アルゴリズム及び演習 第 7 回演習解答

小野 孝男*

2007 年 6 月 4 日

1. 以下, 一般性を失うことなく $\alpha \geq 1-\alpha$ であると仮定する.
   n 個のデータに対する `partition` 操作の実行時間は $O(n)$ である. そこで, 再帰呼び出しのレベルごとに `partition` の実行時間を考えると次のようになる:
   (a) 最初に呼び出される `quicksort` では n 個のデータに対し `partition` を行うので実行時間は $O(n)$.
   (b) そこから再帰的に呼び出される `quicksort` では αn 個と $(1-\alpha)n$ 個のデータのそれぞれに対して `partition` を行い, 実行時間は合計で $O(n)$.
   (c) それらから再帰的に呼び出される `quicksort` で処理するデータの個数はそれぞれ $\alpha^2 n$, $\alpha(1-\alpha)n$, $\alpha(1-\alpha)n$, $(1-\alpha)^2 n$ 個なので `partition` の実行時間の合計はやはり $O(n)$.
   つまり, どのレベルにおいても `partition` の実行時間の合計は必ず $O(n)$ である. 従って, クイックソート全体の実行時間を求めるには**再帰呼び出しがどのくらい深くなるか**がわかればよい.
   ここで, `partition` によってデータの個数が必ず $\alpha : (1-\alpha)$ の比に分割されるので, 深さ k の再帰呼び出しにおいてソートするデータは全体で高々 $\alpha^k n$ 個である. よって, 再帰呼び出しの最大深さを K とおくと $\alpha^K n \leq 2$ であり, $K = \lceil \log_\alpha (2/n) \rceil$ となる. ここで $\alpha < 1$ は定数なので
   $$K = \left\lceil \log_\alpha \frac{2}{n} \right\rceil = \left\lceil \frac{\log(n/2)}{-\log \alpha} \right\rceil = O(\log n)$$
   となる. つまり, 再帰呼び出しの深さは高々 $O(\log n)$ なのでソート全体の実行時間は $O(n) \times O(\log n) = O(n \log n)$ である.
2. 単純に「n 個のデータから最大値と最小値を求める」という方針だと,「最大値を求める」のに $n-1$ 個, 最小値を求めるのに (最大値以外のデータのみを考えればよいので) $n-2$ 回の比較が必要であり, 合計 $2n-3$ 回の比較を行うことになる. つまり, 高々 $3n/2$ 回の比較で最大値と最小値を求めるには, 最大値と最小値を「同時に」求める必要がある.
   そこで, n 個のデータ (当面, n は偶数であると仮定する) $x_1, x_2, \ldots, x_n$ を 2 個ずつ組にし, まず各組ごとで比較を行う. つまり, 整数 i $(1 \leq i \leq n/2)$ に対しまず x_{2i-1} と x_{2i} を比較し, その

* ono@is.nagoya-u.ac.jp

うち大きい方を y_i, 小さい方を z_i とする. この分類には, 全体で $n/2$ 回の比較が必要である.
ここで重要なのは, 「$x_1, \ldots, x_n$ の最大値は $y_1, \ldots, y_{n/2}$ の最大値であり, また $x_1, \ldots, x_n$ の最小値は $z_1, \ldots, z_{n/2}$ の最小値である」ということである. これはほぼ自明である:「最大値」は他のどのデータよりも大きな値であるから, 組になった相手より大きい. 従って y_i の中に存在しなければならない. 最小値についても同様である. このことから, $x_1, \ldots, x_n$ の最大値と最小値はそれぞれ「$y_1, \ldots, y_{n/2}$ の最大値」と「$z_1, \ldots, z_{n/2}$ の最小値」である. これらはそれぞれ $n/2-1$ 回の比較で求めることができるので, 全体でも $n/2+(n/2-1)+(n/2-1) = 3n/2-2$ 回の比較で $x_1, \ldots, x_n$ の最大値と最小値を求めることができる.
最後に n が奇数であるときを考えるが, このときには最後の要素 (x_n) が最大値と最小値のいずれの候補でもあると考えるのが最も簡単である. つまり, $y_{(n+1)/2} = z_{(n+1)/2} = x_n$ と考える. すると最初の比較が $(n-1)/2$ 回, 最大値と最小値を求めるための比較がそれぞれ $(n+1)/2-1 = (n-1)/2$ 回必要であり, 合計の比較回数は $(n-1)/2+2\times(n-1)/2 = 3n/2-3/2$ である.

3. 「2 番目に小さなデータ」を求める最も単純な方法は「最も小さいデータを取り除いた, 残りのデータから最小値を求める」というものであるが, これは n 個のデータから最小値を求め ($n-1$ 回の比較), その後残り $n-1$ 個の中から再度最小値を求める ($n-1-1 = n-2$ 回の比較) ことになり, 比較回数は合計で $2n-3$ である. しかし, この比較にはかなりの無駄が含まれていることがわかる. つまり, 「異なる 2 個のデータよりも大きなデータは, 2 **番目に小さなデータ**となることはない」からである. つまり, 最小値を求める途中の比較結果を活用することにより, 2 回目の比較回数を減らすことが可能である.
まず考えられるのがヒープを使う方法である. つまり, n 個のデータに対して「ヒープを構築する」のに $O(n)$ 回, 「ヒープから最小値を削除する」のに $O(\log n)$ 回の比較が必要であり, 合計で $O(n)+O(\log n)$ という比較回数が得られる. しかし, ここでは $O(\cdot)$ に含まれる定数係数が未確定である. 実際には定数係数は 1 より大きくなる可能性がある. そこで, この考え方を生かして別の方法を考える.
「2 番目に小さなデータ」は, 「最小値と直接比較したデータ」のいずれかである. そこで, 最小値を求めるときに「そのデータと比較を行ったデータ」をリストとして保持することにする. 完全 2 分木のような形で比較すれば, どのデータも比較回数は木の高さ, つまり $\lceil \log n \rceil$ で抑えることができる. ここで, 完全 2 分木のような形にするためにはキューを使うのが簡単である. 必要なデータ構造も含めたプログラム全体をリスト 1 に示す.

リスト 1　2 番目に小さなデータを求める

```
#include <stdlib.h>

typedef struct TreeItem *Tree;
struct TreeItem {
    int data;
    Tree child, sibling;
};

Tree newTree(int x)
{
```

```
    Tree p = malloc(sizeof (TreeItem));
    p->data = x;
    p->child = p->sibling = NULL;
    return p;
}

struct TreeQueue {
    Tree *q;
    int size;
    Tree *head, *last;
};

void clearTrereQueue(TreeQueue *q)
{
    q->size = 0;
    q->head = q->last = q->q;
}

void newTreeQueue(TreeQueue *q, int n)
{
    q->q = malloc(sizeof (Tree) * n);
    clearTreeQueue(q);
}

void insertTree(TreeQueue *q, Tree t)
{
    *q->head++ = t;
    ++q->size;
}

Tree headTree(TreeQueue *q)
{
    return *q->last;
}

void deleteTree(TreeQueue *q)
{
    ++q->last;
    --q->size;
}

void swapQueue(TreeQueue *q, TreeQueue *r)
{
    Tree *t;
    int s;

    t = q->q; q->q = r->q; r->q = t;
    s = q->size; q->size = r->size; r->size = s;
    t = q->head; q->head = r->head; r->head = t;
    t = q->last; q->last = r->last; r->last = t;
}

void compares(TreeQueue *queue, TreeQueue *temp)
{
    clearTreeQueue(temp);
    while (queue->size > 1) {
        Tree t1, t2;
        t1 = headTree(queue);
        deleteTree(queue);
        t2 = headTree(queue);
        deleteTree(queue);

        if (t1->data > t2->data) {
            t1->sibling = t2->child;
```

```
            t2->child = t1;
            insertTree(temp, t2);
        } else {
            t2->sibling = t1->child;
            t1->child = t2;
            insertTree(temp, t1);
        }
    }
    if (queue.size == 1) {
        insertTree(temp, headTree(queue));
        deleteTree(queue);
    }
    swapQueue(queue, temp);
}

int find2ndmin(int a[], int n)
{
    int i;
    int secondmin;
    Tree t;
    TreeQueue queue;
    TreeQueue q;
    newTreeQueue(&queue, n);
    newTreeQueue(&q, n);

    for (i = 0; i < n; ++i) {
        insertTree(&queue, newTree(a[i]));
    }

    while (queue.size > 1) {
        compares(&queue, &q);
    }

    t = headTree(&queue);
    t = t->child;
    secondmin = t->data;
    while ((t = t->sibling) != NULL) {
        if (t->data < secondmin) {
            secondmin = t->data;
        }
    }

    return secondmin;
}
```

find2ndmin の最初の **while** で最小値を求め, そこで得られた情報を元に次の **while** で 2 番目に小さなデータを求めている. 最初の **while** の k 回目の繰り返しの先頭において, キュー queue に入っている木は最後の 1 個を除いて全て 2^{k-1} 個の要素を持つ. 従ってこの繰り返しは $\lceil \log n \rceil$ 回行われ, 最後に残った木の根が最小値である. その要素と比較した要素はその子 (child) を先頭に, sibling をたどることで見付けることができる. 最初の **while** ループの繰り返しが $\lceil \log n \rceil$ 回なので, このリストの長さも高々 $\lceil \log n \rceil$ である. つまり, 次の **while** ループは高々 $\lceil \log n \rceil - 1$ 回実行される. 比較回数は $(n-1) + (\lceil \log n \rceil - 1) \leq n + \log n - 1$ で抑えられる (木の表現については図 1 を見よ).

4. 4 個のデータを a, b, c, d とおく. 比較回数を減らすためには, 1 回の比較でなるべく可能性を均等にわけるのが望ましい. できれば, ちょうど半分ずつにするのがよい. そこで, 次のように比較してゆく:

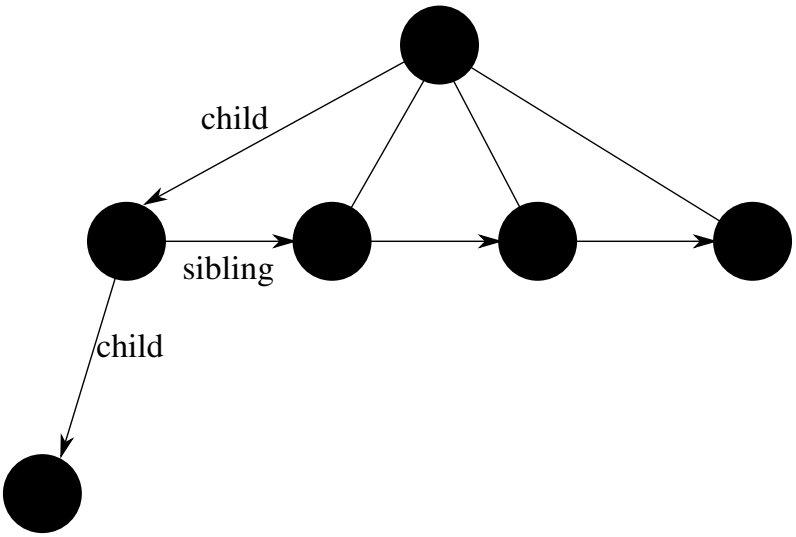

図 1　2 個のポインタによる一般的な木の表現

(a) まず a と b を比較し, 改めて大きい方を a, 小さい方を b とする. これで, 4 個のデータの並び方の可能性は 24 通りから 12 通りに減る.

(b) 次に c と d を比較し, 改めて大きい方を c, 小さい方を d をする. 残る可能性は $12/2 = 6$ 通りである.

(c) ここで a と c を比較する. 一般性を失うことなく $a > c$ と仮定する. また, $a > b$, $c > d$ も保存されるものとする. この時点で a が最大であることがわかり, 残りは c, d に対して b がどこに入るかの 3 通りである.

(d) そこで, b と c を比較する. $b > c$ なら $a > b > c > d$ であることがわかる.

(e) この時点で $a > c > (b, d)$ であり, b と d を比較すれば 4 個のデータをソートすることができる.