゴリズム設計技法を考える．極端な場合は，入力データを蓄える配列以外には定数個の変数しか使わないというものである．そのようなアルゴリズムのことを本稿では「定数領域アルゴリズム」(constant-work-space algorithm) と呼んでいる．先に述べたように $1/n$ の循環小数表現を求めるアルゴリズムは定数領域アルゴリズムの具体例である．しかし実際にはもう少しメモリに余裕があると考えてよい．たとえば，入力データの個数 $n$ が 1 テラ ($10^{12}$) であるとき，さすがにサイズ $n$ の配列を別に確保するのは難しいが，$\sqrt{n}$ に相当する 1 メガ ($10^6$) のサイズの配列なら問題ないという場合もあるだろう．そこで，理論的には $\sqrt{n}$ 程度の作業領域で効率もよいアルゴリズムに興味があるが，そのようなアルゴリズムは実はあまり知られていない．本稿では，第 2 回の解説でそのようなアルゴリズムにも触れる．

個々の問題を省メモリの立場でどのように解決するかも興味深いが，本稿ではより一般的に，作業領域が限定された環境下でのアルゴリズム開発技法について紹介する．

### ➤ 並列シミュレーション法

最初の例題では「兎と亀法」を紹介した．整数 $n$ に対して $1/n$ の値を正確に計算するのに，1 つの制御変数だけでも定数領域アルゴリズムを設計することができたが，非常に遅いのが難点であった．「兎

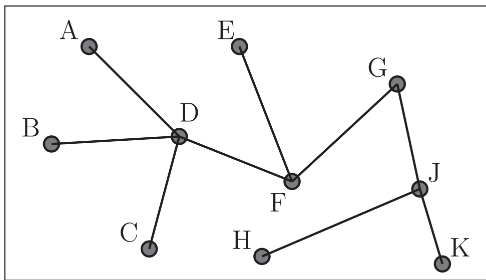

図 -5　平面上に描かれた木の例

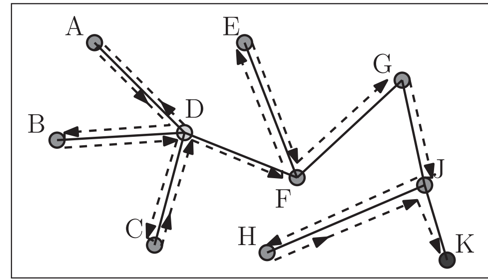

図 -6　図 -5 の木で $D$ から始まるオイラー閉路で節点 $K$ までの部分の発見，あるいは節点 $D$ からの深さ優先探索．

と亀法」のアイディアは制御変数を 2 つに増やすことであった．このアイディアのおかげで高速化を達成することができた．次に紹介する問題に対しても同じ考え方を適用できることを示そう．

連結で閉路を含まない無向グラフを木という．木の任意の 2 節点間には同じ節点を複数回通らない単純なパスが 1 つだけ存在することはよく知られている．節点数に比例するメモリが使えるなら，単純なパスを発見するのも容易であるが，定数領域でも同じことが可能であることを示そう．

10 個の節点からなる木の例が**図 -5** に示されている．この木を指定するのに，ここでは単純な隣接リストを仮定している．節点 $D$ には $A, B, C, F$ の 4 節点が隣接しているが，$Adj(D) = <A, B, C, F>$ という形式で与えられるものとする．

さて，この木において 2 節点 $D$ と $K$ を指定して，両者の間の単純なパスを見つける問題を考えよう．木の各辺を 2 本の辺で置き換えると，どの節点の次数（接続する辺の本数）も偶数になるので，オイラーグラフができあがる．よく知られているように，オイラーグラフには，すべての辺を 1 度だけ通る閉路が存在する．実際，オイラー閉路を求める定数領域アルゴリズムを設計するのは容易である．

図 -5 の例でアルゴリズムを説明することにしよう．まず，アルゴリズムは節点 $D$ から始める．$D$ の隣接リスト $Adj(D) = <A, B, C, F>$ を調べ，リストの最初の節点 $A$ へと移動する．次に節点 $A$ の隣接リストを調べるが，$Adj(A) = <D>$ なので，$D$ に戻る．再び $D$ の隣接リストを調べるが，直前の節点は $A$ だったので，$A$ を隣接リストの中で見つけ，その次の節点を選んで移動する．この例では，$Adj(D) = <A, B, C, F>$ なので，$A$ の次は $B$ である．同様にして，$B \rightarrow D \rightarrow C \rightarrow D \rightarrow F$ と移動する．以下，同様にして，$F \rightarrow E \rightarrow F \rightarrow G \rightarrow J \rightarrow H \rightarrow J \rightarrow K$ と順に移動して，$K$ に到達する．これは木の上での深さ優先探索（DFS：Depth-First Search）にほかならない．この移動の様子を表したのが**図 -6** である．

このようにすると，木 $T$ の任意の 2 節点 $s$ と $t$ に対して，$s$ から $t$ に至る経路を生成することができる．その関数（あるいは手続き）を DFSPath$(T, s, t)$ と名付けよう．このようにして見つけた経路は一般に単純ではない．同じ辺を 2 度含むことがあるからである．しかし，2 度含まれる辺は出力しないようにすることができれば単純な経路を出力することができる．

そのために，DFSPath$(T, s, t)$ で求めた各辺について，その辺が 2 度使われているかどうかを判定し，1 度しか使われていない辺だけを出力するようにすればよい．使われた辺を配列に記憶することができれば簡単であるが，ここでは作業用の配列を用いることはできない．ではどうするか？

以前と同じである．ある特定の辺 $e$ が何回使われたかを知るためには，もう一度 DFSPath$(T, s, t)$ を呼び出して，辺 $e$ の出現回数を求めればよいのである．したがって，今度も 2 重ループの形で指定された 2 節点を結ぶ単純なパスを求めることができる．しかし，2 重ループなので 2 乗に比例する時間がかかってしまう．

以前と同じように高速化は可能だろうか？　実

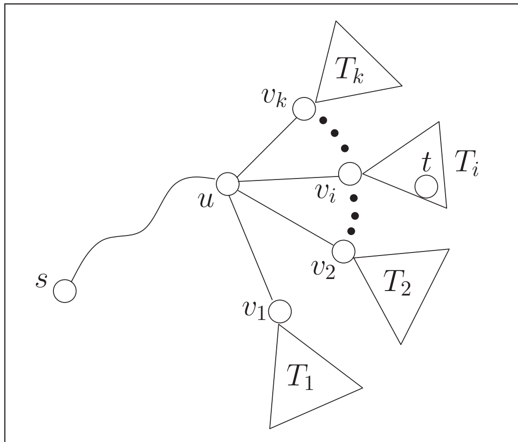


**図 -7　節点 $u$ の隣接節点を根とする部分木の中でどれが目標頂点 $t$ を含むか.**

は，以前と同様のアイディアで高速化が可能である．**図 -7** は，始点 $s$ から目標頂点 $t$ に至る単純なパスの中で節点 $u$ までの部分が計算できた状態で，$u$ の次にどの隣接辺を進めばよいかを判定しようとしている状況を示したものである．図のように，$u$ の隣接節点を根とする部分木の中で目標頂点 $t$ を含むものが見つかれば，$u$ の次にはその部分木に向かって進めばよいことになる．

図のように，まだ調べていない $u$ の隣接節点を $v_1, v_2, ..., v_k$ とし，各 $v_i$ を根とする部分木を $T_i$ と書くことにしよう．$T_i$ が $t$ を含んでいるかどうかは，$T_i$ において $v_i$ から深さ優先探索を行うことによって，定数領域と線形時間で判定することができる．しかし，これらの部分木を順に調べたのでは全体の計算時間を改善することはできない．そこで，2 つの部分木 $T_i$ と $T_{i+1}$ を選んで，それらに同時に深さ優先探索を適用する，というのがここでのアイディアである．具体的には，2 つの部分木での深さ優先探索を交互に 1 ステップずつ実行するのである．

一方の部分木 $T_i$ における深さ優先探索が目標頂点 $t$ を見つけることなく終了した場合には，次に $T_{i+2}$ の探索に移る．$T_{i+1}$ が先に終了した場合も同じである．

そうではなく，$T_i$（または $T_{i+1}$）における深さ優先探索で $t$ が見つかった場合には，もちろん，そこで探索を終了し，$u$ から $v_i$（または $v_{i+1}$）への辺を出力の一部として出力した後，現在の注目頂点を $u$ から $v_i$（または $v_{i+1}$）に移し，同じ操作を続ける．

実は，もう 1 つの場合がある．それは，$u$ に関する部分木のうち最後の 2 個を残して探索を行う場合である．このとき，一方の部分木が $t$ を含むことを先に発見してしまえば，これはすぐ上に述べた場合と同じである．そうではなく，$t$ が発見される前に，一方の部分木での探索が完了してしまうことがある．この場合，$t$ は最後に残った部分木に必ず含まれるので，探索を中止するのが肝心である．

図 -6 の例に戻って説明しよう．始点は $D$ である．$Adj(D) = <A, B, C, F>$ なので，それぞれの節点を根とする部分木に目標点 $t=K$ が含まれているかどうかを調べる．最初は，$A$ と $B$ を根とする部分木 $T(A)$ と $T(B)$ を調べる．先に $T(A)$ に目標点が含まれないことが分かるので，次に $T(B)$ と $T(C)$ を調べる．$T(B)$ が目標点を含まないので，次は $T(C)$ と $T(F)$ を調べることになる．$T(C)$ と $T(F)$ の部分木を 1 節点ずつ交互に調べていくが，$T(C)$ に目標点がないことがすぐに分かり，しかも $T(F)$ が最後の部分木なので，この場合には $T(F)$ を調べることなく，$T(F)$ に目標点が含まれていると判定することができる．そこで，節点 $D$ に移動し，$D$ の隣接節点のうち，まだ調べていない $E$ と $G$ について調べる．今度も部分木 $T(E)$ に目標点がないので，ただちに $G$ に移動することができる．

**定理 1**　$n$ 個の節点を持つ木と，任意の 2 節点 $s$ と $t$ が与えられたとき，定数領域だけを用いて $O(n)$ 時間で $s$ から $t$ に至る単純なパスを求めることができる．

解説記事なので詳細な証明は与えないが，無駄な探索を行っていないことを証明できれば計算時間の線形性が言える．ある部分木で $t$ が発見された場合，そこでの探索は無駄ではない．同時に別の部分木でも探索を行っているが，探索を同時に進めているので，無駄な探索に要した時間は，有効な探索に要した時間と同じだと言えるので，相殺される．最後に 2 個の部分木だけが残って，一方の探索が $t$ を見つけずに先に終了した場合も，その部分木を再び探索

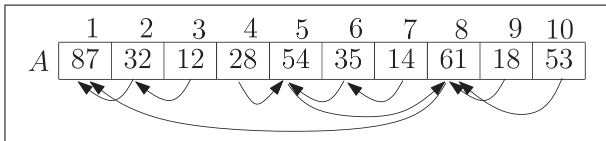

図 -8　直近上位要素発見問題（自分より大きい最も近い要素を求める問題）

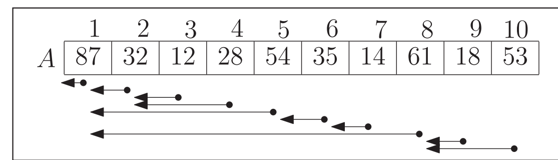

図 -9　自分より左の直近上位要素を求めるための探索

することはないので無駄ではない．また，最後に残った部分木での探索についても，他方の探索が完了した時点で終了しているので，やはり無駄な探索のコストは相殺されるのである．

## 省メモリアルゴリズム：直近上位要素発見

省メモリアルゴリズムの別の例として，直近上位要素発見問題 (Nearest Larger Neighbor Problem) について考えよう．今あなたは戦場にいるものとしよう．次々と仲間が戦死していく中で，兵士は最も近くにいる上司を探して，その命令に服さなければならない．それぞれの兵士について直近の上司を探す問題はどの程度難しいだろうか？

この問題を 1 次元配列の上で考えてみよう．具体的には，$n$ 個の数値データを格納した配列 $A[1..n]$ が与えられているものとする．もしすべての要素が異なるなら，最大値以外のどの要素 $A[i]$ についても，$A[i]$ より大きな要素が存在するが，それらの中で (インデックスの差の意味で) 最も近いものを求めたい．図 -8 に具体例を示す．この例では，$A[1] = 87$ が最大なので，この要素は上位要素を持たないが，それ以外の要素の上位要素は矢印で示されている．

この 1 次元の問題はどの程度難しいだろう？　作業領域はいくら使ってもよいという方針でアルゴリズムを設計してみよう．配列の各要素 $A[i]$ について，$A[i]$ より大きい要素の中で最も近いもの（直近上位要素と呼ぼう）を求めたい．直近上位要素は左右どちらにあるか分からないので，ここでは，$A[i]$ より左にある直近上位要素の場所（インデックス）と，右にある直近上位要素の場所を蓄える配列 L[] と R[] を用意する．図 -8 の例で，$A[5] = 54$ については，左に進むと $A[1] = 87$ でようやく上位要素を発見するので，$L[5] = 1$ である．5 番目から右に進むと，$A[8] = 61$ が最初の上位要素なので，$R[5] = 8$ である．$L[5] = 1$ と $R[5] = 8$ のどちらが 5 番目に近いかを判断して，$A[8]$ が $A[5]$ の直近上位要素であると分かる．

左の直近上位要素を求める問題と，右の直近上位要素を求める問題は対称的であるので，ここでは左の直近上位要素を求めるアルゴリズムだけを説明しよう．最も単純な方法は，各要素 $A[i]$ について，インデックス $j$ を $j = i-1, i-2, \ldots$ と順に下げていきながら，$A[i]$ より大きい要素 $A[i]$ があるかどうかを調べる．そのような要素が見つかれば，左の直近上位要素が見つかったことになるので，$L[i] = j$ として $A[i]$ に対する処理を終わる．配列の左端 $A[1]$ まで探しても $A[i]$ より大きい要素が見つからなかった場合は，$L[i] = -n$ とする．

図 -9 は，左の直近上位要素を求めるために行う探索の様子を矢印を用いて図示したものである．$A[8] = 61$ については多数の要素を見ていることが分かる．極端な場合，左端 $A[1]$ に最大値があり，$A[2..n]$ の部分の要素は昇順に並んでいるとすると，どの $A[i]$, $i \geq 2$ についても $A[i-1], A[i-2], \ldots, A[2]$ は $A[i]$ 以下なので，必ず $A[1]$ まで探索しなければならないことになる．つまり，$i-1$ 個の要素との比較が必要になる．よって，その総和 $1+2+ \ldots + (n-1)$ は $n(n-1)/2$ となり，$n^2$ に比例する手間がかかることが分かる．

もう少し高速化はできないだろうか？　実は，スタックを使うと最悪の場合でも線形時間で処理を終えるようにすることができる．次にその方法を紹介しよう．

図 -9 の例を用いて基本的な考え方を説明しよ

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 87 | 32 | 12 | 28 | 54 | 35 | 14 | 61 | 18 | 53 |
|  |  |  |  |  |  |  | 14 |  |  |  |
|  |  |  | 12 | 28 |  | 35 | 35 |  | 18 | 53 |
|  |  | 32 | 32 | 32 | 54 | 54 | 54 | 61 | 61 | 61 |
|  | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 | 87 |

**図 -10　自分の左の直近上位要素発見の際のスタックの内容の変遷**

う．$A[4]$ の左を探索するとき，$A[3] = 12, A[2] = 32$ の順に見ていく．$A[3] < A[4], A[2] > A[4]$ なので，$A[2]$ が $A[4]$ の左直近上位要素である．この後同様の探索を進めていくが，$A[3]$ は $A[4]$ 以下だったので，$A[5]$ 以降の要素から左に探索を進めたときに，$A[3]$ がその要素の左直近上位要素になることはない．なぜなら，その直前に調べる $A[4]$ の方が大きいからである．したがって，この時点で $A[3]$ を将来の探索の候補から外してよい．具体的には，残すべき要素だけをスタックを用いて管理すればよいのである．残すべき要素は単調減少列になっていることに注意されたい．**図 -10** はスタックの内容がどのように推移するかを示したものである．たとえば，$A[5] = 54$ に到達したとき，$A[1] = 87 > A[2] = 32 > A[4] = 28$ が $A[1..4]$ の部分に対応する単調減少列である．この単調減少列から要素を順に取り出して，現在の値より大きい要素を見つけると，それが求める直近上位要素である．この場合には $A[4], A[2], A[1]$ の順に取り出し，最後の $A[1]$ で初めて自分より大きい要素を見つけることになる．ここで自分の左の直近上位要素の位置を作業用の配列に蓄えた後，単調減少列を修正する．この例では $A[2] = 32$ と $A[4] = 28$ は $A[5]$ より小さいので，単調減少列から取り除き，最後に $A[5]$ 自身を列に挿入する．これらの操作はスタックを用いると自然に実行できる．同じ操作を逆方向に行えば，自分の右の直近上位要素の位置も求めることができる．

上記のアルゴリズムを擬似言語で記述すると次のようになる．

---

**左にある直近上位要素を求めるアルゴリズム**

**入力**　すべて異なる $n$ 個の要素からなる配列 $A[1..n]$.

**出力**　各要素に対する直近上位要素の位置.

**アルゴリズム**

```
配列 L[1..n] : 自分の左の直近上位要素の位置
スタック : S.
L[1] = −∞, S = (1).
for i = 2 to n do{
  L[i] = −∞.
  while(S が空でない ){
    スタック S からインデックス j をポップする.
    if A[j] > A[j] then j をスタックにプッシュし,
    L[i] = j としてループから出る (break).
  }
  i をスタックにプッシュする.
}
```

---

右にある直近上位要素を求めるアルゴリズムも同様である．最初の走査ですべての要素について左にある直近上位要素を求め，次の走査で右にある直近上位要素を求める．最後に各要素について，左右どちらの直近上位要素が近いかを判断すればよい．

上記のアルゴリズムは 2 重ループの構造をしているので，解析に慣れていない読者はやはり要素数の 2 乗に比例する時間がかかってしまうと誤解することがあるだろう．しかし，実際の計算時間は要素数に比例する程度である．確かにある要素については左の直近上位要素を求めるために多数の要素と比較しなければならないことは起こる．図 -10 の例で，要素 $A[5] = 54$ について調べようとするとき，スタックには (28, 32, 87) が順に蓄えられている．28, 32, 87 の順に調べて，ようやく $A[5]$ 以上の要素を発見する．3 回の比較が必要だったが，28 と 32 は $A[5]$ 以下だったので，$A[6]$ 以降の探索には必要がないということでスタックから外されることに注意しよう．つまり，28 と 32 のために比較が必要になったが，それらはスタックから外されるので，それ以降の探索では比較の対象になることはないのである．このことから，左にある直近上位要素を求めるアルゴリズムが線形時間（要素数に比例する時間）で動作することが分かる．

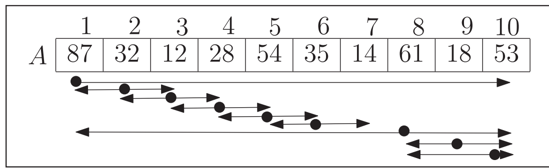


図 -11　自分より大きい最も近い要素を求めるまでの移動距離

**定理 2**　長さ $n$ の配列が与えられたとき，$O(n)$ の作業領域を用いると，その配列に対する直近上位要素発見問題を $O(n)$ 時間で解くことができる．

残念ながら，上記のアルゴリズムは $O(n)$ の作業領域を必要とする．作業用の配列を使わず，定数領域で問題を解くことはできないだろうか？　この問題の場合には実に簡単である．各要素 $A[i]$ において，左右に自分より大きい要素を見つけて，近い方の上位要素のインデックスを配列に蓄えることなく，単に出力するのである．アルゴリズムは次の通り実に単純である．

---

**直近上位要素を求める定数領域アルゴリズム 1**

**入力**　すべて異なる $n$ 個の要素からなる配列 $A[1..n]$.

**出力**　各要素に対する直近上位要素の位置.

**アルゴリズム**

```
for i = 1 to n do{
  L = −∞, R = ∞;
  for d = 1 to i−1 do
    if A[i−d] > A[i] then {L = i−d. break.}
  for d = 1 to n−i do
    if A[i+d] > A[i] then {R = i+d. break.}
    if L = −∞ かつ R=∞ then ∞ を出力.
  if i−L < R − i then L を出力
  else R を出力.
}
```

---

アルゴリズムは簡潔であるが，最悪時の計算時間は $O(n^2)$ である．配列の内容が単調減少の場合などが最悪である．左への探索は早く終わるが，右への探索は常に配列の右端まで到達するので $O(n^2)$ に時間がかかるのである．右への探索を先にしても，今度は単調増加の列が最悪の場合となる．では，左右に 1 歩ずつ探索を進めるという考え方はどうだろう？　プログラム的には下に示すように軽微な変更である．

---

**直近上位要素を求める定数領域アルゴリズム 2**

**入力**　すべて異なる $n$ 個の要素からなる配列 $A[1..n]$.

**出力**　各要素に対する直近上位要素の位置.

**アルゴリズム**

```
for i = 1 to n do{
  N = ∞.  d = 1.
  while( N = ∞ かつ d ≤ max(i − 1, n − i)){
    if i − d ≥ 1 かつ A[i − d] > A[i]
      then {N = i − d. break.}
    if i + ≤ n かつ A[i + d] > A[i]
      then {N = i + d. break.}
    d = d + 1.
  }
  N の値を出力.
}
```

---

単調減少列や単調増加列は先のアルゴリズムに対しては最悪の入力例であったが，今度は左右に 1 歩ずつ調べているので，今度は最善の場合になる．では，どんな場合が最悪だろうか？

最悪の場合を考えるためにアルゴリズムの計算時間を解析してみよう．アルゴリズムでは，各 $i$ について，$i$ からの距離 $d$ を 1 ずつ増やしながら自分より大きな要素を探している．**図 -11** は，自分より大きい要素を見つけるまでに移動した距離を示したものである．$i$ から移動した距離を $d_i$ と表すことにすると，全体の計算時間 $T(n)$ は，

$$T(n) = O\left(\sum_{i=1}^{n} d_i\right)$$

で与えられる．

したがって，$d_1 + d_2 + \cdots + d_n$ の上界が求まればよい．議論を分かりやすくするために，$d_1, \ldots, d_n$ を値の降順に並び替えたものを $d'_1 \geq d'_2 \geq \cdots \geq d'_n$ としよう．

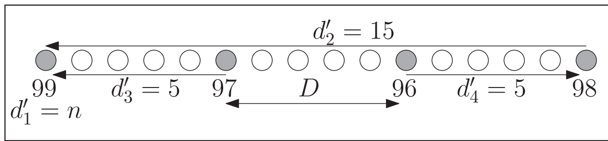

**図 -12　マークが付けられた $k$ 個の要素の配置例**

このとき，$d'_k$ はどれぐらい大きくなり得るだろうか？　それ以上の値は $d'_1, ..., d'_k$ であるが，それぞれの値に配列の要素が対応している．それら $k$ 個の要素にマークをつけたとしよう（**図 -12** 参照）．そうすると，全部で $n$ 個の要素の中に $k$ 個だけマークのついた要素があるが，それらの中で最も近い（つまり，インデックス差が最小の）ペアの距離を $D$ とする．仮定より，すべての要素が異なる値を持つので，このペアの要素のどちらかが他方より大きい．したがって，小さい方の要素から大きい方の要素までの距離 $D$ は，その要素の対応する $d$ の値より小さくなることはない．一方，$d'_k$ はマークが付けられた要素に対応する距離の中での最小値だったから，　$D \geq d'_k$
を得る．

距離 $D$ とは何だっただろうか？　マークづけされた $k$ 個の要素で構成されるペアの中で最も近かったものの距離である．この最小距離が最大になるのは，配列の両端にマークされた要素を置き，その間に $k-2$ 個のマークを等間隔で並べたときである．そのときの間隔は $(n-1)/(k+1)$ である．つまり，

$$\frac{n-1}{k+1} \geq D \geq d'_k$$

を得る．これより，

$$\sum_{k=1}^{n} d'_k \leq \sum_{k=1}^{n} \frac{n-1}{k+1} = nH_n + O(n)$$

が得られる．ここで，$H_n$ は $n$ 次の調和数（harmonic number）として知られているもので，$H_n = O(\log n)$ である．よって，全体の計算時間は

$$T(n) = O\left(\sum_{i=1}^{n} d_i\right) = O\left(\sum_{i=1}^{n} d'_i\right) = O(n \log n)$$

で与えられる．

**定理 3**　長さ $n$ の配列に対する直近上位要素発見問題は，すべての配列要素が異なるなら，$O(1)$ の作業領域を用いて $O(n \log n)$ 時間で解くことができる．

| 12 | 15 | 29 | 33 | 72 | 21 | 33 | 19 |
|---|---|---|---|---|---|---|---|
| 25 | 27 | 39 | 18 | 65 | 43 | 28 | 92 |
| 94 | 65 | 67 | 58 | 49 | 37 | 31 | 20 |
| 23 | 25 | 28 | 36 | 47 | 78 | 21 | 30 |
| 18 | 22 | 32 | 29 | 45 | 52 | 53 | 60 |
| 71 | 65 | 45 | 38 | 28 | 87 | 85 | 83 |
| 43 | 45 | 62 | 89 | 92 | 95 | 36 | 37 |
| 87 | 21 | 27 | 24 | 76 | 87 | 88 | 99 |

**図 -13　2 次元配列上での直近上位要素発見問題．いくつかの要素について，直近上位要素が矢印で示されている．**

ここで用いた技法は双方向探索（bidirectional search）と呼ばれるものである．一方向にだけ探索をするのでは $O(n^2)$ 時間かかるところが，探索の方向を双方向にするだけで $O(n \log n)$ 時間に改善できたのである．

上では明確には述べなかったが，配列の中に同じ値のものが含まれていると上で述べた解析はうまくいかない．極端な場合には，配列の中に 2 種類の値しか含まれないということもある．その場合にはまったく別の考え方でアルゴリズムを設計する必要がある．幸いにも，1 次元配列の場合には，次の定理に示すように効率の良いアルゴリズムが知られている．

**定理 4**　長さ $n$ の 1 次元の配列に $k$ 種類の値が格納されているとき，$O(n \times \min(k, \log n))$ の時間で直近上位要素問題を解く定数領域のアルゴリズムが存在する．

同様の考え方を 2 次元配列上での直近上位要素発見問題に適用することができる．**図 -13** に示したように，全部で $n$ 個の要素からなる $\sqrt{n} \times \sqrt{n}$ のサイズの行列が与えられたとき，各要素でユークリッド距離の意味で最も近い上位要素（自分より大きな要素）を $O(1)$ の作業領域だけで $O(n \log n)$ 時間で求めることができるのである．考え方は，自分に近い要素から順に調べていくという単純なものであるが，少し工夫すると上記の計算複雑度を達成することが

できる．また，極端な場合には2種類の値しか含まれない場合もある．特に0と1の値だけで定義される2値画像の場合，この問題は距離変換として知られている問題であり，線形の作業領域を用いると線形時間で解けることが知られているが，定数領域でも$O(n \log n)$時間で解けることを示したのが上の結果である．

直近上位要素発見問題については，$O(n)$の作業領域を用いると$O(n)$時間で解けるが，$O(1)$の作業領域だけでも$O(n \log n)$時間で解けることが分かった．では，$O(n)$時間で解くためには$\Omega(n)$の作業領域が必要だろうか？　実は，$O(\sqrt{n})$の作業領域だけでも十分であることが分かっている．さらに，階層化の考え方を導入すると，計算時間を$O(n)$に保ったまま，作業領域を$O(\log n)$（つまり，$O(\log^2 n)$ビット）に削減することも可能であるが，議論が少し複雑なので本稿では述べない．

## まとめ

技術の進展に伴ってメモリの価格は一昔前に比べて飛躍的に安価になったが，それに伴って問題サイズも増大しており，いつの時代になってもメモリに関する制約は厳しいものがある．しかし，膨大なメモリに慣れ親しんだ現代のプログラマは，半世紀以上前の研究者が味わった厳しいメモリ制約に驚くばかりで，省メモリのための科学的な方法論のないままに経験と勘のみに頼ってアルゴリズムの設計を行っていることはないだろうか．本稿は，アルゴリズム研究者の視点から，省メモリのためのアルゴリズム設計技法を紹介することを目的とした解説である．

本稿で紹介するアルゴリズム設計技法は，そのほとんどが言われてみるとなぜそんな簡単なことに気がつかなかったのかというような素朴なものであるが，ふんだんにメモリを使ってきた研究者やプログラマにはなかなか思いつかないものではないだろうか．

実は，理論計算機科学の中でも最も難解な計算複雑度理論では対数領域アルゴリズムの名の下に，定数領域アルゴリズムが長年にわたって研究されてきた．ただ，難解なアルゴリズムが多く，実用的な観点からは使い辛いものが多そうである．それと，今回も最後の方で少し触れたが，作業領域を$O(\log n)$ビットに限定するのは，いささか極端すぎるのではないかと思われる．画像への応用などを考えても，$O(\sqrt{n})$程度の作業領域を考えるのが妥当ではないだろうか．次回は，応用に焦点を当てて解説する．

**参考文献**

1）Leavitt, W. G.：A Theorem on Repeating Decimals, American Mathematical Monthly, Vol.74, No.6, pp.669-673 (June-July 1967).

(2011年5月3日受付)

**■浅野 哲夫（正会員）** t-asano@jaist.ac.jp
1977年大阪大学大学院工学博士．大阪電気通信大学教授を経て1997年より北陸先端科学技術大学院大学教授．計算幾何学に関する研究に従事．本会，ACM, 電子情報通信学会各フェロー．