アルゴリズムの開発における尺取法の活用

倪　　永茂

はじめに

AIにおけるディープラーニング（深層学習）やデータサイエンスが盛んにいわれる今日、アルゴリズムの重要性がますます注目されるようになった。アルゴリズムの研究は、コンピュータ・サイエンスのまさに核心にある（A.V. エイホら、1977）。本文でいうアルゴリズムとは、チューリングマシンによって記述できる手続きのことをいう。文献として残されている最初のアルゴリズムは紀元前3世紀頃にエジプトでつくられた『ユークリッドの原論』に見出すことができるが、数学から独立して本格的に研究開発するのはまだ100年未満の歴史しかなく、新しい学問といってよい。

本文はアルゴリズムのうち、尺取法（英語 Two Pointer Algorithm）について考察するものである。尺取法は従来のアルゴリズム研究ではそれほど重要視される対象ではなく、アルゴリズムに関する専門書のなかでも言及されることはあまりなかった。しかし、アルゴリズムの優劣を決めるうえで重要な計算時間において、そのオーダーを下げることに尺取法が有効であることは近年明らかになってきている。

以下では尺取法の基本としてその概説と特徴、および尺取法を採用したアルゴリズムの実例を示したうえで、尺取法の応用例等を考察していく。なお、本文で扱う数値はすべて整数（整数にエンコードできる文字等のデジタル量を含む）である。

Ⅰ 尺取法の基本

ここでは、尺取法とはなにかをまず説明したうえで、その特徴について考察する。

1 尺取法とは

図1のように、一次元の配列に対して、配列の位置を示すポインタを2つ用意し、それぞれを、たとえば、*left* と *right* と名付けしよう。

配列は各要素が横に一列に並ぶものとイメージする。最初に、*left* と *right* の初期値として、ともに配列の左端（または右端）、あるいは、*left* が左端、*right* が右端に指すようにする。

2つのポインタのうち、*right* が *left* よりも左側にならないよう、片方を先行させる。つまり、*left* と *right* がともに配列の左端であれば、*right* を先行させる。*left* と *right* がともに配列の右端であれば、*left* を先行させる。*left* と *right* が配列の両端を指すのであれば、どちらかを先行させても構わないが、途中にぶつかったらストップする。

ある条件を満たしていれば、先行のポインタをそのまま先行させる。その条件を満たさなくなれば、もう片方の後行ポインタを先行ポインタに近づくように動かす。こうすることによって、条件が満たされる部分配列のありか（そしてその長さ）が分かるわけである。

配列のどの要素も高々2回だけポインタが指すので、配列全体として、長さを n とすれば、$O(n)$ 個の要素の走査になる。条件の確認が部分配列の長さに関わらず $O(1)$ で処理できるなら、全体の処理時間は $O(n)$ になる。

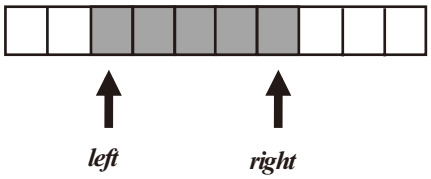

図1　尺取法のイメージ

2 尺取法の実例とその特徴

以下では尺取法を利用した、代表的なアルゴリズムを2つ例示する。

2.1 unique 関数

Microsoft 社の表計算ソフト Excel 2016 には、指定した範囲内において重複値（要素）を取り除く関

数として、unique 関数がある。その実装は尺取法を利用するとつぎのようになる。

与えられる数列を *a* とし、その長さを *n* とする。

```
int unique(int *a, int n) {
 if (n == 0) return 0;
 //(1) 数列 a を値の大きさ順にソートする
 int left = 0, right = 0; //(2)
 while (right < n) {
  while (right < n &&
                    a[left] == a[right])
    ++right; //(3)
  if (right < n)
    a[++left] = a[right++]; //(4)
 }
 n = left+1;
 //(5) 数列 a を与えられた順にソートする
 return n;
}
```

関数の引数は数列を格納する配列 *a* と数列の長さ *n* であり、関数の戻り値は重複値を取り除いた後に残されている数列の長さ（つまり、ユニーク値の個数）である。関数のなかでは、ユニーク値を順次、関数の引数である配列 *a* に戻しているので、*a* の中身は関数の処理によって変わることに注意しよう。たとえば、*n* = 6、*a* = (4, 0, 4, -3, -3, 0) は関数の処理によって、関数の戻り値が 3、*a* = (4, 0, -3) となる。

さて、関数各行末の一部にコメントとして番号をつけた。それらの番号に沿って説明する。

(1) 数列の中身を値の昇順（あるいは降順）にソートする。時間計算量はソートアルゴリズムのそれにしたがう。ソートすることの目的は隣り合う要素の比較によって重複値であるかどうかがわかるようにするためである。

(2) 2 つのポインタを初期化する。

(3) *right* ポインタは先行するが、*left* ポインタの指す要素と重複（値が一致すること）でなくなると止まる。

(4) *right* ポインタの指している要素を *left* ポインタの指している要素のつぎのところに書き込む。*right* ポインタの指している要素が上記 (3) によって、*left* の指している要素と異なることが保証されている。したがって、(3) と (4) をカバーする繰り返しによって、異なる要素が過不足なく *left* ポインタの指している位置に順次書き込まれる。また、この繰り返しでは、*left* が値の異なる要素を 1 回だけ走査し、*right* が各要素を 1 回だけ走査するので、時間計算量が $O(n)$ である。

(5) 重複した要素が取り除かれた数列を与えられた順に並ばせるために、もう一度ソートする。

関数全体の時間計算量はソートアルゴリズムによって左右され、クイックソートなら $O(n \log n)$ となり、より効率的な $O(n)$ ソートを採用すれば $O(n)$ に低減する。

考察として、配列の要素を値の大きさ順でソートしておくこと、値の異なる要素を *left* ポインタに書き込んで集めること、*right* ポインタと *left* ポインタのところの要素を比較だけで異なる要素を過不足なく調べられることが特徴だと言えよう。

本関数の機能を実現するもうひとつの方法として、ハッシュ法（倪、2020）を利用することがあげられる。つまり、与えられた数列を順にその要素をハッシュテーブルに登録し、重複がなければユニーク値として出力すればよい。時間計算量が $O(n)$ である。

2.2 回文半径を計算する Manacher 法

回文とは、与えられた文字列を始めから読んだ場合と、終わりから通常と逆の方向で読んだ場合に、全く同じになる文字列のことで、英語では palindrome という。なお定義によっては、文字列において大小文字の区別を無視したり、単語間のスペースを無視することも許される。英語でいうと、madam、level、racecar 等の単語がその例である。

文字列 *s* の *i* 番目要素（配列最初の添字は 0）s_i を中心とする連続する部分文字列が、最大で左右何文字まで回文かという値を回文半径という。

文字列 *s* の長さを奇数長 *n* とすると、Manacher 法 (Manacher, 1975) は全ての s_i に対して回文半径を効率よく計算し、全体の時間計算量が $O(n)$ であることとして知られている。

以下ではそのアイデアを尺取法に適したものに書き直し、C 言語プログラムで示す。なお、回文半径は、数列 *r* に格納される。

```
void manacher(int *r, char *s, int n) {
 int i = 0, j = 0;
  while (i < n) {
   while (j <= i && i+j < n &&
             s[i-j] == s[i+j]) ++j; //(1)
   r[i] = j; //(2)
```

```
    int k = 1;
    while (k <= i && i+k < n && //(3)
      k+r[i-k] < j) r[i+k] = r[i-k], ++k;
    i += k, j -= k; //(4)
  }
}
```

関数の引数は 3 つ、回文半径を格納する数列 *r*、文字列 *s*、文字列 *s* の長さ *n* である。関数は戻り値がないが、数列 *r* にそれぞれの位置の回文半径が格納されている。たとえば、*n* = 19、*s* = wasitacaroracatisaw とすると、*r* = (1, 1, 1, 1, 1, 1, 2, 1, 1, 10, 1, 1, 2, 1, 1, 1, 1, 1, 1) が関数の処理によって得られる。$r_7 = 2$ は aca に対応した半径であり、$r_{10} = 10$ は文字列 *s* 全体が回文であり、中央が 10 番目の文字というところにあることに対応している。

では、関数についてみていこう。関数のなかでは、変数 *i* は *right* ポインタ、*j* と *k* は *left* ポインタの役割を果たす。*left* ポインタを 2 つ使っていることが特徴的であろう。各行末の一部につけた番号に沿って説明する。

(1) 変数 *i* を中心にその左右の文字 (*i* - *j*, *i* + *j*) が同じであるかどうかを調べ、同じ文字であれば半径を広げていく。

(2) 回文半径を格納する数列 *r* に回文半径 *j* を書き込む。

(3) 回文半径以内の数列 *r* は変数 *i* を中心に左右対称なので、r_{i+k} に r_{i-k} の値をそのまま書き込む。

(4) すでに調べたところを飛ばして、変数 *i* と *j* を設定しなおす。

関数の時間計算量について、長さ *n* の文字列 *s* における各文字に対して、変数 *i, j, k* は高々 1 回しか走査しないので *O*(*n*) となる。また、関数のなかで新たに必要とするメモリは 3 つの変数 *i, j, k* を格納するためのものだけであるので、空間計算量は *O*(1) となる。

なお、Manacher 法は奇数長の文字列にしか正しく計算できないが、偶数長の文字列に対しては、文字列に現れないダミ文字を文字の間に挟み、奇数長にすることがよく知られている技法のひとつである。たとえば、文字列 abba に対して、ダミ文字 _（アンダーバー）を挿入して a_b_b_a にすれば、長さ 7 に変わる。対応する回文半径は (1, 1, 2, 4, 2, 1, 1) と算出される。

Ⅱ 尺取法の応用

上記の 2 例をもとに、尺取法を適用するための条件である部分数列の特性を整理し、時間計算量を下げる尺取法の応用について考察する。

1 部分数列の特性

left ポインタが先頭位置、*right* ポインタが後尾位置として指している部分数列の性質を調べることの効率性が尺取法の時間計算量に大きく影響する。

・単調性

部分数列の要素数 (= 長さ) や、各要素の値が非負 (ゼロか正) である数列の累積和等は単調性を有する。その単調性を利用すれば、効率の向上が期待できる。

たとえば、与えられた数列において、連続した部分数列の累積和が最大となる値はいくつか。累積和が指定した値以上 (または以下) となる部分数列の最短長 (または最大長) はどれくらいか。数え上げ問題として、累積和が指定した値以上 (または以下) となる、異なる部分列の数は最大いくつあるか、等などの応用問題が考えられる。これらのいずれも、尺取法を使えば時間計算量を *O*(*n*) にすることができる。さらに、前述した unique 関数や、本文の後半で説明する kth_pair 関数では尺取法を適用するための前処理として、数列をソートしておくことも単調性をもたせるためである。

・可算性

部分数列のある性質を 1 回の走査で数えられることをここでは加算性という。その可算性を利用すれば、時間計算量の低減を図ることができる。

たとえば、指定した 1 つあるいは複数の値をそれぞれ指定した個数を含む部分数列の位置や最短部分数列の長さはどれくらいか。同じ値 (あるいは同じ文字、または指定した文字列のパターン) がつづく部分数列の最大長はなにか、等を求める問題が相当する。これらも、尺取法を適用すれば計算の効率が改善される。

2 応用例 ― 水溜問題

道路の幅が一定の路面は舗装が悪く、あちこちに凸凹になっていて、雨が降ると水が溜まってしまう。さて、溜まった雨水の最大量はどれくらいか。

この問題を単純化して、つぎのようにモデル化する。幅が 1 単位の道路において、路面に沿って 1 単位ごとにその高さが数列 h ($h_i \geqq 0$) によって与えられ、道路の長さが n 単位である。なお、道路の両端に深い溝ができており、溝には水が溜まらないものとする。n の範囲は $0 \sim 10^9$、h_i の値は $0 \sim 10^9$ ($0 \leqq i < n$) とする。

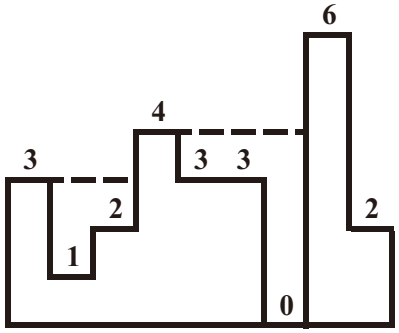

図２　水溜問題

例として、h = (3, 1, 2, 4, 3, 3, 0, 6, 2)、n = 9 が与えられた場合のグラフを図 2 に示す。溜まる雨水の最大量は 9 であり、点線はそのときの雨水の高さである。

h に関する部分数列の加算性を利用して、尺取法を適用するとつぎのプログラムが得られる。なお、関数中の Int は整数 int 同士の積で得られる値の大きさを格納できる疑似データ型であり、C 言語なら long long int としてよい。

```
Int pool(int *h, int n) {
  Int ans = 0; int ma = n; //(1)
  int left = 0, right = 1; //(2)
  while (right < n) {
    Int s = 0; //(3)
    while (right < n && //(4)
      h[left] > h[right])
      s += h[right++]; //(5)
    if (right == n) { //(6)
      ma = left; break;
    }
    ans += (Int)(right-left-1)*
           h[left]-s; //(7)
    left = right++; //(8)
  }
  left = n-2, right = n-1;//(2)
  while (ma <= left) {
    Int s = 0; //(3)
    while (h[left] < h[right])//(4)
      s += h[left--]; //(5)
    if (ma <= left)
      ans += (Int)(right-left-1)*
             h[right]-s; //(7)
    right = left--; //(8)
  }
  return ans;
}
```

プログラムは引数が数列 h とその長さ n の 2 つであり、戻り値が溜まった雨水の最大量である。

プログラムのアイデアは、*left* ポインタの指す路面と *right* ポインタの指す路面の高さの違いに注目することである。すなわち、道路の右端に向かって両ポインタを動かしていき、*right* の高さが同じかより高ければ、両ポインタ間に溜まる雨水を計算する。

路面の最も高いところが道路の右端でない場面に備え、プログラムの後半では、道路の右端から左に向かって両ポインタを動かす。*left* の高さが同じかより高ければ、両ポインタ間に溜まる雨水を計算する。

各行末の一部につけた番号に沿って説明する。

(1) *ans* は雨水の最大量を格納するための変数であり、*ma* は最も高い路面に対応する配列の添字である。*ma* の初期値は n とし、道路の最も高い箇所が不明の状態だとする。

(2) 尺取法における 2 つのポインタ *left* と *right* を初期化する。同じところでは路面の高さが同じなので、初期化の値は *right* = *left* + 1 としている。

(3) 変数 *s* は路面の高さの累積和を格納する。

(4) プログラムの前半では、*right* の指す路面の高さが *left* のそれよりも低ければ、*right* が右側に向けて移動していく。プログラムの後半では、逆に *left*t の指す路面の高さが *right* のそれよりも低ければ、*left* が左側に向けて移動していく。

(5) 路面の高さの累積和を取る。

(6) 道路の最も高いところが道路の左端か道路の途中である場合は、*ma* にその添字を記録しておく。

(7) 上記 (4) のループが終了した時点で、*right* の指す路面と *left* の指す路面との間に溜まる雨水を *ans* に加算していく。

(8) *left* ポインタと *right* ポインタを更新する。*left*（または *right*）をそのときの *right*（または *left*）の値で

更新することに留意しよう。

プログラムの時間計算量は 2 つのポインタ *left* と *right* の動きによってそれぞれが数列 *h* を高々 2 回走査するので $O(n)$ となる。このように、*left* ポインタと *right* ポインタは最大値を見つけながら往復して最大 2 回走査することが本プログラムの特徴だといえよう。

プログラムのなかではいくつかの変数以外に余分なメモリを必要としないことに留意しよう。つまり、空間計算量に関しては $O(1)$ である。

本プログラムに関してはスタックを活用する時間計算量 $O(n)$ のプログラムを作成することもできるが、スペースのため割愛する。ただ、スタックを使うと空間計算量は $O(n)$ になり、尺取法に比べて不利になる。

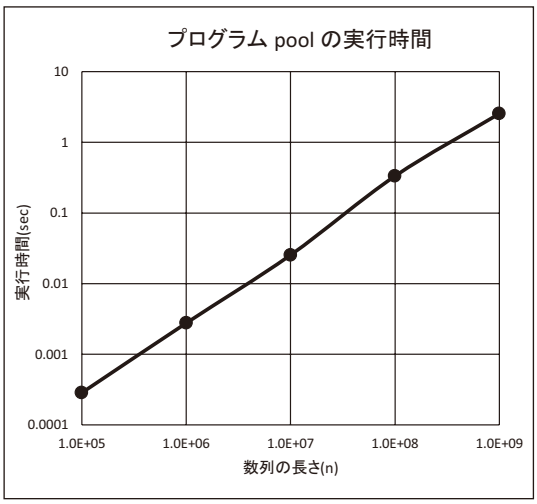

図 3 プログラム pool の実行時間

さて、プログラム pool の時間計算量を検証するために、数列の長さ *n* と実行時間との関係を調べた。*n* を 10 万、100 万、1 千万、1 億、10 億の 5 通り、数列の各要素として 0 ～ 10^9 の疑似乱数を Xorshift 法（Marsaglia、2003）によって作成して使った。

検証結果は図 3 である。横軸も縦軸も対数（双対数グラフ）にし、各ケースの実行時間を結んだらほぼ直線になっている。実行時間を 1 秒下回ったのは数列の長さが約３億までの範囲である。なお、プログラムを動かしたのは、CPU Intel Core i7 2.6 GHz 6 コア、メモリ 16 GB というスペックを有する MacBook Pro (16-inch, 2009) であった。実行時間は OS やハードウェアによって変わるが、数列の長さ *n* と実行時間との比例関係が図 3 によって確認できたことが重要であろう。

3 時間計算量 $O(1)$、$O(\log n)$ のアルゴリズムとのコンビネーション

適用した尺取法の時間計算量が $O(n)$ であれば、さらに時間計算量が $O(1)$ や $O(\log n)$ であるアルゴリズムを適用しても、全体の時間計算量が $O(1)$ や $O(n \log n)$ に抑えられるので、効率のよいアルゴリズムとして認められる。

時間計算量が $O(1)$ のアルゴリズムはハッシュ法等であり、時間計算量が $O(\log n)$ であるアルゴリズムは二分探索法（バイナリサーチ）、二分木（バイナリツリー）、AVL 木、赤黒木、BIT 木（Binary Indexed Tree）、高速フーリエ変換、素集合データ構造に対する操作（Union-Find アルゴリズム）等、数多く存在する。それらとのコンビネーションが尺取法を応用するうえでヒントになるのではないか。

以下では尺取法とのコンビネーションによる実例を見てみよう。

3.1 値がすべて異なる最も長い連続部分配列 の長さ： longest_unique_subarray 関数

与えられた配列について、以下の条件をみたす連続した部分配列のうち、最長のものの長さをもとめよ。

条件：連続した部分配列の要素の値はどれもユニーク、つまり他の要素の値と重複しない。

例示すると、配列 (2, 3, 5, 6, 2, 7, 3) について、答えは 5 である。それが (3, 5, 6, 2, 7) に対応するからである。

スペースの都合で詳細は割愛するが、ハッシュ法（倪、2020）を使えば、時間計算量が $O(n)$ となる解答プログラムを作成できる。つまり、*left* と *right* ポインタを配列の先頭におき、*right* ポインタを先行させながら、指している要素とその添字をハッシュテーブルに登録する。重複登録となれば、前回登録した際の添字と現在の添字直前の間にある部分配列の長さが解答の候補になる。そして、重複登録では、前回の添字を今回の添字で更新し、*left* ポインタを動かす。*right* ポインタが配列の末尾にくれば、解答の候補から最長のものを選べばよい。

ハッシュテーブルへの登録や更新は $O(1)$ でできるので、全体の時間計算量は $O(n)$ となる。なお、解答の候補についてはその長さをそれまでの最大値

（初期値が 0）と比較し、超えていれば最大値を更新すればよい。

3.2 *k* 番目に小さいペアの積： kth_pair 関数

各要素の値が正である数列が与えられ、数列の任意の 2 つの要素の積で作られたペアの積のうち、*k* 番目に小さい値を求めよ、という問題について考えよう。

形式的に記述すると、与えられる数列を a とし、その長さを *n* とするなら、$a_i \times a_j$ $(1 \leqq i, j \leqq n)$ というペアの積を昇順に並べたときに、*k* 番目に小さいペアの値の大きさは何か。なお、ペア数は n^2 個あるので、*k* は 1 ～ n^2 の範囲内であることが前提条件になる。たとえば、*a* = (2, 1, 3) なら、ペアの積でつくられて数列は (4, 2, 6, 2, 1, 3, 6, 3, 9) であり、ソートすると (1, 2, 2, 3, 3, 4, 6, 6, 9) が得られ、それぞれが 1 ～ 9 番目に小さい値となる。

k 番目に小さい値を算出する効率的アルゴリズムはいくつか知られている（倪、2019）。しかし、本問題はペアの積を取るだけで、$O(n^2)$ になってしまう。そこで、時間計算量を改善するために、尺取法を適用してみる。

```
Int pairs(int *a, int n, Int g) {
  Int s = 0; //(1)
  int left = 0, right = n-1; //(2)
  while (left <= right) {
    while (left <= right &&  //(3)
  (Int)a[left]*a[right] > g) --right;
    if (left <= right)
      s += 2*(right-left)+1; //(4)
    ++left; //(5)
  }
  return s;
}
```

関数 pairs の引数は 3 つあり、要素の値が昇順にソートされた数列を格納する配列 *a* とその数列の長さ *n*、そしてペアの積の値に関する上限 *g* である。関数の目的は *a* のなかの、ペアの積の値が *g* 以下であるペアの個数を算出することであり、関数の戻り値はこれらのペア数である。

各行末の一部につけた番号に沿って説明する。

(1) 変数 *s* はペアの積が *g* 以下であるペアの個数を格納するものである。

(2) 尺取法における 2 つのポインタを初期化する。*left* は数列の左端、*right* は数列の右端を指すようにしてある。

(3) *right* ポインタは先行するが、*left* ポインタが指す要素との積が *g* 以下になるよう、*right* ポインタを左側に動かしていく。

(4) *left* ポインタから *right* ポインタまでの部分数列は、つぎに説明する関数 kth_pair のなかのソートによって得られた単調増加数列になっているので、その部分数列のなかのどの要素も *left* ポインタの指している要素との積が g 以下の値になる。なので、これら $2(right - left) + 1$ 個のペアを変数 *s* に加算する。

(5) *left* ポイントを右側に動かす。

関数の時間計算量は前述したとおり $O(n)$ である。

関数 pairs を使うのはつぎの関数 kth_pair である。

```
Int kth_pair(int *a, int n, Int k) {
  // 数列 a を値の昇順にソートする
  Int l = (Int)a[0]*a[0]-1;
  Int r = (Int)a[n-1]*a[n-1];
  while (l+1 < r) {
    Int m = (l+r)/2;
    if (pairs(a, n, m) >= k) r = m;
    else l = m;
  }
  return r;
}
```

関数の引数は 3 つあり、数列を格納する配列 *a* と数列の長さ *n*、そして本問題が指定した *k* 番目である。関数の戻り値は *k* 番目に小さいペアの積の値である。

関数のなかでは二分探索法を活用して、初期値として、最小の値 *l* は a_0 の 2 乗 -1、最大の値 *r* は a_{n-1} の 2 乗とする。その中央の値 *m* が上記の関数 pairs に使われる。それによって *m* を中心とした左半分または右半分を切り捨てて調べる範囲を半分に縮小していき、最終的に範囲が 1 になったところで終了する。

さて、二分探索法の時間計算量はよく知られているように $O(\log N)$ であり、関数 pairs と合わせて時間計算量が $O(n \log N)$ になる。ただし、*N* は数列の最大値の 2 乗と最小値の 2 乗との差を示す。加えて、関数 kth_pair の最初に示されたソートアルゴリズムも時間計算量は $O(n \log n)$ が一般的である。

$N \geqq n$ であれば、関数全体の時間計算量は $O(n \log N)$ となる。

本関数における尺取法の効率は、ポインタ *left* と *right* との間にあるペア数は $O(1)$ で算出できることによる。

関数 kth_pair の時間計算量を検証するために、数列の長さ n と実行時間との関係を調べた。n を 10 万、100 万、1 千万、1 億、10 億の 5 通り、k を 1、$n^2/2$、n^2 の 3 つとし、数列の各要素を Xorshift 法によって 1 ～ 10^9 の値を疑似乱数的に発生して得ることとした。数列をソートした後では、$k = 1, n^2$ 時の答えは確かに自明ではあるが、実験では特例計算せず、他の k と同様の手続きで計算を行った。実行時間は誤差を減らすため、各ケース（図 4 でのマーカーのところ）に対して計測プログラムを 10 回実行し、その平均を取った。

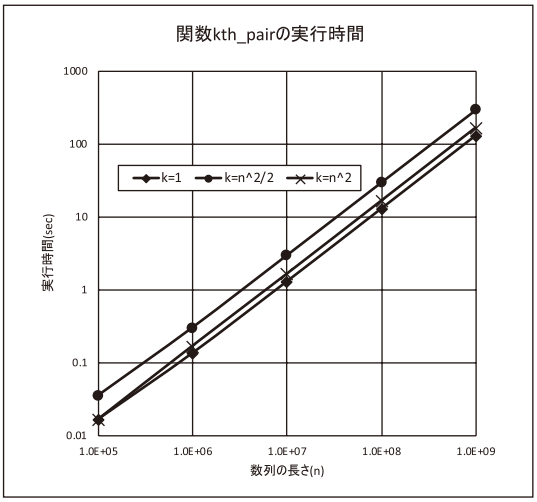

図 4　関数 kth_pair の時間計算量に関する検証実験

検証結果は図 4 である。横軸も縦軸も対数にし、各ケースの実行時間を k の値別に結んだらほぼ直線になっている。3 本のうち、実行時間の短い順で $k = 1, n^2, n^2/2$ である。実行時間が 1 秒を下回るのは数列の長さが 300 万前後までであろう。

ほぼ直線である理由はつぎのように考える。数列の作り方により、その要素の最大値が 10^9 程度、最小値が 1 程度で、しかも k と n とはほぼ無関係である。したがって本検証実験では、時間計算量 $O(n \log N)$ において $\log N$ がほぼ定数になり、n の値に比例した時間計算量（実行時間）になるわけである。なお、$k = 1, n^2, n^2/2$ の違いによる実行時間の差は尺取法における両ポインタの動きが違うことによるものだと考えられる。

本問題において、ペアの計算方法についてのバリエーションがいくつか考えられるが、その一部は関数 pairs におけるペア数の計算式 (4) を変えることで対処できることを指摘しておく。

終わりに

尺取法は最近の競技プログラミングコンテストブームで注目されるようになった理由は時間計算量を低減する手法として有効であること、応用範囲が広いこと等である。しかし、尺取法は従来のアルゴリズム関連の専門書に取り上げられることはあまりなかった。

本文は尺取法に関する基本的な使い方を説明したうえで部分数列の特性を整理し、尺取法の応用について言及した。数多く考案され、時間計算量が $O(\log n)$ であるアルゴリズムとのコンビネーションに尺取法のさらなる応用を期待する。

参考文献

エイホ , A. V.、ホップクロフト , J. E.、ウルマン , J. D. R（野崎昭弘他訳）（1977）、『アルゴリズムの設計と解析Ⅰ』サイエンス社

倪 永茂（2019）、「n 番目に小さい値を求めるアルゴリズムについて」宇都宮大学国際学部研究論集 (48)、87-94

倪 永茂（2020）、「ハッシュ法の実装と応用」宇都宮大学国際学部研究論集 (49)、123-129

Manacher, Glenn K. (1975), *A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String*, J. ACM 22(3), pp. 346-351.

Marsaglia, George (2003), *Xorshift RNGs*, Journal of Statistical Software Vol. 8 (14) pp. 1-6.

Utilization of Two Pointer Method in Algorithm Development

NI Yongmao

Abstract

The reason why two pointer method has attracted attention in the recent boom of competition programming contests is that it is effective as a method to reduce the amount of time computation and has a wide range of applications. However, it was not often mentioned in traditional algorithmic-related text.

In this article, we first outlined and described the features of two pointer method, and then gave some algorithms that adopt this method. We explained the basic usage of two pointer method, organizes the properties of the subsequence, and mentions it's applications. It is expected further that it's applications will be made in combination with algorithms whose time complexity is $O(\log n)$.

（2020 年 5 月 25 日受理）