アルゴリズム研究シリーズ － In-Place 法

倪　　永茂

はじめに

コンピュータ・プログラムの本質はアルゴリズムであり、アルゴリズムの良さを表す尺度として、つぎの 2 つが一般的使われる。

(1) 時間計算量

アルゴリズムがどれくらいの時間オーダーで処理が終了するかを見る尺度であり、入力データのサイズと時間オーダーとの関係で示される。ユーザーの実感として、プログラムがすぐに終了するかどうかということである。プログラムが停止しないことはないとわかったケースでも、数年も数日も待つのは無論のこと、数時間でも待てないのが人情というものである。CPU の性能がムーアの法則に沿って 18 ヶ月で 2 倍向上すると一昔いわれていたが、時間計算量が指数のアルゴリズムなら、現存のどんなに速いコンピュータを使ったとしても、入力データのサイズが数百万や数千万といった実用的な問題にはまったく手に負えない。

(2) 空間計算量

アルゴリズムがどれくらいのメモリ領域を使うかを見る尺度であり、入力データのサイズと空間オーダーとの関係で示される。ユーザーの実感として、コンピュータに搭載したメモリ（RAM）が少ない場合に、プログラムが正常に動くかどうかということである。しかし、今日のパソコンでは、容量が 4 ～ 32 GB のメモリが搭載されることが一般的である。入力データのサイズに換算すると、メモリ不足よりも時間計算量のほうが圧倒的に問題になるので、空間計算量を議論することは現実的な必要性はほとんどないと個人的に考える。それでも、空間計算量を意識することで新しいアルゴリズムの考案や、メモリが多く搭載できない工業製品、特殊のハードウェアには有効であろう。

注意しておきたいのは、時間計算量と空間計算量とはアルゴリズムによってトレードオフの関係にある。メモリを大量に使うことで実行時間が短くなったり、あるいは、実行時間を遅くすることで、メモリの使用量を削減したりすることがある。

アルゴリズムの優劣を判断する物差しとして、ほかに長さや可読性（理解のしやすさ）等をあげることができるが、定量的に議論することは難しい。

本文はアルゴリズム研究シリーズの第 1 弾として In-Place 法を取り上げる。In-Place 法は現実的な必要性というよりも、好奇心や学問としての側面で研究されている。

Ⅰ In-Place 法

In-place 法とは、コンピュータ・サイエンスにおいて、追加のメモリ領域をほとんど必要としないアルゴリズムの総称である。言い換えると、空間計算量 $O(1)$ であるアルゴリズムの総称を本文で意味する。

実例として、整数を格納する 2 つの変数 a、b の値の交換（swap 操作）について考える。一般的には、3 つめの一時変数 t を用意して、

$t = a$、$a = b$、$b = t$

という 3 つの操作を順に行い、a、b の値を交換する。

ところが、メモリの使用が極限までに制限され、3 つめの変数が使えないとなると、たとえば、排他的論理和（XOR）の性質を活用して、

$a = a$ XOR b、$b = a$ XOR b、$a = a$ XOR b

という 3 つの操作を順に行えば同様に a、b の値が交換される。

このように、メモリを極限的に利用制限することによって、アイデアや斬新なアルゴリズムが生まれる可能性があり、それが In-Place 法の狙いでもある。

Ⅱ In-Place 法の活用

ここでは In-Place 法の使い方について、例示しながら、考察して行く。

1 In-Place 法と配列

空間計算量の定義から、同種類の入力データを大量に処理することが必要になってくる。ここでは、基本データ構造のひとつである配列に関する問題を取り上げ、In-Place 法の特徴をみていくことにする。

配列について、本文では長さ n の配列 a に対し、要素を $a_0, a_1, ..., a_{n-1}$ という順に、それぞれの添字を 0, 1, ..., n-1 で記述する。

1.1 Reverse ― 与えられた数列に対し、その数列を逆順に並べ替える数列を求めよ

形式に記述すると、入力として、整数の数列を a ($-10^9 \leqq a_i \leqq 10^9$)、その長さを n ($0 \leqq n \leqq 10^9$) とすると、a_j ($j = n$-1, n-2, ..., 1, 0) という順である配列をつくって出力せよ、という問題である。

たとえば、入力 (-2, 3, 1, 0) に対して、出力は (0, 1, 3, -2) である。

a と違う配列 b を用意して、a の個々の要素を逆順に b に代入するという解答が考えられるが、b を使わないという制限では、以下の解答が一般的だと思われる。

```
void reverse(int *a, int n) {
  int i, j, t;
  i = 0, j = n-1;
  while (i < j) {
    t = a[i], a[i] = a[j], a[j] = t;
    i++, j--;
  }
}
```

プログラムの引数は入力の整数数列 a とその長さ n であり、プログラムの戻り値はないが、数列 a が求めたい数列になっている。

プログラムの中身は単純なので、説明について割愛するが、明らかにプログラムの時間計算量が $O(n)$、空間計算量が $O(1)$ となっていて、In-Place という条件を満たしている。

本プログラムは実用性が高く、たとえば、言語 Python においてリスト型のメソッド reverse( ) が用意されている。

1.2 Unique ― 与えられた整数の数列に対し、値の異なる要素を求めよ

形式に記述すると、入力として、整数の数列を a ($-10^9 \leqq a_i \leqq 10^9$)、その長さを n ($0 \leqq n \leqq 10^9$) とすると、$a_i \neq a_j$ ($0 \leqq i, j < n$) となる a_i ($0 \leqq i < n$) を出力せよ、という問題である。

例示すると、入力 a = (-2, 3, 1, 3) に対して、出力は (-2, 1, 3) になる。

では、プログラムをみていこう。

```
int unique(int *a, int n) {
  if (n == 0) return 0;
  heapSort(a, n); // ヒープソートを適用
  int i = 0, j = 0;
  while (j < n) {
    while (j < n && a[i] == a[j]) ++j;
    if (j < n) a[++i] = a[j++];
  }
  return i+1;
}
```

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 = (-3, 0, 4) となる。

スペースの都合でプログラムの説明を割愛するが、ヒープソートの空間計算量が後述するように $O(1)$ であり、さらに、プログラムのなかで使った変数は i、j の 2 つだけなので、本プログラムの空間計算量が $O(1)$ となる。また、時間計算量について、ヒープソート以外では $O(n)$ であることは容易に示せるので、本プログラムの時間計算量がヒープソートにしたがい、$O(n \log n)$ である。

処理後の配列 a に格納された数列が昇順になっていて、入力順でないことに注意しよう。空間計算量が $O(1)$ に制限される条件下で入力順に戻すことは難しい。

1.3 FirstMissingPositive ― 与えられた整数の数列に対し、数列にない最小の正整数を求めよ

形式に記述すると、入力として、整数の数列を a ($-10^9 \leqq a_i \leqq 10^9$)、その長さを n ($0 \leqq n \leqq 10^9$) とすると、$a_j > 0$、かつ $aj = \mathrm{Min}_{j=0}^{n-1} a_i$ という条件を満たす a_j ($0 \leqq j < n$) を出力せよ、という問題

である。ただし、Min は最小値関数を指す。

たとえば、入力 a = (-2, 3, 1, 0) に対して、出力は 2 である。なぜなら、最小の正整数 1 が a に含まれているが、そのつぎに小さい正整数 2 が含まれていないからである。

問題自体は In-Place 法を想定してつくられたものかもしれないが、ひとつの解き方として、長さ m（$m = 10^9 + 1$）である整数の配列 b を別途用意したうえ、入力の配列 a の正整数である各要素 a_i（$a_i > 0$）に対応して、b_k（$k = a_i$）にフラグを立てる。この処理は時間 $O(n)$ で終わる。そして、配列 b の要素 b_1 から走査し、フラグの立っていないインデックスを見つければ、そのインデックスが解答になる。走査にかかる時間は最大 $O(n)$ であるので、全体の時間計算量は $O(n)$ であり、空間計算量は $O(m)$ である。

では、In-Place 法で解くには、つまり空間計算量を $O(1)$ にするにはどうすればよいか。つぎのプログラムで確認しよう。

```
int firstMissingPositive(int *a,int n) {
  int i, t; //(1)
  for (i = 0; i < n; ++i) {
    if (i == a[i]) continue; //(2)
    while (1) {
      t = a[i]; //(3)
      if (t >= n || t < 0 || //(4)
          t == i || t == a[t]) break;
      a[i] = a[t], a[t] = t; //(5)
    }
  }
  for (i = 1; i < n; ++i) {
    if (i != a[i]) break; //(6)
  }
  if (i == n) i = n+(a[0]==n);//(6)
  return i;
}
```

プログラムの引数は入力の整数数列 a とその長さ n であり、プログラムの戻り値は数列にない最小の正整数である。

プログラムの真髄は入力配列をうまく流用し、要素の値が i のものを添字 i のところに格納すること、つまり、$a_i = i$ になるように処理されることである。ただし、a_0 だけは例外である。a_0 に 0 だけでなく、n や他の値が格納されることもありうる。

では、プログラムの一部の行末につけた番号に沿って説明する。

(1) プログラムのなかで使われる変数は 2 つのみである。

(2) 添字 i の要素はその値が i、つまり $a_i = i$ であれば、正しい格納になっているので、さらなる処理はしない。

(3) 添字 i の要素の値を変数 t に代入し、つぎの (4) と (5) を処理する。

(4) つぎの (5) の処理のための条件を確認する。つまり、変数 t の値が n 以上や負、あるいは、$t = i$（ループに陥るケース）、$a_t = t$（正しい格納）の場合には、繰り返し処理を中止する。

(5) 添字 i の要素に添字 t の値を代入し、変数 t の値を添え字 t に代入し、$a_t = t$ にする。

(6) 添字 1 から走査し、値が a_i でないものがあれば、その添字 i を戻り値とする。配列にそれぞれ 1 ～ n-1 の値がすべて格納していれば、a_0 の値が n であるかどうかを調べる。

空間計算量は上記の (1) により、$O(1)$ であることは明らかであろう。時間計算量については、(3) での 1 つの t について、(5) での 1 回めの代入で $a_t = t$ にすることができ、それ以降の繰り返し処理でも、変数 t の値は変わるが、1 回の代入で $a_t = t$ にすることに変わりがない。したがって、全体の時間計算量は $O(n)$ となる。

上記 3 つの問題において、いずれも入力である配列が流用されることが大きな特徴といえよう。プログラム開発側の方針として、入力データの書き換えを厳禁するというのもあるが、In-Place 法では、余分のメモリ領域を使わないことを最優先にするので、入力配列の書き換えはやむをえない。したがって、In-Place を実現するヒントとして、入力配列をうまく使い回すことであろう。

2 In-Place 法とヒープソート

ソートアルゴリズムはコンピュータ・サイエンスにおいてよく研究されたテーマのひとつである。考案された多くのソートアルゴリズムのなかでは、空間計算量が $O(1)$ であるものとして、選択ソート、挿入ソート、バブルソート等がよく知られている。

代表的なクイックソートについて、空間計算量が $O(n)$ であるといわれる。理由として、プログラムに

おいて再帰呼出し（プログラムのなかで自分自身を実行すること）があることがあげられる。再帰呼出しの仕組みはプログラム言語によって異なるが、スタックメモリ領域を使うのが一般的である。明示しないスタックメモリ領域を入力データのサイズに比例して使うということで空間計算量が O(n) であるとされる。

さて、以下では、時間計算量が $O(n \log n)$ であることを実現しているのはヒープソート（Cormen et al.、2001）について考察する。再帰呼出しを利用した実装もみられるが、空間計算量を正しく調べるため、非再帰呼出しによる実装を示す。

```
void heapSort(int *a, int n) {
  int i, j, k, t;
  for (i = 1; i < n; ++i) {
    j = i;
    while (j > 0) {
      k = (j-1)/2;
      if (a[k] >= a[j]) break;
      t = a[j], a[j] = a[k], a[k] = t;
      j = k;
    }
  }
  for (i = n-1; i >= 0; --i) {
    t = a[i], a[i] = a[0], a[0] = t;
    j = 0;
    while (1) {
      if (2*j + 1 >= i) break;
      k = 2*j + 1;
      if (k+1 < i && a[k] < a[k+1]) k++;
      if (a[k] <= a[j]) break;
      t = a[j], a[j] = a[k], a[k] = t;
      j = k;
    }
  }
}
```

プログラムは引数が配列 a とその長さ n の 2 つであり、戻り値はないが、終了時に各要素が昇順に並べ替えられ、配列 a に格納されている。

空間計算量が $O(1)$ であることは明らかであろう。プログラムのなかで使う変数として、i、j、k、t の 4 つしかないからである。また、時間計算量が $O(n \log n)$ であることについては、最初の for という繰り返しでは、各要素に対し、その 1/2 の添字にある要素と比較・交換処理を繰り返して行うこと、2 つめの for という繰り返しでは、末尾から先頭に進みながら、各要素に対し、その 2 倍の添字近辺にある要素との比較･交換処理を繰り返して行うことによって確認できる。2 つの for による繰り返しでは、いずれも $O(n \log n)$ という時間量になっていて、全体としての時間計算量も $O(n \log n)$ である。

このように、ヒープソートは入力配列をうまく比較･交換することによって、余分なメモリ領域をほとんど使わずに入力データを高速にソートする、という意味では、最も優れたソートアルゴリズムのひとつとして評価されている。

このヒープソートを利用すれば、たとえば、n 番目に小さい値を配列から見つけること等は空間計算量 $O(1)$ で実現する。

3 In-Place 法と尺取法

尺取法では、配列に対して、2 つのポインタを用意し、片方のポインタを先行させる。ある条件が成立すると、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。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り、全体の時間計算量は $O(n)$ になる。空間計算量についてはポインタ 2 つの追加、さらに条件確認のためのいくつかの変数の追加だけなら、$O(1)$ となる。

実例として、以下の問題について考えよう。

非負の整数でつくられる数列 a（数列 a の長さは n）が与えられ、つぎの条件を満たす連続した部分数列がいくつ存在するか。

条件：連続した部分数列に含まれる整数の累積和が K_1 以上かつ K_2 以下。ただし、K_1 、K_2 は整数、$K_1 \leqq K_2$ 。

また、現有のコンピュータシステム環境やプログラミング環境に合わせて、数値に関する制約条件をとりあえず $0 \leqq a_i$、n、K_1、$K_2 \leqq 10^9$ とする。

たとえば、$n = 6$、$a = (2, 4, 0, 5, 2, 3)$、$K_1 = 7$、$K_2 = 13$ であれば、答えは 8 となる。条件を満たす連続した部分数列はつぎの 8 つあるからである。(2, 4, 0, 5)、(2, 4, 0, 5, 2)、(4, 0, 5)、(4, 0, 5, 2) 、(0, 5, 2)、

(0, 5, 2, 3)、(5, 2)、(5, 2, 3)。

問題に対する解き方はさまざま考えられるが、In-Place 法と尺取り法とのコンビによるプログラムは以下のようであろう。なお、プログラム中の 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 cnt_subarray(int *a, int n,
  int K1, int K2) {
  int left = 0;
  int right1 = 0, right2 = 0;
  Int s1 = 0, s2 = 0, ans = 0;
  while (right1 < n) {
    while (right1<n && s1+a[right1]<K1)
      s1 += a[right1++];
    while (right2<n && s2+a[right2]<=K2)
      s2 += a[right2++];
    ans += right2-right1;
    if (left < right1) s1 -= a[left];
    else right1++;
    if (left < right2) s2 -= a[left];
    else right2++;
    left++;
  }
  return ans;
}
```

プログラムは引数が与えられる配列 a とその長さ n、そして、K_1 と K_1 の 4 つであり、戻り値が答えである。プログラム実行中、配列 a の書き換えはしていない。

スペースの都合でプログラムに対する詳細の説明は割愛するが、使っている変数は、$left$、$right_1$、$right_2$、s_1、s_1、ans の 6 つであるので、空間計算量は $O(1)$ となる。時間計算量については、数列(配列)の各要素は $left$、$right_1$、$right_2$ がそれぞれ 1 回走査するので、全体では $O(n)$ となる。つまり、時間計算量 $O(n)$、空間計算量 $O(1)$ となるプログラムになっている。

プログラムの特徴としては、ポインタ $left$ と $right_1$ との間にある部分数列の累積和が K_1 以上、$left$ と $right_1$ 直前の間にある部分数列の累積和が K_1 以下、という非負整数の累積和の単調性を活用したところにある。

このように、尺取法は In-Place 法と相性がよく、活用されることを期待する。

4 In-Place 法とビット操作

In-Place 法ではその定義から、入力データ以外に入力データサイズに比例するメモリ領域は使えない。その制限を突破するには、入力データの順序を変えるだけでなく、個々の入力データにある情報をもたせるのも一案だろう。つまり、入力データの要素にビット情報を追加して活用するという発想である。

たとえば、多くのコンピュータ言語ではバイト単位のメモリ領域で整数を格納する。よく見かける 0 ～ 10^9 範囲内の整数を格納するには 30 ビットで足りるが、アクセスの効率性等の観点から、32 ビットを使うのは一般的である。つまり、余った 2 ビットは活用できるわけで、異なる 4 種類の情報をもたせることが可能である。

ビット操作に関するもうひとつの応用は、ビット操作の特徴（倪、2018）を活用するというものである。たとえば、XOR 演算（排他的論理和）は以下の 2 つの問題を解くことに有用である。

4.1 問題 1

正整数の数列 a（数列 a の長さは n）において、つぎの条件を満たす正整数はどれか。

条件 : その正整数は奇数回しか数列に存在せず、他の正整数はすべて偶数回存在する。

具体例 : n = 9、a = (3, 5, 2, 5, 3, 2, 5, 2, 5) に対し、正解は 2 である。

ところが、XOR 演算の特徴として、

x XOR 0 ⇒ x 、x XOR x ⇒ x

がよく知られている。つまり、偶数回現れる整数なら、その XOR 演算では 0 になり、XOR 演算の結果で残ったのは奇数回現れる整数になるわけである。したがって、XOR 演算を使えば、時間計算量が $O(n)$、空間計算量が $O(1)$ であるプログラムづくりが可能となる。

4.2 問題 2

長さが n の整数数列 a において、つぎの条件を満たす整数はどれか。

条件 : その整数は数列に出現する回数が 3 の倍数以外であり、他の整数はすべて 3 の倍数回出現する。

数値に関する制約条件 : $-10^9 \leqq a_i \leqq 10^9$、$0 < n \leqq 10^9$、しかも、$n$ は 3 の倍数ではない。

具体例 : n = 7、a = (-3, 5, -3, -3, 5, 5, 5) に対し、正解は 5 である。n = 8、a = (3, -15, 3, 3, -15, -15, 3, 3) に対し、正解は 3 である。n = 8、a = (0, 15, -7, -7, 15, 15, -7, 0) に対し、正解は 0 である。

では、問題 2 に対する解答方法を考えてみよう。

(1) ハッシュ法を使う方法

ハッシュ法（倪、2020）を使えば、異なる値が与えられた数列に何回存在するかは直ちに判別可能である。この方法では、時間計算量は $O(n)$、空間計算量は $O(n)$ になる。しかし、In-Place 法では余分なメモリ領域を使わないので、ハッシュ法の採用は見送るしかない。

(2) ヒープソートを使う方法

入力数列をヒープソートによってソートし、連続した同一値の要素数が 3 の倍数以外であれば、その値が解答になる。この方法では、時間計算量は $O(n \log n)$、空間計算量は $O(1)$ であり、In-Place 法の定義に合致する。したがって、ヒープソートによる解法は汎用性が高く、実行時間はつぎに示す (3) より多少遅いが、十分実用的であろう。

(3) ビット操作を使う方法

上記の (2) よりも時間計算量をさらに $O(n)$ に改善するプログラムをつぎに示す。

```
int findOnlyOne(int *a, int n) {
  int i;
  int zero = 0, once = 0, twice = 0;
  for (i = 0; i < n; ++i) {
    if (a[i] == 0) zero++;
    else {
      once = ~twice & (once ^ a[i]);
      twice = ~once & (twice ^ a[i]);
    }
  }
  if (zero % 3) return 0;
  return (n % 3 == 1)? once: twice;
}
```

プログラムの引数は 2 つ、入力数列 a とその長さ n、戻り値は数列に現れる回数が 3 の倍数でない要素の値である。

プログラムのなかのビット操作について、「~」は NOT、「&」は AND、「^」は XOR を意味する。本方法は明らかに、時間計算量が $O(n)$、空間計算量が $O(1)$ である。

確かに、ビット操作による解法に思いつくことはそう簡単ではないが、ビット操作の優位性を見出す一例であろう。

終わりに

IT 社会の進行で、スマートフォンや情報機器が一般の国民にとっても日常生活に欠かせないものになっている。これからはさらに AI 等の活用が社会に浸透していくことが期待されているなか、政府も産業界もコンピュータ・サイエンスの専門家の育成に力を入れている。

本文はアプリやソフトウェアの中核をなすアルゴリズムに対し、これまで専門書にあまり集中して議論してこなかった In-Place 法について、その考え方や、活用方法について、

・データ配列
・ソートアルゴリズム
・尺取法との連携
・ビット操作の応用可能性

という 4 つの視点からまとめたものである。

空間計算量の使用を $O(1)$ に制限される条件は確かに厳しいものだが、リソースを無駄なく活用する発想が独創的なアルゴリズムを生む可能性を秘めているので、新しいアルゴリズムの考案に貢献するものと期待する。

参考文献

Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford (2001), *Introduction to Algorithms, Second edition*, The MIT Press.

倪 永茂（2018）、「C 言語プログラミング実践教育におけるビット演算」宇都宮大学国際学部研究論集 (46)、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

Algorithm Research Series: In-Place Method

NI Yongmao

Abstract

The in-place method is a generic term for algorithms that require little or no additional memory space in computer science. In other words, it refers to algorithms whose space complexity is $O(1)$. This article discusses the In-Place method, which has not been discussed in many technical books on the core algorithms. As its concept, and usage, we mentioned

- Data array
- Sorting Algorithm
- Cooperation with two pointer method
- Application of bit manipulation

Although the condition of limiting the use of space computation to $O(1)$ is severe, the idea limiting the use of memory has the potential to creat new effective algorithms, which is the aim of the in-place method.

（2020 年 5 月 25 日受理）