Research Contribution

Roughly Sorting: A Generalization of Sorting

YOSHIHIDE IGARASHI* and DERICK WOOD**

A sequence $\alpha=(a_1, \ldots, a_n)$ is k-sorted if and only if for all $1 \leq i, j \leq n$, $i<j-k$ implies $a_i \leq a_j$. A k-sorting algorithm called k-Bubblesort is first designed, since it is a generalization of Bubblesort. Next, a k-sorting algorithm called k-Quicksort is designed, as well as an algorithm for transforming a rougly sorted sequence into a less roughly sorted sequence. It is shown that k-Quicksort is time-optimal within a constant factor if k satisfies a certain condition. Parallel implementation of k-Bubblesort and k-Quicksort are also discussed.

1. Introduction

Sorting is an essential part of data processing and algorithm design. However, in some applications it is sufficient to sort sequences roughly rather than completely. The concept of rough sorting has appeared in papers by Sado and Igarashi [11, 12] on parallel sorting on a mesh-connected processor array. They have designed fast parallel algorithms incorporating a method of roughly sorted subfiles. Their algorithms are faster than an algorithm based on merging four completely sorted subfiles iteratively. This is a good example of how rough sorting contributes to the improvement of the overall time efficiency. Another example is a dynamic data structure in which the sorted order is often disturbed by inserting new items. In this case rough sorting of the initial data may be acceptable, and it may be better to have a searching algorithm that works for roughly sorted files. Whenever we need a completely sorted file, we sort the roughly sorted files. The notion of presorted lists [4, 8, 9, 10] is a related concept, but different from that of roughly sorted lists.

In this paper, we study rough sorting as a generalization of sorting. In Section 2, we give definitions and notations for understanding this paper. In Section 3, we design a rough sorting algorithm that is a generalization of Bubblesort. A parallel implementation of the algorithm is given in Section 4. In Section 5, we design a rough sorting algorithm that is a generalization of Quicksort. This algorithm is time-optimal within a constant factor (that is, we ignore the coefficient of the leading term and the minor terms) under a certain condition. We also describe an algorithm for transforming a roughly sorted sequence into a less roughly sorted sequence.

*Department of Computer Science, Gunma University, Kiryu, 376 Japan.
**Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

2. Preliminaries

Our notion of rough sortedness is formalized by the following definition:

Definition 2.1 A sequence $\alpha=(a_1, \ldots, a_n)$ is k-sorted if and only if, for all i, j in $\{1, \ldots, n\} i<j-k$ imples $a_i \leq a_i$.

Suppose that $(a_{\sigma_1}, \ldots, a_{\sigma_n})$ is the completely sorted sequence of $\alpha=(a_1, \ldots, a_n)$. If for all $i (1 \leq i \leq n)$, $|i-\sigma_i| \leq k$, then we say that α is k-deviated. We can easily show that if α is k-sorted then it is also k-deviated. However, a k-deviated sequence is not necessarily k-sorted. For example, $\alpha=(3, 1, 4, 2)$ is 2-deviated but not 2-sorted. From Definition 2.1, α is sorted if and only if α is 0-sorted. The radius of α is defined to be the smallest k such that α is k-sorted, and is denoted by ROUGH(α). The radius is a presorted measure satisfying the axioms introduced by Mannila [9]. As shown in Altoman and Igarashi [2], for a given sequence α of length n we can find ROUGH(α) in $O(n)$ time. We now introduce the notion of a b-block, which is important for the next section.

Definition 2.2 Given a sequence $\alpha=(a_1, \ldots, a_n)$, a nonnegative integer b, and an integer i, $1 \leq i \leq n-b+1$, the b-block of α at position i is the subsequence $(a_i, \ldots, a_{i+b-1})$ of α. A b-block of α is a b-block at some position i.

It is well known that 0-sorted sequences can be characterized by the following local conditions: $\alpha=(a_1, \ldots, a_n)$ is sorted if and only if every 2-block of α is sorted.

This local characterization of sorted sequences can be generalized by the following theorem:

Theorem 2.1[7] Let $\alpha=(a_1, \ldots, a_n)$ and k be a nonnegative integer. Then α is k-sorted if and only if every $(2k+2)$-block of α is k-sorted.

The size of blocks specified in the above theorem is optimal in the sense described in the next theorem.

Theorem 2.2[7] For every $k \geq 0$, there exists a sequence α satisfying the following two conditions:

1. α is not k-sorted.

2. Every $(2k+1)$-block of α is k-sorted.

When $\alpha=(a_1, \ldots, a_n)$, $\alpha_{i..j}$ denotes the subsequence $(a_i, \ldots, a_j)$ of α. When α is a sequence, $|\alpha|$ denotes the length of α. The computing time of each algorithm given in the following sections holds both in the worst case and on the average case, unless stated otherwise.

3. *k*-Bubblesort

In this section we consider a generalization of Bubblesort. It is well known that any sequence of length n can be sorted by at most n-1 bubbling passes, where the i-th bubbling pass is described as follows:

for $j:=1$ **to** $n-i$ do
  if $a_j > a_{j+1}$ **then** *exchange* (a_j, a_{j+1})
  {a_j is the j-th item of the sequence}

The bubbling operation compares two adjacent items and exchanges them if they are not in sorted order. In other words, the bubbling operation 0-sorts a 2-block. We now extend the bubbling operation so that it k-sorts a $(2k+2)$-block. This operation is called k-bubbling. Obviously, ordinary bubbling is 0-bubbling. We shall give an algorithm, k- Bubblesort, that k-sorts a given sequence by means of k-bubbling. Note that, in general, there are many k-sorted sequences corresponding to a $(2k+2)$-block and, for that matter, many k-sorted sequences corresponding to any given sequence. Uniqueness is achieved only when $k=0$. Therefore, k-bubbling transforms a $(2k+2)$-block into one of its k-sorted sequences. However, for technical reasons we specify that k-bubbling is a particular transformation defined by the following procedure:

Procedure k-BUBBLE $(\alpha, j, j+2k+1)$;
begin
  if $j=1$ **then begin** {simply k-sort $\alpha_{1..2k+2}$}
    for $i:=1$ **to** $k+1$ **do**
      for $s:=1$ **to** i **do**
        if $a_{k+1+i} < a_s$ **then** *exchange* (a_{k+1+i}, a_s)
    end
  else begin {make a_{j+2k+1} not less than any of $\alpha_{j..j+k}$}
    for $i:=1$ **to** $k+1$ **do**
      if $a_{j+2k+1} < a_{j+i-1}$ **then** *exchange* (a_{j+2k+1}, a_{j+i-1})
    end
end

Using k-BUBBLE, a k-bubbling pass from the first $(2k+2)$-block to the last $(2k+2)$-block can be described as follows:

for $j:=1$ **to** $n-2k-1$ **do**
  k-BUBBLE $(\alpha, j, j+2k+1)$

At the beginning of the j-th ($j \geq 2$) stage of a bubbling pass, the items in $\alpha_{j+k+1..j+2k}$ satisfy the condition of k-sortedness within $\alpha_{1..j+2k}$, since these items satisfy the condition of k-sortedness within the subsequence at the previous stage. Therefore, these items need not move during the j-th stage. This is why we specify k-bubbling as described in k-BUBBLE. Hereafter, we say that terms are k-sorted with respect to a subsequence if and only if the terms satisfy the condition of k-sortedness within the subsequence.

Lemma 3.1 Suppose that $\alpha=(a_1, \cdots, a_n)$ and the following k-bubbling pass is executed:

for $j:=1$ **to** $n-2k-1$ **do**
  k-BUBBLE $(\alpha, j, j+2k+1)$

Then, for all i such that $n-k \leq i \leq n$, a_i is not less than any item in $\alpha_{1..i-k-1}$; that is, the last $k+1$ items are k-sorted with respected to α.

Proof. The proof is by induction on the leftmost position j of the $2k+2$ positions currently being processed in the k-bubbling pass. The set of these $2k+2$ positions are called the window of the k-bubbling.

Basis: $j=1$. At the end of the first stage of the k-bubbling pass the last $k+1$ items in $\alpha_{1..2k+2}$ are obviously k-sorted with respect to $\alpha_{1..2k+2}$.

Induction step: $j \geq 1$. Suppose that at the end of the j-th stage of the k-bubbling pass, the last $k+1$ items in the window are k-sorted with respect to $\alpha_{1..j+2k+1}$. During the $j+1$st stage, the items in $\alpha_{j+k+2..j+2k+1}$ remain unchanged and any item in $\alpha_{1..j+k+1}$ cannot be greater than the items in the same position at the j-th stage. Hence, during the $j+1$st stage, the items in $\alpha_{j+k+2..j+2k+1}$ are k-sorted with respect to $\alpha_{1..j+2k+2}$. From the inductive hypothesis, at the beginning of the $j+1$st stage a_{j+k+1} is no smaller than any item in $\alpha_{1..j}$. Furthermore, a_{j+2k+2} at the end of the $j+1$st stage is the maximum of the items in $\alpha_{j+1..j+k+1}$ and a_{j+2k+2}.

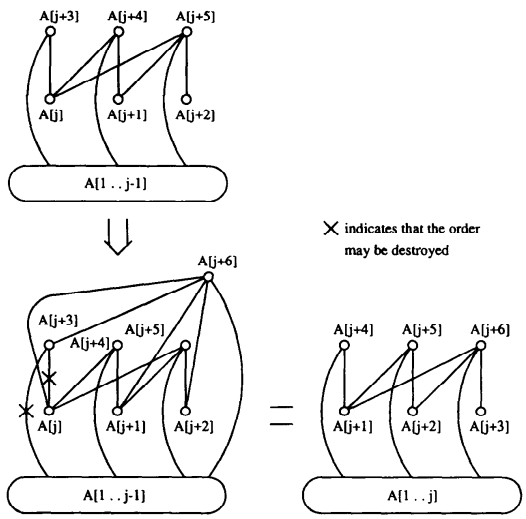

Fig. 1 Change of a partial order by a one-position shift of the window in a 2-bubbling pass.

Hence, a_{j+2k+2} is no smaller than any item in $\alpha_{1\ldots j+k+1}$ at the end of the $j+1$st stage. Thus, the items in $\alpha_{j+k+2\ldots j+2k+2}$ are k-sorted with respect to $\alpha_{1\ldots j+2k+2}$ at the end of the $j+1$st stage. (The situation for $k=2$ is shown in Fig. 1.) Therefore, at the end of the k-bubbling pass, the last $k-1$ items in α are k-sorted with respect to the whole sequence. □

From Lemma 3.1 we can design a k-bubble sorting algorithm as follows: k-Bubblesort consists of $\lceil (n-k-1)/(k+1) \rceil$ k-bubbling passes. The length of the first bubbling pass is $n-2k-1$. The length of each subsequenct bubbling pass is $k+1$ shorter than the previous pass. k-Bubblesort is a natural extension of Bubblesort, and is also called the rough bubblesort. The algorithm can be described as follows:

```
procedure RBUBBLE (α, 1, n, k);
begin
  for i:=1 to ⌈(n-k-1)/(k+1)⌉ do
    for j:=1 to n-k-(k+1)i do
      k-BUBBLE (α, j, j+2k+1)
    {if n-k-(k+1)i<1 then k-BUBBLE is executed once in the second for loop}
end
```

Lemma 3.2 α is k-sorted by RBUBBLE $(\alpha, 1, n, k)$.

Proof. Let $P(i)$ be the following assertion:

At the end of the i-th bubbling pass in the computation of RBUBBLE $(\alpha, 1, n, k)$, the last $(k+1)i$ items are k-sorted with respect to the whole sequence.

If $(k+1)i>n$, the last $(k+1)i$ items in $P(i)$ should be read as the whole sequence. We prove that $P(i)$ is true for all i, $1 \le i \le \lceil (n-k-1)/(k+1) \rceil$, by induction on i.

Basis: $i=1$. From Lemma 3.1, $P(1)$ is true.

Induction step: $i \ge 1$. Suppose that $P(i)$ holds and $i < \lceil (n-k-1)/(k+1) \rceil$. From the definition of k-BUBBLE, during the $i+1$st bubbling pass any item in $\alpha_{n-(k+1)(i+1)+1\ldots n-(k+1)i}$ does not move unless it is smaller than an item in $\alpha_{1\ldots n-(k+1)(i+1)}$. Therefore, the items in $\alpha_{n-(k+1)i+1\ldots n}$ remain k-sorted throughout the $i+1$st bubbling pass. From Lemma 3.1, at the end of the $i+1$st bubbling pass the items in $\alpha_{n-(k+1)(i+1)+1\ldots n-(k+1)i}$ are k-sorted with respect to $\alpha_{1\ldots n-(k+1)i}$. Hence, at the end of the $i+1$st bubbling pass the last $(k+1)(i+1)$ items in α are k-sorted. Thus, the lemma holds. □

We evaluate the computing time of RBUBBLE as the number of calls of k-BUBBLE. By a simple calculation we obtain the next lemma.

Lemma 3.3 The number of calls of k-BUBBLE to k-sort $\alpha=(a_1 \ldots, a_n)$ by RBUBBLE is $\lfloor (n-k-1)/(k+1) \rfloor (n-2k+r)/2+(1-\delta(r))$, where $r=n$ modulo $(k+1)$, and $\delta(r)=1$ if $r=0$ and 0 otherwise (the Kroenecker delta).

Corollary 3.4 When n is a multiple of $k+1$, the number of calls of k-BUBBLE to k-sort $\alpha=(a_1, \ldots, a_n)$ by RBUBBLE is $(n-k-1)(n-2k)/(2k+2)$.

We next evaluate the computing time of RBUBBLE in terms of the number of comparison-exchange operations instead of the number of calls of k-BUBBLE. Consider the sorting of a reverse ordered sequence of length n by bubble sorting. The sequence is initially $(n-1)$-sorted. Each bubbling pass reduces the unsorted size of the sequence by one. Therefore, in the worst case $n-k-1$ bubbling passes are required to obtain a k-sorted sequence from a given sequence of length n. Thus, this straightforward method for k-sorting takes $(n-k-1)(n+2)/2$ comparison-exchanges.

From Lemma 3.3 the next theorem is immediate.

Theorem 3.5 The number of comparison-exchanges to k-sort a sequence of length n by RBUBBLE is $g(k+1)+k(k+1)\lceil (n-k-1)/(k+1) \rceil /2$, where g is the number of calls of k-BUBBLE given in Lemma 3.3 (i.e., $g=\lfloor (n-k-1)/(k+1) \rfloor (n-2k+r)/2+(1-\delta(r))$).

Corollary 3.6 When n is a multiple of $k+1$, the number of comparison-exchanges to k-sort a sequence of length n by RBUBBLE is $(n-k-1)(n-k)/2$.

From Corollary 3.6 we can say that RBUBBLE is faster than the straightforward method (i.e., k-sorting by bubble sorting) by $k(n-k-1)$ comparison-exchanges. This is a pleasant result. For large k, if we use an $\mathbf{O}(k)$ partition algorithm for k-sorting $\alpha_{1\ldots 2k+2}$ in k-BUBBLE, the computing time given in Theorem 3.5 can be marginally improved. However, the purpose of this section is to study a generalization of Bubblesort, but not to design a fast k-sorting algorithm. In Section 5, a fast and optimal k-sorting algorithm for k satisfying a certain condition is given.

4. Parallel Implementation of RBUBBLE

In this section we consider how RBUBBLE can be implemented by a parallel compute. Our model of a parallel computer consists of a number of processors and a global array memory. The processors can read from or write into components of the array memory concurrently unless they access the same component. Each processor operates independently of the others, but is synchronized by a global clock. We assume that each item of a given sequence of length n is initially stored in the corresponding component of the array.

We consider that the computing time for the compare-exchange is one time unit. Then the computing time for k-BUBBLE $(\alpha, j, j+2k+1)$ is $k+1$ time units if $j \ge 2$, whreras the computing time for k-BUBBLE $(\alpha, 1, 2k+2)$ is $(k+1)(k+2)/2$. Thus the first k-bubbling pass of RBUBBLE takes $(k+1)(k+2)/2+(n-2k-2)(k+1)$ time units. To achieve simplicy in our anaylysis of the computing time and the timing for parallel implementation, we assume that k-BUBBLE $(\alpha, 1, 2k+2)$ takes $(k+1)^2$ time units instead of $(k+1)(k+2)/2$ time units. For example, we consider that the first k-bubbling pass of RBUBBLE takes $(k+1)^2+(n-2k-2)(k+1)=(k+1)(n-k-1)$ time units.

In our parallel implementation, the first processor is used for processing the first k-bubbling pass, the second processor is used for the second k-bubbling pass, and so

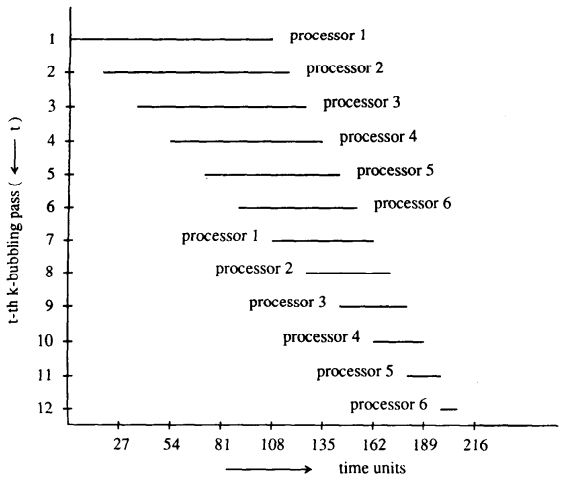

Fig. 2 Timing of the RBUBBLE computation for $n=39$ and $k=2$.

on. To start the second k-bubbling pass, the process in the first k-bubbling pass involving the items from the first position to the $(2k+2)$ th position should be completed. Since $2(k+1)^2 \geq (k+1)(k+2)/2+(k+1)^2$, a time delay of $2(k+1)^2$ time units is sufficient for starting the second k-bubbling pass. In general, there is no timing problem if the $i+1$ st k-bubbling pass starts at $2(k+1)^2$ time units after the i-th k-bubbling pass started. In this fashion, we can implement RBUBBLE in parallel on the model with $\lceil (n-k-1)/(2k+2) \rceil$ processors. The timing of the parallel RBUBBLE computation for $n=39$ and $k=2$ is shown in Fig. 2. Such a parallel implementation can be described as follows:

```
procedure PRBUBBLE (α, 1, n, k);
begin
  for i:=1 to ⌈(n-k-1)/(k+1)⌉ do in parallel
    start the (i modulo ⌈(n-k-1)/(2k+2)⌉)-th
    processor at 2(k+1)²(i-1) clock time;
    for j:=1 to n-k-(k+1)i do
      k-BUBBLE (α, j, j+2k+1) by (i modulo
        ⌈(n-k-1)/(2k+2)⌉-th processor
    {if n-k-(k+1)i<1 then k-BUBBLE is ex-
     ecuted once in the for loop}
    end
end
```

By simple calculation we can evaluate the computing time of the parallel version of RBUBBLE (i.e., PRBUBBLE) in the next theorem.

Theorem 4.1 For n and k such that $n>k+1$, the computing time of PRBUBBLE for k-sorting a sequence of length n is $(n-k-1)(k+1)+\lfloor (n-2k-2)/(k+1) \rfloor (k+1)^2+(n \text{ modulo } k+1)(k+1)$ time units, where the computing time for the compare-exchange operation is the time unit.

Corollary 4.2 When n is a multiple of $k+1$, the computing time of PRBUBBLE for k-sorting a sequence of length n is $(2n-3k-3)(k+1)$ time units, where the computing time for the compare-exchange operation is the time unit.

5. *k*-Quicksort

Quicksort is a divide-and-conquer method for sorting. It works by partitioning a file into two parts and then sorting those parts recursively in the same way. This method can be also used for k-sorting.

Let PARTITION (α, b, γ) be a procedure for finding the median m of and constructing a partition (β, γ) of α by m (that is, any item in $\beta \leq m \leq$ any item in γ), where we assume that $||\beta|-|\gamma|| \leq 1$.

```
procedure RQUICK (α, 1, |α|, k);
begin
  if |α| ≤ k+1 then return
  else begin
    PARTITION (α, β, γ);
    return RQUICK (β, 1, |β|, k) followed by
           RQUICK (γ, 1, |γ|, k)
  end
end
```

The above procedure RQUICK can be considered a generalization of Quicksort. Since the necessary and sufficient number of comparison-exchanges for finding the median among n items is $\Theta(n)$ [3], the next theorem is immediate.

Theorem 5.1 Let α be a sequence of n items. Then α can be k-sorted by RQUICK $(\alpha, 1, |\alpha|, k)$ and its computing time is $O(n \log (n/(k+1)))$ time units.

As stated in the above theorem, RQUICK takes $O(n \log (n/(k+1)))$ time even in the worst case. However, the constant factor and the effect of minor terms associated with the linear algorithm for finding the median are relatively large for practical-sized n This results in RQUICK being much worse on average than a rough quicksorting associated with randomly choosing the pivot for the partition each time instead of chooosing the median, although the latter method takes $\mathbf{O}(n^2)$ time in the worst case.

Let k be a function of n, and denote it by $k(n)$, If $\lim_{n\to\infty} k(n)/n=1$, then α can be k-sorted by simply PARTITION $(\alpha_{1\ldots n-k+1}\alpha_{k-1\ldots n}, \alpha_{1\ldots n-k+1}, \alpha_{k-1\ldots n})$ in $O(n-k(n))$ time. In this case RQUICK is not time-optimal, since the order of $n-k(n)$ is less than linear in n. For example, when $k(n)=n-\sqrt{n}$, α can be k-sorted in $O(\sqrt{n})$ time by the method mentioned above.

Theorem 5.2 RQUICK $(\alpha, 1, |\alpha|, k)$ is time-optimal for k-sorting within a constant factor, if $\lim_{n\to\infty} k(n)/n<1$ and if one of the following three conditions is satisfied:

(1) For some positive constant c_1, $c_1<\lim_{n\to\infty} k(n)/n$.

(2) $\lim_{n\to\infty} \log k(n)/\log n=0$.

(3) For some positive constant c_2, $c_2 < \lim_{n\to\infty} \log k(n)/\log n < 1$.

Proof. We assume that $\lim_{n\to\infty} k(n)/n < 1$.

Case 1. There exists a positive constant c_1 such that $c_1 < \lim_{n\to\infty} k(n)/n$.

In this case RQUICK takes $\mathbf{O}(n)$ time, from Theorem 5.1. On the other hand, to k-sort α, each item in $\alpha_{1\,..\,n-k+1}$ or in $\alpha_{k-1\,..\,n}$ must be compared with another item at least once. It theorefore takes at least $\Theta(n)$ time by any comparison-based k-sorting algorithm.

Case 2. $\lim_{n\to\infty} \log k(n)/\log n = 0$.

In this case RQUICK takes $O(n \log n)$ time from Theorem 5.1. The optimal number of comparisons needed to sort a sequence of n items is $\Theta(n \log n)$ [1], and the optimal number of comparisons needed to sort a k-sorted sequence of n items is $\Theta(n \log (k+1))$ [5, 7]. Let $T(n)$ be the computing time for k-sorting a sequence of n items. Then from those facts $\Theta(n \log n) < T(n) + \Theta(n \log (k+1))$. Since $\lim_{n\to\infty} \log k(n)/\log n = 0$, $T(n) \geq \Theta(n \log n)$.

Case 3. There exists a positive constant c_2 such that $c_2 < \lim_{n\to\infty} \log k(n)/\log n < 1$. In this case RQUICK takes $O(n \log n)$ time. Suppose that in this case RQUICK is not time-optimal. Then there exists an algorithm for k-sorting in $T(n)$ time such that $\lim_{n\to\infty} T(n)/(n \log n) = 0$. We can construct the following sorting algorithm. We first k-sort a given sequence by the above mentioned type of $T(n)$-time algorithm. Let the obtained k-sorted sequence be $\alpha_{1\,..\,n} = A_1 A_2 \ldots A_t$, where for each i $|A_i| = k$. For simplicity of description we assume that n is a multiple of k, but with a minor modification the algorithm will also work properly for any n. Next we use a linear time partition algorithm as follows:

for $i := 1$ **to** $t-1$
  PARTITION $(A_i A_{i+1}, A_i, A_{i+1})$

Then for the obtained sequence $\alpha_{1\,..\,n} = A_1 A_2 \ldots A_t$, any item in A_i is not greater than any item in A_j if $i < j$. Finally sort each A_i by a sorting algorithm whose computing time is asymptotically not greater than $c_3 n \log n$. Then final sequence is completely sorted.

Let c_4 be a constant such that $\lim_{n\to\infty} \log k(n)/\log n < c_4 < 1$. Then the total computing time for sorting by the method mentioned above is not asymptotically greater than

$$(n/k(n)) \times c_3 k(n) \log k(n) + T(n) + cn \leq c_3 c_4 n \log n + T(n) + cn.$$

Since $T(n)$ and cn are minor terms, this means that there exists an asymptotically faster sorting algorithm with a smaller constant factor of the leading term in its computing time than any comparison-based sorting algorithm. However, since the optimal time for comparison-based sorting is $\Theta(n \log n)$, there should exist a positive constant c_5 such that the number of comparisons needed to sort by any algorithm is greater than $c_5 n \log n$. From this contradiction we can conclude that RQUICK is time-optimal within a constant factor. □

For k such that $\lim_{n\to\infty} k(n)/n = 0$ and $\lim_{n\to\infty} k(n) \log n = 1$, we do not know whether RQUICK $(\alpha, 1, |\alpha|, k)$ is time-optimal. For example, $k(n) = n/\log n$ satisfies the above conditions. We conjecture that in this case RQUICK is also optimal.

If a given sequence is already roughly sorted, we may skip an initial part of the recursion for partitioning in RQUICK. In this fashion we can improve the computing time for k-sorting rougly sorted sequences.

procedure ADJUST $(\alpha, 1, n)$:
{Assume $\alpha = (a_1, \ldots, a_n)$. For simplicity we assume that n is a multiple of 2ROUGH (α). If α does not satisfy this condition, the procedure needs a minor modification.}

begin
  $p := \text{ROUGH}(\alpha)$;
  $r := n/p$;
  for $i := 1$ **to** r **do begin**
    $\alpha_i := \alpha_{p(i-1)+1\,..\,pi}$;
    PARTITION $(\alpha_i, \alpha_{i_1}\, \alpha_{i_2})$
  end;
  for $i := 1$ **to** $r-1$ **do begin**
    $\beta_i := \alpha_{i_2}\alpha_{(i+1)_1}$;
    PARTITION $(\beta_i, \beta_{i_1}, \beta_{i_2})$
  end;
  return $\alpha_{1_1}\beta_{1_1}\beta_{1_2} \ldots \beta_{(r-1)_1}\beta_{(r-1)_2}\alpha_{r_2}$
end

Lemma 5.3 Let α be a p-sorted sequence of length n. Then ADJUST $(\alpha, 1, n)$ returns a sequence $\gamma = (c_1, \ldots, c_n)$ satisfying the following conditions:

(1) γ is a permutation of α.

(2) Any item of $(c_{p(i-1)+1}, \ldots, c_{ip})$ is not smaller than any item of $(c_{p(i-2)+1}, \ldots, c_{p(i-1)})$ for each $i (2 \leq i \leq n/p)$ and is not greater than any item of $(c_{pi+1}, \ldots, c_{p(i+1)})$ for each $i (1 \leq i \leq n/p - 1)$.

Proof. It is obvious that γ is a permutation of α. We use the following notation. For x and y, a pair of sequences, $x \leq y$ means that any item in x is not greater than any item in y. Since α is initially p-sorted, the following inequalities hold for each i:

$$\alpha_{1_1} \ldots \alpha_{(i-2)_2} \leq \alpha_{i_1} \leq \alpha_{(i+1)_2} \ldots \alpha_{r_2},$$

$$\alpha_{1_1} \ldots \alpha_{(i-1)_1} \leq \alpha_{i_2} \leq \alpha_{(i+2)_1} \ldots \alpha_{r_2}.$$

After the execution of the first PARTITION in ADJUST, for each i the above inequalities hold and $\alpha_{i_1} \leq \alpha_{i_2}$. After the execution of the second PARTITION in ADJUST, for each i the following inequalities hold:

$$\alpha_{1_1}\beta_{1_1} \ldots \beta_{(i-1)_1} \leq \beta_{(i-1)_2} \leq \beta_{i_2} \ldots \beta_{(r-1)_2}\alpha_{r_2},$$

$$\alpha_{1_1}\beta_{1_1} \ldots \beta_{(i-1)_1} \leq \beta_{i_1} \leq \beta_{i_2} \ldots \beta_{(r-1)_2}\alpha_{r_2}.$$

Therefore, the lemma holds (a computing process by ADJUST $(\alpha, 1, n)$ is depicted in Fig. 3). □

procedure FRQUICK $(\alpha, 1, n, k)$;
{Assume that $\alpha = (a_1, \ldots, a_n)$ and that n is a multiple of 2 ROUGH (α). If α does not satisfy this condition, the procedure needs a minor modification.}

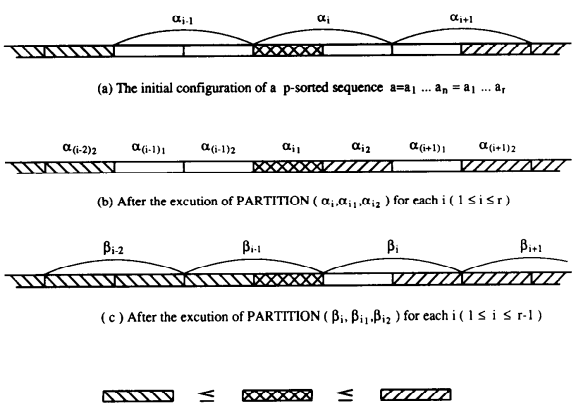

Fig. 3 Computation process using ADJUST (α, n).

```
begin
  if n≤k+1 then return
  else begin
    P:=ROUGH (α);
    χ:=ADJUST (α, 1, n)
    let χ:=b1 . . . bn;
    for i:=1 to n/p do begin
      χi:=bp(i-1)+1 . . . bpi;
      PARTITION (χi, βi, γi);
      return RQUICK (βi, 1, |βi|,k) RQUICK (γi,
        1, |γi|, k) to χi
    end;
    return χ1 . . . χn/p
  end
end
```

Theorem 5.4 Let α be a sequence of n items. Then α can be k-sorted by FRQUICK (α, 1, n, k) in $O(n \log \lceil (\mathrm{ROUGH}(\alpha)+1)/(k+1) \rceil)$ time.

Proof. From Lemma 5.3 the correctness of FRQUICK is immediate. The computing time of PARTITION is $O(n)$, and it has been shown that ROUGH (α) can be computed in $O(n)$ time [2]. Hence the computing time of ADJUST (α, 1, n) is $O(n)$. From Theorem 5.1 the computing time of the **for** loop in FRQUICK is $O(n \log \lceil (\mathrm{ROUGH}(\alpha+1)/(k+1) \rceil)$. Therefore, the lemma holds. □

The reason we use ROUGH (α)+1 instead of ROUGH (α) in Theorem 5.4 is to avoid the case for $\log \lceil \mathrm{ROUGH}(\alpha)/(k+1) \rceil = 0$. The proof of the next theorem is analogous to that of Theorem 5.2.

Theorem 5.5 FRQUICK (α, 1, n, k) is time-optimal for k-sorting ROUGH (α)-sorted sequences within a constant factor, if $\lim_{n\to\infty} k(n)/\mathrm{ROUGH}(\alpha) < 1$ and if one of the following three conditions is satisfied:

(1) For some positive constant c_1, $c_1 < \lim_{n\to\infty} k(n)/\mathrm{ROUGH}(\alpha)$.

(2) $\lim_{n\to\infty} \log k(n)/\log \mathrm{ROUGH}(\alpha) = 0$.

(3) For some positive constant c_2, $c_2 < \lim_{n\to\infty} \log k(n)/\log \mathrm{ROUGH}(\alpha) < 1$.

We can implement RQUICK in parallel. After the execution of PARTITION(α, β, γ) in RQUICK(α, 1, $|\alpha|$, k), RQUICK(β, 1, $|\beta|$, k), and RQUICK(γ, 1, $|\gamma|$, k) can be executed in parallel if we have an appropriate parallel computer. If such parallel executions are always possible after the partitions, RQUICK(α, 1, n, k) can be implemented in $O(f(n) \log (n/(k+1)))$ time, where $f(n)$ denotes the computing time needed to find the median of n items by the parallel model. In a similar way, we can also implement FRQUICK in parallel. We do not explain here the implementations in detail, since they depend to a large extent on the parallel model.

6. Concluding Remarks

We have designed k-Bubblesort and k-Quicksort as generalizations of Bubblesort and Quicksort, respectively. Sorting roughly sorted sequences and related problems are also important [2, 5, 6, 7]. Other interesting problems are the design and analysis of rough sorting versions of other well-known sorting algorithms, the design of an efficient algorithm for merging two k_1-sorted sequences into a k_2-sorted sequence, and the trade-off between the computing time and (k_1, k_2) of the the merging algorithm. These problems are worth further investigation.

Acknowledgements

Part of the work was carried out while the first author was a visitor in the Department of Computer Science, University of Waterloo. This visit was supported by Japan Society for the Promotion of Science, and the Natural Sciences and Engineering Research Council of Canada. The work of the second author was supported under a Natural Sciences and Engineering Research Council of Canada, Grant No. A-5692. The first author wishes to thank Professor Maarten H. van Emden for useful discussion of this work during his stay in Waterloo.

References

1. AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Mass., 1974.
2. ALTMAN, T. and IGARASHI, Y. Roughly Sorting: Sequential and Parallel Approach, *Journal of Information Processing*, **12** (1989), 154-158.
3. BLUM, M., FLOYD, R. W., PRATT, V. R., RIVEST, R. L. and TARJAN, R. E. Time Bounds for Selection, *J. Comput. Syst. Sci.* **7** (1972), 448-461.
4. COOK, C. R. and KIM, D. J. Best Sorting Algorithm for Nearly Sorted Lists, *Comm. ACM*, **23** (1980), 620-624.
5. ESTIVILL-CASTRO, V. and WOOD, D. A New Measure of Presortedness, *Tech. Report CS-87-58, Dept. of Computer Science, Univ. of Waterloo*, 1987.
6. IGASASHI, Y. and KORTELAINEN, J. Time Lower Bounds for Sorting Roughly Sorted Sequences, Tech. Report COMP89-19, IEICE (1989), 21-27.
7. IGARASHI, Y. and WOOD, D. Roughly Sorting: A Generalization of Sorting, *Tech. Report COMP87-20, IECEJ* (1987), 11-19.

8. LEVCOPOULOS, C. and PETERSSON, O. Heapsort-Adapted for Presorted Files, Workshop on Algorithms and Data Structures, Ottawa, *Lecture Notes in Computer Science*, **382**, Springer, Berlin (1989), 499–509.
9. MANNILA, H. Measures of Presortedness and Optimal Sorting Algorithms, **IEEE Trans. Comput**. CS-34 (1985), 318–325.
10. MEHLHORN, K. Sorting Presorted Files, 4-th GI Conf. on Theory of Computer Science, Aachen, *Lecture Notes in Computer Science*, **67** , Springer, Berlin (1979), 199–212.
11. SADO, K. and IGARASHI, Y. A Divide-and-Conquer Method of the Parallel Sort, *Tech. Report AL84-68, IECEJ* (1985), 41–50.
12. SADO, K. and IGARASHI, Y. A Fast Parallel Pseudo-Merge Sort Algorithm, *Tech Report AL85-16, IECEJ* (1985), 21–30.

(Received June 5, 1989; revised February 20, 1990)