アルゴリズム 68－5
（1999．5．10）

作業場所が小さいマージソートと計算量の評価

萩原洋一
東京農工大学総合情報処理センター
池田諭　　中森眞理雄
東京農工大学工学部情報コミュニケーション工学科
小金井市 中町2-24-16 〒184-8588
E-Mail: { hagi | bisu | nakamori } @cc.tuat.ac.jp

あらまし

マージソートは時間計算量が $O(n \log n)$ (n はソートされるレコードの個数) であり高速であるが, 内部ソートとして実行する場合, 作業場所として大きさ n の配列を要するのが欠点であるとされている. 本論文では, 作業場所として数語だけを要するマージソートを提案する. 新しいマージソートの時間計算量は $O(n \log^2 n)$ であり, 従来のアルゴリズムより悪いが, これは作業場所とのトレードオフの結果である.

キーワード

ソート, マージソート

Merge Sort with Small Working Area and Its Complexity

Yoichi Hagiwara
Computer Center, Tokyo University of Agriculture and Technology
Satoshi Ikeda　　Mario Nakamori
Department of Computer Science
Tokyo University of Agriculture and Technology
Nakacho 2-24-16, Kaganei, Tokyo 184-8588
E-Mail: { hagi | bisu | nakamori } @cc.tuat.ac.jp

Abstracts

Mergesort is one of the fastest sorting algorithm, since it requires only $O(n \log n)$ of computing time. Mergesort, however, requires an array of size n as working area, when executed as an internal sort. In the present paper, we propose an algorithm which is a modified version of mergesort. The space complexity of the proposed algorithm is only $O(1)$. The time complexity is $O(n \log^2 n)$, which is worse than the existing merge sort and is the result of the tradeoffs between time and space complexities.

key words

sort, mergesort

1. はじめに

ソート (sort; 整列 (order) とも言う) とは, 多数のレコードを, あるキー (key) に関して昇順あるいは降順に並べかえる操作である. ソートは, それ自体がアルゴリズム研究の対象として興味深いだけでなく, さまざまな情報処理において, 前処理として現れるため, 重要である. 今までに, ソートのアルゴリズムとして, きわめて多数の, 多様なものが提案されている [1], [2].

ソートのアルゴリズムは, 種々の観点から分類される. 計算機の内部メモリだけで行うソートは内部ソート, 外部記憶装置の使用を前提としたソートは外部ソートとよばれる. もっとも, 内部メモリが十分大きな計算機では, 外部ソートは内部ソートとして実行することが可能である.

レコードの桁数が既知である場合に, そのことを積極的に利用するソートアルゴリズムもある. 基数法 (radix sort) はその代表的な例である.

ソートのアルゴリズムの評価方法は, 理論的な観点からは, 最悪の場合の処理手数, 平均の処理手数などが用いられる. もちろん, 最も重要なのは, 実際にデータを用いて実行したときの実際の処理時間である. 例えば, クイックソート (quick sort) は, 最悪の場合の処理手数は $O(n^2)$ (n はレコードの個数) であるが, 平均の処理手数は $O(n \log n)$ であることが知られており [3], 実際の処理時間も短いと言われている.

また, レコードの個数が小さいときは, 手数が $O(n^2)$ の初歩的なソートアルゴリズムである挿入法 (insertion sort) が, 実際的には, 他のソートアルゴリズムより速いことも, しばしば観察されている.

理論的には, レコードの桁数が既知であることを利用しないいかなるソートアルゴリズムも, 最悪の場合の処理手数は $O(n \log n)$ を下回らないことが知られている (基数法は, レコードの桁数が既知であることを用いるので, 処理手数は $O(n \log n)$ より小さい $O(n)$ である).

本論文で取り上げるマージソートは, 理論的な手数が (最悪の場合においても) $O(n \log n)$ のアルゴリズムである. マージソートは, しばしば, 3本のテープを用いる外部ソートとして説明されることが多いが, もちろん, 内部ソートとして実行可能である. マージソートは, (レコードを昇順に並べる場合) レコード列を単調非減少で極大な部分列である連 (run) に区切り, 隣り合う連 (1番目の連と2番目の連, 3番目の連と4番目の連,...のように) をマージ (merge) する, という操作を繰り返す方法である.

マージソートを内部ソートとして実行する場合, 隣り合う連をマージした結果は, オリジナルの配列に直に書き込むのではなく, 一旦作業場所に書き出す必要がある. この作業場所は, 最悪の場合, 大きさが n となるので, 一般には, マージソートはオリジナルの配列と同じ大きさの配列を作業場所として要すると考えられている.

本論文では, 作業場所として数語しか要しないマージソートを提案する. ただし, 本論文で提案するマージソートの手数は $O(n \log n)$ ではなく, 最悪の場合, $O(n \log^2 n)$ となるが, このことは, 所要メモリと処理時間のトレードオフの現れと考えられる.

以下に, 2 ではマージソートの考え方を述べ, 3 では本論文のマージソートの基礎となる配列の回転の方法を述べる. 4 では新たなマージソートのアルゴリズムを述べ, 5 ではそのアルゴリズムの手数を評価する.

2. マージソートの考え方

以下では, n 個のレコードをソートする場面を考え, それらのレコード列は配列 a の中に入っているとする. また, レコード中のキー以外の要素は省略することにする. したがって, 配列要素 $a[i]$ のレコードのキー値を単に $a[i]$ と記すことにする.

マージソートの概略は次のとおりである.

```
var...(略) ...
repeat
  runcount := ' 1 ';
  top2runs(a, 0, n, q, r);   p := 0 ;
  while q < r do
    begin
      runcount := ' ≥ ';
      merge(a, p, q, r) ;
      top2runs(a, r, n, q1, r1);   p := 0 ;   p := r;   q := q1;   r := r1
    end
until runcount := ' 1 '
```

ただし, 変数 *runcount* は文字型, その他の変数は *integer* 型である.

また, 手続 $top2runs(a, l, u, q, r)$ は, 配列 a 中の $a[l+1], \ldots, a[u]$ の範囲の冒頭の2つの連 $a[l+1], \ldots, a[q]$ と $a[q+1], \ldots, a[r]$ を抽出し, q と r を返す. すなわち,

$$a[l+1] \leq \ldots \leq a[q] > a[q+1] \leq \ldots \leq a[r]$$

となる q と r を求める. もし,

$$a[l+1] \leq \ldots \leq a[u]$$

であるなら, $q = u,\ r = u$ とする.

手続 $merge(a, p, q, r)$ は, 配列 a 中の連 $a[p+1], \ldots, a[q]$ と連 $a[q+1], \ldots, a[r]$ をマージした結果を $a[p+1], \ldots, a[r]$ に入れる (ただし, $p \leq q \leq r$ とする).

手続 $top2runs(a, l, u, q, r)$ の手順は次のとおりである.

```
procedure top2runs(a, l, u, q, r) ;
  var...(略) ...
  begin
    q := l + 1 ;   extend := ' stop ' ;
    while extend = ' go ' do
      if q = u
        then extend := ' stop '
        else
          if a[q] > a[q + 1]
```

```
            then extend := ' stop '  else q := q + 1 ;
        if q = u
          then  begin r := q;  extend := ' stop '  end
          else  begin r := q + 1;  extend := ' go ' ; end
        while extend = ' go ' do
          if r = u
            then extend := ' stop '
            else
              if a[r] > a[r + 1]
                then extend := ' stop '  else r := r + 1 ;
    end
```

手続 $merge(a, p, q, r)$ の手順は次のとおりである.

```
  procedure merge(a, p, q, r) ;
    var...(略) ...
    begin
      i := p + 1 ;  j := q + 1 ;
      for k := 1 to r − p do
        begin
          if i < q then x := a[i] else x := ∞ ;
          if j < r then y := a[j] else y := ∞ ;
          if x ≤ y
            then
              begin work[k] := x ;  i := i + 1
              end
            else
              begin work[k] := y ;  j := j + 1
              end ;
        end
      for k := 1 to r − p do
        a[p + k] := work[k];
    end
```

上記の手続 $merge$ において, $work$ は添字 $1, 2, \ldots$ をもつ作業用の配列であり, 大きさは n だけ確保しておく必要がある.

3. 準備 — 配列の回転

本論文で提案するマージソートでは, 配列の回転が本質的な役割を果たすので, 本節ではその手順について考察する.

まず, 大きさ m の配列 b の内容を右に k 語回転する手順, すなわち, 配列要素 $b[i]$ の内容を

$0 < i \leq m-k$ ならば $b[i+k]$ に

$m-k < i \leq m$ ならば $b[i-m+k]$ に

移す手順を考える.

この問題に対しては, 所要作業場所が $O(1)$ で計算手数が $O(m)$ の巧妙なアルゴリズムが知られている [4] (素朴な方法では, 所要作業場所が $O(1)$ なら計算手数が $O(km)$ になり, 計算手数を $O(m)$ にするには所要作業場所が $O(n)$ となる).

```
procedure rotate(b, m, k) ;
  begin
    reverse(b, 0, m − k) ;
    reverse(b, m − k, m) ;
    reverse(b, 0, m)
  end
```

ただし, $reverse(b,l,u)$ は, 配列要素 $b[l+1],\ldots,b[u]$ の並びを左右逆転する手続きである. すなわち,

```
procedure reverse(b, l, u) ;
  begin
    i := l + 1 ;  j := u ;
    while i < j do
      begin  swap(b[i], b[j]) ;  i := i + 1 ;  j := j + 1
      end
  end
```

ここで, $swap(x,y)$ は, 変数 x と変数 y の内容を交換する手続きである.

上記の配列の回転 $rotate$ は, 対象とする配列の範囲に "隙間" がある場合にも拡張できる.

今, $p < q \leq r < s$ とし, 配列要素 $b[p+1],\ldots,b[q]$ と $b[r+1],\ldots,b[s]$ を右に $q-p$ 語回転する手続 $rotate2(b,0,m,p,q,r,s)$ を考える. ただし, 配列要素 $b[q]$ の右隣の要素は $b[r+1]$ と解釈し, 配列要素 $b[s]$ の右隣の要素は $b[p+1]$ と解釈する.

```
procedure rotate2(b, p, q, r, s)
  begin
    reverse(b, p, q) ;
    reverse(b, r, s) ;
    reverse2(b, p, q, r, s)
  end
```

手続 $rotate$ の3番目の $reverse$ が $rotate2$ では $reverse2$ に変っている. この $reverse2$ は, 配列要素 $b[p+1],\ldots,b[q]$ と $b[r+1],\ldots,b[s]$ を, "隙間" を越えて左右逆転する手続である. すなわち,

```
procedure reverse2(b, p, q, r, s) ;
  begin
    i := p + 1 ;  j := s ;
    while i < j do
```

```
    begin
      swap(b[i], b[j]) ;
      if i < q then i := i + 1 else i := r + 1 ;
      if j > r + 1 then j := j − 1 else j := q ;
    end
  end
```

前述の $rotate(b, 0, m, k)$ は $rotate2(b, 0, m-k, m-k+1, m)$ と書くこともできる.

4. 相対連を用いたマージ

配列 a 中の隣り合う連

$$A_1 : a[p+1], \ldots, a[q], \quad A_1 : a[q+1], \ldots, a[r]$$

に対して, これらをマージした結果は, 一般には, A_1 と A_2 の要素を複数ずつ交互に並べたものとなる. マージされたときに連続する A_1 と A_2 の要素の列を相対連とよぶ. すなわち,

$$p = p_0 \le p_1 < \cdots < p_{l-1} < p_l = q = q_0 < q_1 < \cdots < q_{l-1} \le q_l = r$$

$$a[i] \le a[j] \quad (p_t < i \le p_{t+1}, \ \ q_t < j \le q_{t+1}\text{のとき})$$

$$a[i] \ge a[j] \quad (p_t < i \le p_{t+1}, \ \ q_{t-1} < j \le q_t\text{のとき})$$

となるとき, 配列要素の列 $a[p_t+1], \ldots, a[p_{t+1}]$ や $a[q_t+1], \ldots, a[q_{t+1}]$ が相対連である.

このように連が相対連に区切られているとき,

$rotate2(a, p_1, p_2, q_0, q_1)$;
$rotate2(a, p_3, p_4, q_2, q_3)$;
$rotate2(a, p_5, p_6, q_4, q_5)$;
.....

のように実行することにより, 相対連の数は半分になる. この操作を連 A_1, A_2 における相対連の数が 1 となるまで繰り返すと, 連 A_1, A_2 のマージが完了する.

なお, 上記各 $rotate2$ の手間は

$$p_2 - p_1 + q_1 - q_0 + p_4 - p_3 + q_2 - q_1 + p_6 - p_5 + q_4 - q_3 + \cdots$$

であり, これは $r - p$ 以下であることに注意されたい.

相対連を利用したマージ $merge2$ を次に示す.

```
  procedure merge2(a, p, q, r) ;
    var...(略) ...
    begin
      i0 := p;  j0 := q;  k0 := q;  l0 := r;
      while (i0 < k0)and(j0 < l0) do
        begin  relative(a, i0, k0, i1, a[j0 + 1]);
```

```
        if i1 < k0
          then
            begin  relative(a, j0, l0, j1, a[i1 + 1]);
              if j1 < l0
                then
                  begin  relative(a, i1, k0, i2, a[j1 + 1]);
                    if j1 < l0
                      then  relative(a, j1, l0, j2, a[i2 + 1])
                      else  j2 := l0 ;
                    retate2(a, i1, i2, j0, j1)
                  end
                else
                  begin i2 := k0 ; j2 := l0
                  end
            end
          else
            begin j1 := l0 ; i2 := k0 ; j2 := l0
            end
        i0 := i2 ; j0 := j2
      end
  end
```

ここで, $relative(a, l, u, p, x)$ は配列要素の列 $a[l+1], \ldots, a[u]$ において

$$a[p] \leq x < a[p+1]$$

となる p を求める手続である ($a[u] \leq x$ のときは p として u を返す).

```
  procedure relative(a, l, u, p, x) ;
    var...(略) ...
    begin
      p := l + 1 ; extend := ' go ' ;
      while (extend := ' go ') and (p ≤ u) do
        if a[p] ≤ x
          then p := p + 1
          else extend := ' stop ' ;
      p := p − 1 ;
    end
```

前述のマージソートのプログラムで *merge* の代りに *merge2* を用いたものが本論文で提案するマージソートである.

5. 計算量の評価

本論文で提案したマージソートは, 作業場所として配列を用いていないので, 空間計算量は $O(1)$ である.
時間計算量 (手数) を求めるために, 隣り合う連

$$A_1 : a[p+1], \ldots, a[q], \quad A_1 : a[q+1], \ldots, a[r]$$

を $merge2$ によってマージする手数を考察する.

これら 2 つの連に含まれる相対連の数はそれぞれ $q-p$ 以下, $r-q$ 以下で, 相対連の数はこれら 2 つの連では等しいので, $\min(q-p, r-q)$ である. したがって, $merge2$ の手間は $(r-p)\log_2 \min(q-p, r-q)$ 以下であり, したがって, $(r-p)\log_2 n$ 以下である.

配列中の各連の長さを $r_1, r_2, \ldots, r_m$ とするとき, 連1と連2, 連3と連4,... を $merge2$ によってマージするのに要する手間の和は高々

$$(r_1 + r_2 + \cdots + r_m)\log_2 n = n \log_2 n$$

である. マージソートはこれを $\log_2 n$ 回行うので, 手間の総和は $O(n \log^2 n)$ となる.

長さが r_1 と r_2 の連を $merge2$ によってマージする手数が, 最悪の場合, $O(r_1 + r_2)$ であることから, 本論文で提案したマージソート全体で, 最悪の場合, $O(n \log^2 n)$ である.

6. まとめ

所要メモリの大きさが $O(1)$ のマージソートを提案し, 時間計算量が $O(n \log^2 n)$ であることを示した. 今後は, このアルゴリズムの現実的な実行時間を調べる必要がある.

参考文献

[1] D. E. Knuth, *The Art of Computer Programming, Vol.3 (Sorting and Searching)*, Addison-Wesley, 1973.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] 溝井, 尾崎, クイックソートの時間計算量の確率的解析, 信学論 (A), J78A, 1142-1148 (1995).

[4] J. L. Bentley, *Programming Pearls*, Addison-Wesley, 1986.