Permuting In Place*

Faith E. Fich
Department of Computer Science
University of Toronto
Toronto, Ontario
Canada M5S 1A4

J. Ian Munro
Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

Patricio V. Poblete
Departamento de Ciencias de la Computación
Universidad de Chile
Casilla 2777
Santiago, Chile

Abstract

We address the fundamental problem of permuting the elements of an array of n elements according to some given permutation. Our goal is to perform the permutation quickly using only a polylogarithmic number of bits of extra storage. The main result is an algorithm whose worst case running time is $O(n \log n)$ and that uses $O(\log n)$ additional $\log n$-bit words of memory. A simpler method is presented for the case in which both the permutation and its inverse can be computed at (amortised) unit cost. This algorithm requires $O(n \log n)$ time and $O(1)$ words in the worst case. These results are extended to the situation in which we are to apply a power of the permutation. A linear time, $O(1)$ word method is presented for the special case in which the data values are all distinct and are either initially in sorted order or will be when permuted.

1. Introduction

Given an array $A[1..n]$ and a permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, we wish to rearrange the elements of the array in place, according to the permutation. More precisely, the rearrangement consists of performing the equivalent of the n *simultaneous* assignments

$$A[\pi(i)] \leftarrow A[i] \text{ for } i \in \{1, \ldots, n\}.$$

*This research was supported in part by the Natural Science and Engineering Research Council of Canada under grant numbers A-8237 and A-9176, the Information Technology Research Centre of Ontario, and the Chilean FONDECYT under grant numbers 90-1263 and 91-1252.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

8	4	12	2	6	10	14	1	3	5	7	9	11	13	15

4	8	12	1	2	3	5	6	7	9	10	11	13	14	15

Figure 1: Three arrangements of a list of length 15.

Thus if the array A originally contains the sequence of values $a_1, a_2, \ldots, a_n$, then, afterwards, it contains the sequence $a_{\pi^{-1}(1)}, a_{\pi^{-1}(2)}, \ldots, a_{\pi^{-1}(n)}$.

By in place we mean that the algorithm performs the rearrangement by repeatedly interchanging pairs of elements. Hence, the set of values in the array and the number of times each occurs always remain the same. In particular, this definition precludes moving the elements into an auxiliary array and then putting each element, one at a time, into its correct location in the original array. It also means that the array cannot be padded with blanks (see, for example, [7]) to make it a more convenient size to work with.

The problem of rearranging data arises in a variety of situations. Some examples are transposing a rectangular matrix [12, ex. 1.3.3–12], [7], rotating a bit map image, and exchanging two sections of an array [2, p. 134].

If we are sorting a list with very large records, it might be more efficient to manipulate pointers to the records rather than the records themselves and, afterwards, put each record into its appropriate location [13, p. 74]. Another way to sort [13, p. 76] is to first compare every pair of keys, then, for each record, count the number of other records that should precede it and rearrange the records accordingly.

In certain circumstances, searching is *not* most efficient when the elements of the array are in sorted order. If a large number of searches are going to be performed, it could be advantageous to rearrange a sorted array first. For example, one can simplify the index computation in a binary search by organizing the elements into a data structure that implicitly represents a complete binary search tree (via a breadth first, left to right, enumeration of its nodes) [1, ex. 12–6, pp. 136, 183–184], [13, ex. 6.2.1–24, pp. 422, 670], [12, p. 401]. Like a heap, an element in location $i < n/2$ has its left and right children in locations $2i$ and $2i + 1$, respectively. When the data is stored on a disk with each block capable of holding b elements or in a virtual memory in which each page can contain b elements [13, pp. 472–473], the number of reads can be minimized by arranging the elements using a similar implicit representation of a complete $(b+1)$-ary search tree [12, p. 401]. These three different arrangements are illustrated in Figure 1. In the third case, $b = 3$.

We are interested in the amount of time and additional space needed to rearrange the elements of the array according to the input permutation. Previously known algorithms for this problem used either quadratic time or a linear amount of additional space in the worst case. In this paper, we present a concise algorithm that takes time $O(n \log n)$ and uses only $O(\log^2 n)$ bits of additional storage. We have a simpler method for the case in which both the permutation and its inverse are given that takes the same amount of time and uses only

i	1	2	3	4	5	6	7	8	9	10
$\pi(i)$	3	9	4	2	1	6	8	7	5	10

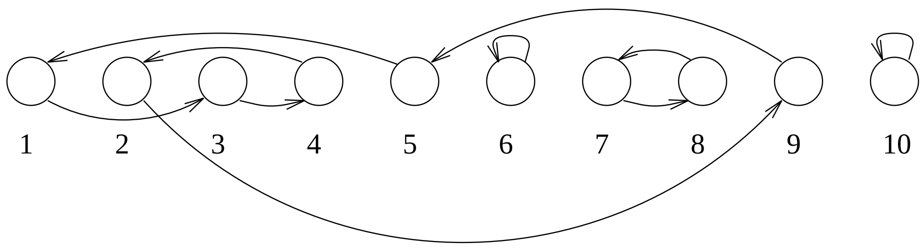

Figure 2: A graphical representation of a permutation.

$O(\log n)$ bits. Furthermore, we show how to use the fact that a permutation is known to rearrange the array either to or from sorted order to obtain an algorithm that takes time $O(n)$ and uses only $O(\log n)$ bits of additional storage.

Throughout the paper, we assume that the permutation π is given by means of an oracle. This models the situation where the value of $\pi(i)$ is computed from i (e.g. transposing a rectangular array) or the permutation cannot be changed (e.g. because it is being used by other processes). If the permutation is given by an array and its entries can be used to record information as the algorithm proceeds (perhaps destroying the permutation in the process), data rearrangement can be done efficiently, using $O(n)$ time and $O(\log n)$ additional bits of storage [13, ex. 5.2–10, pp. 80, 595]. This is also the case when the permutation is given as a product of disjoint cycles.

2. Cycle Leader Algorithms

The cycle structure of the permutation can be exploited to obtain efficient algorithms for rearranging data. Permutations are composed of one or more disjoint cycles, as illustrated in Figure 2. The arrows follow the direction $i \to \pi(i)$, indicating the direction in which the data should flow.

A basic operation is *ROTATE* which, starting from some designated location in a cycle, moves the values from one location of the cycle to the next. We call this designated location the *cycle leader*. For example, the cycle leader could be the smallest location in the cycle. Once the cycle leader has been identified, *ROTATE* can be performed by proceeding through the cycle, starting with its cycle leader and exchanging the value located at the cycle leader with the values in the successive locations of the cycle. Each of these other values is involved in exactly one exchange (that takes it to its correct final location). Thus the time taken by such an algorithm is proportional to n, the length of the array, plus the amount of time it takes to find all the cycle leaders.

If the permutation π is given as a product of disjoint cycles, identifying a leader for each

procedure ROTATE$(leader)$
$i \leftarrow \pi(leader)$
while $i \neq leader$ **do**
    interchange the values in $A[leader]$ and $A[i]$
    $i \leftarrow \pi(i)$

Figure 3: Rotating the values in a cycle.

cycle is very straightforward: merely take the first element in each cycle. The problem is more interesting when π is given as a mapping from the elements $1, \ldots, n$. One way to find all the cycle leaders in this case is to consider the locations $1, \ldots, n$ one at at time and, for each, determine whether it is a cycle leader.

If an additional, initially empty, bit vector of length n is available, it is easy to determine whether a location is the smallest element in its cycle in constant time. Specifically, when the value in an array location is moved, the bit corresponding to that location is set to 1. Since the locations are considered from smallest to largest, a location under consideration will have its corresponding bit equal to 0 exactly when it is a cycle leader [12, ex. 1.3.3–12(b), pp. 180, 517–518]. Similarly, if π is given as an array whose elements can be modified, the same effect can be achieved by setting $\pi(i)$ to i when the value in array location i is moved.

Determining whether a location is the smallest element in its cycle can also be accomplished using only a constant number of pointers into the array (each $\log_2 n$ bits long). Specifically, starting at the given location, proceed along the cycle until either the entire cycle has been traversed or a smaller location is encountered. In the first case, the given location is a cycle leader; otherwise it is not. The total amount of time to consider all n locations is $O(n^2)$. The worst case is achieved when $\pi = (1\ 2\ 3 \cdots n)$. For random permutations, an average of $O(n \log n)$ steps are performed [11], [13, p. 595].

These two ideas can be combined into the algorithm illustrated in Figure 4.

Theorem 1. *In the worst case, permuting an array of length n, given the permutation, can be done in $O(n^2/b)$ time and $b + O(\log n)$ bits of auxiliary space (consisting of a bit vector of length b plus a constant number of pointers), for $b \leq n$.*

The array A is conceptually divided into $\lceil n/b \rceil$ regions. Each region has size b, except for the last region, which might be smaller. The bit vector V is used to keep track of which locations in the region are encountered as the region is processed. If the location under consideration for being a cycle leader has a corresponding bit with value 0, its cycle is traversed until a smaller location is encountered. If no smaller location is encountered, then the location is a cycle leader and the cycle is rotated. Furthermore, if the location under consideration has a corresponding bit with value 1, then the location was previously encountered as part of a cycle containing some smaller location in the region and, hence, it is not a cycle leader. Theorem 1 follows from these observations.

If both π and π^{-1} are available, then it is possible to get a more efficient algorithm. Determining whether a given location is the smallest element in its cycle can be done by

```
for k ← 1 to ⌈n/b⌉ do
    s ← (k − 1)b
    if k ≤ ⌊n/b⌋ then l ← b
                 else l ← n − b⌊n/b⌋
    for i ← 1 to l do
        V[i] ← 0
    for i ← 1 to l do
        if V[i] = 0 then V[i] ← 1
                         j ← π(s + i)
                         while j > s + i do
                             if j ≤ s + l then V[j − s] ← 1
                             j ← π(j)
                         if j = s + i then ROTATE(s + i)
```

Figure 4: Rearranging an array using a bit vector V of length b.

starting at the location and proceeding alternatively forwards and backwards along the cycle until either the entire cycle has been traversed or a smaller location is encountered. (See Figure 5.)

Theorem 2. *In the worst case, permuting an array of length n, given the permutation and its inverse, can be done in $O(n \log n)$ time and $O(\log n)$ additional bits of storage.*

The analysis is similar to the bidirectional distributed algorithm for finding the smallest element in a ring of processors [8]. Specifically, the algorithm cannot determine whether i is a cycle leader after proceeding t steps forwards and t steps backwards along the cycle starting from i, only if the t elements following i and the t elements preceding i in its cycle are all larger than t. Furthermore, this will be the case for at most $1/t$ of the choices for i.

It is interesting to compare this algorithm to the $O(n^2)$ time $O(\log n)$ space algorithm mentioned above. Their expected behaviour on a random permutation is identical. However, the algorithm in Figure 5 eliminates the bad cases.

It is not necessary that the cycle leader be the smallest (or largest) location in the cycle. One approach is to apply a hash function to the elements of the permutation and take as cycle leader the location in the cycle that hashes to the smallest value. Starting at a given location, the algorithm proceeds along the cycle until either the entire cycle has been traversed or another location that hashes to a smaller value is encountered. If the hash function is randomly chosen from among a set of 5-wise independent hash functions, then this results in a randomized algorithm using $O(\log n)$ space and expected time $O(n \log n)$ for every input permutation [9].

It was not clear to us whether it was possible to have a deterministic algorithm that used $n \log^{O(1)} n$ time and $\log^{O(1)} n$ space without having the inverse permutation available. However, using a rather different approach, we were able to devise such an algorithm.

```
for i ∈ {1, ..., n} do
    j ← π(i)
    if j ≠ i then k ← π^{-1}(i)
                  while i < j and i < k do
                      if j = k then ROTATE(i)
                                    exit
                      j ← π(j)
                      if j = k then ROTATE(i)
                                    exit
                      k ← π^{-1}(k)
```

Figure 5: Rearranging an array using a permutation and its inverse.

Theorem 3. *In the worst case, permuting an array of length n, given the permutation, can be done in $O(n \log n)$ time and $O(\log^2 n)$ additional bits of storage.*

First consider the following characterization of the minimum locations in the cycles of the permutation π. Let $E_1 = \{1, \ldots, n\}$ and $\pi_1 = \pi$. For $r > 1$, we inductively define $E_r \subseteq E_{r-1}$ to be the set of *local minima* encountered following the permutation π_{r-1}, that is, $E_r = \{i \in E_{r-1} \mid \pi_{r-1}^{-1}(i) > i < \pi_{r-1}(i)\}$. We also define $\pi_r : E_r \to E_r$ to be the permutation that maps each element of E_r to the next element of E_r that is encountered following the permutation π_{r-1}. In other words, if $i \in E_r$, then $\pi_r(i) = \pi_{r-1}^m(i)$, where $m = \min\{m > 0 \mid \pi_{r-1}^m(i) \in E_r\}$, and $\pi_r(i) = \pi^M(i)$, where $M = \min\{M > 0 \mid \pi^M(i) \in E_r\}$. We call E_r the set of *order r elbows.*

For example, if $\pi = (1\ 5\ 3\ 6\ 10\ 4\ 2\ 9\ 8\ 7\ 11)$, as illustrated in Figure 6, then

$$
\begin{array}{ll}
E_1 = \{1,2,3,4,5,6,7,8,9,10,11\}, & \pi_1 = (1\ 5\ 3\ 6\ 10\ 4\ 2\ 9\ 8\ 7\ 11), \\
E_2 = \{1,2,3,7\}, & \pi_2 = (1\ 3\ 2\ 7), \\
E_3 = \{1,2\}, & \pi_3 = (1\ 2), \\
E_4 = \{1\}, \text{ and} & \pi_4 = (1).
\end{array}
$$

No more than half the elements in any cycle are local minima. Thus $|E_r| < |E_{r-1}|/2$. The minimum element of a cycle of π is always in E_r if the cycle contains more than one element of E_{r-1}. Hence the minimum element in a cycle is the unique elbow of maximum order in its cycle.

Computing the sets of elbows $E_1, E_2, \ldots$ and the corresponding permutations $\pi_1, \pi_2, \ldots$ can be done either in a top-down fashion, analogous to recursive descent parsing, or bottom-up, analogous to shift-reduce parsing. We explore both approaches in the two algorithms that follow.

The unidirectional nature of our oracle means that, given $i \in E_r$, it is easy to compute $\pi_r(i)$ but difficult to compute $\pi_r^{-1}(i)$. Thus determining whether i is a local minimum of π_r (i.e. whether $i \in E_{r+1}$) is difficult starting from i, but easy starting from i's predecessor in π_r. For example, in Figure 6, starting from 5 we can recognize that its successor, 3, is a local

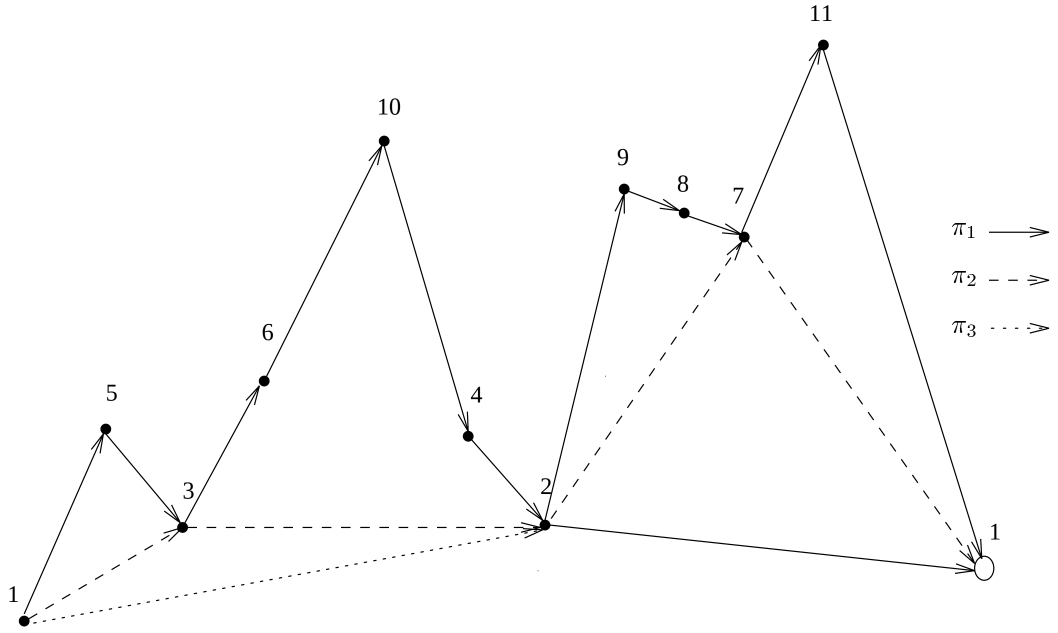

Figure 6: A cycle and its elbows.

minimum in π_1. In turn, starting from 3 we can recognize that 2, its successor in π_2, is a local minimum in π_2.

For each cycle, the algorithm in Figure 7 chooses the leader to be the unique element i in the cycle such that $\pi_{s-1} \cdots \pi_1(i) \in E_s$, where s is the maximum order of any element in the cycle. In other words, an element can be rejected as soon as it is seen not to be the element of maximum order. Dolev, Klawe, and Rodeh [4] and Peterson [14] independently discovered the usefulness of this choice of cycle leader for electing a leader in a unidirectional ring of processors using $O(n \log n)$ messages.

The procedure call $NEXT(r)$ is used to compute successive elements of π_r. It does this recursively, using successive elements of π_{r-1}.

For each $i \in \{1, \ldots, n\}$, the main program tests whether $\pi_1(i) \in E_2$ by comparing $\pi_1(i)$ with i and $\pi_1\pi_1(i)$; if so, it next tests whether $\pi_2\pi_1(i) \in E_3$ by comparing $\pi_2\pi_1(i)$ with $\pi_1(i)$ and $\pi_2\pi_2\pi_1(i)$; etc. One test is performed each each iteration of the inner for loop. Eventually, a value of r is found such that $\pi_{r-1} \cdots \pi_1(i) \in E_r$ but $\pi_r\pi_{r-1} \cdots pi_1(i) \notin E_r + 1$. There are three possible ways this can occur. One is when $\pi_r\pi_{r-1} \cdots \pi_1(i) = \pi_{r-1} \cdots \pi_1(i)$. Then, since π_r is a permutation, this implies that the cycle of π_r containing $\pi_{r-1} \cdots \pi_1(i)$ is trivial and thus $\pi_{r-1} \cdots \pi_1(i)$ is the minimum element in the cycle of π that contains it. In this case, i is a cycle leader. The other possible situations are when either $\pi_{r-1} \cdots \pi_1(i) < \pi_r\pi_{r-1} \cdots \pi_1(i)$ or $\pi_r\pi_{r-1} \cdots \pi_1(i) > \pi_r\pi_r\pi_{r-1} \cdots \pi_1(i)$. In both these cases, the algorithm detects that $\pi_r\pi_{r-1} \cdots \pi_1(i)$ is not the minimum element in its cycle and, thus, that i is not a cycle leader.

Using $O(\log^2 n)$ space, very little information about each permutation can be stored. We show that, essentially, only the most recently detected elbow of each order need be stored. The algorithm in Figure 7 stores this information in the array *elbow*. For $r > 1$, $elbow[r]$

```
procedure NEXT(r)
if r = 1 then elbow[0] ← π(elbow[1])
        else while elbow[r − 1] < elbow[r − 2] do
                    elbow[r − 1] ← elbow[r − 2]
                    NEXT(r − 1)
             while elbow[r − 1] > elbow[r − 2] do
                    elbow[r − 1] ← elbow[r − 2]
                    NEXT(r − 1)

return

for i ∈ {1, ..., n} do
      elbow[0] ← elbow[1] ← i
      for r ← 1, 2, ... do
            {loop invariant: elbow[r] = π_{r−1} ··· π_2 π_1(i) ∈ E_r}
            NEXT(r)
            if elbow[r] > elbow[r − 1]
            then  elbow[r] ← elbow[r − 1]
                  NEXT(r)
                  if elbow[r] > elbow[r − 1] then exit
                  elbow[r + 1] ← elbow[r]
            else  if elbow[r] = elbow[r − 1] then ROTATE(i)
                  exit
```

Figure 7: A recursive algorithm that rearranges an array using $O(n \log n)$ time and $O(\log^2 n)$ space.

is used to store an element of E_r. Immediately prior to a call to $NEXT(r)$, the elements satisfy the condition

$$elbow[r] = elbow[r-1] \xrightarrow{\pi_{r-1}} \cdots \xrightarrow{\pi_1} elbow[0]$$

and immediately afterwards, they satisfy the condition

$$elbow[r] \xrightarrow{\pi_r} elbow[r-1] \xrightarrow{\pi_{r-1}} \cdots \xrightarrow{\pi_1} elbow[0].$$

The procedure $NEXT(r)$ computes $\pi_r(elbow[r])$, leaving $elbow[r]$ unchanged and putting the result in $elbow[r-1]$ (and, of course, updating the earlier elements in the array).

If $r > 1$, $NEXT(r)$ recursively computes successive elements along π_{r-1}, starting from $elbow[r]$, until a local minimum is detected. Since $elbow[r] \in E_r$, $elbow[r] < \pi_{r-1}(elbow[r])$. Initially, $elbow[r-1] = elbow[r]$ and $elbow[r-2] = \pi_{r-1}(elbow[r])$; thus $elbow[r-1] < elbow[r-2]$. The first while loop in $NEXT(r)$ advances $elbow[r-1]$ (and $elbow[r-2], \ldots, elbow[0]$, recursively) as long as π_{r-1} is strictly increasing. The cycle of π_{r-1} containing $elbow[r]$ is not trivial; thus, when the first while loop terminates $elbow[r-1] >$

$elbow[r-2] = \pi_{r-1}(elbow[r-1])$. The second while loop continues advancing $elbow[r-1]$ (and $elbow[r-2], \ldots, elbow[0]$, recursively) as long as π_{r-1} is strictly decreasing. At the end of the second while loop, $elbow[r-1] < elbow[r-2] = \pi_{r-1}(elbow[r-1])$. Hence $elbow[r-1]$ is the next local minimum of π_{r-1}. In other words, $elbow[r-1] = \pi_r(elbow[r])$.

Suppose $i' = \pi_r\pi_{r-1}\cdots\pi_1(i) \in E_r - E_{r+1}$. At the beginning of the r'th iteration of the inner for loop of the main program, $elbow[r] = \pi_{r-1}\cdots\pi_1(i) = \pi_r^{-1}(i') \in E_r$. When the computation detects that $i' \notin E_{r+1}$, either $elbow[r] = \pi_r^{-1}(i')$ and $elbow[r-1] = i'$ or $elbow[r] = i'$ and $elbow[r-1] = \pi_r(i')$. Furthermore, $elbow[0] = \pi_1\cdots\pi_{r-1}(elbow[r-1])$. It is easy to prove by induction on r that if $j \in E_r$ and M is the smallest positive integer such that $\pi_1\cdots\pi_{r-1}(j) = \pi^M(j)$, then $\pi^L(j) \notin E_r$ for $1 \leq L \leq M$. Thus, starting from i' and proceeding along π, the element $\pi_1\cdots\pi_{r-1}\pi_r(i')$ will be reached before $\pi_r^2(i')$ Similarly, starting from i' and proceeding backwards along π, the element $i = \pi_1^{-1}\cdots\pi_{r-1}^{-1}\pi_r^{-1}(i')$ will be reached before $\pi_r^{-2}(i')$ Therefore testing whether i is a cycle leader involves proceeding along π in a subsegment of the region between $\pi_r^-2(i')$ and $\pi_r^2(i')$.

Considering all $i' \in E_r - E_{r+1}$, the permutation π is evaluated less than $4n$ times. (In fact, π is evaluated at most 4 times at each element.) Since E_r is empty for $r > \log_2 n$, the algorithm proceeds a total of $O(n \log n)$ steps along π.

Each call of $NEXT(1)$ proceeds one step along π. Each call of $NEXT(r)$, for $r > 1$, involves at least two recursive calls to $NEXT(r-1)$. The total amount of work performed by this call (excluding the work performed by the recursive calls) is proportional to the number of recursive calls it makes. For each i, every iteration of the inner for loop performs a constant amount of work (excluding the work performed by the subroutines it calls) and, except for possibly the last iteration, involves two calls to $NEXT$. Thus the total amount of work performed by the entire algorithm is proportional to the total number of steps the algorithm proceeds along π plus $O(n)$ steps to rotate all the cycles.

When π consists of a single cycle with the elements $\{0, \ldots, n-1\}$ ordered lexicographically with respect to the reverse of their $(\log n)$-bit binary representations (for example, $\pi = (0\ 4\ 2\ 6\ 1\ 5\ 3\ 7)$), the algorithm actually uses $\Omega(n \log n)$ steps. Hence the running time of the algorithm is in $\Theta(n \log n)$.

The algorithm uses $\Theta(\log n)$ variables, each capable of holding one element in $\{1, \ldots, n\}$. With care in implementation, only $O(\log n)$ bits are needed for representing the program stack. Thus a total of $\Theta(\log^2 n)$ bits of additional space are used by this algorithm.

An iterative, bottom up version of the algorithm in Figure 7 is given in Figure 8. As $elbow[0]$ is advanced along the cycle of π containing i, $elbow[r]$, for $r > 1$, records the most recent element of E_r that has been detected. Each time a local minimum along π_{r-1} is detected, $elbow[r]$ is advanced to that location. The variable $state[r]$ encodes information about the portion of the cycle of π_r being examined.

The first time an element of E_r is detected, $state[r]$ is set to GOT_ONE and $elbow[r] = \pi_{r-1}\cdots\pi_1(i)$. When the next element of E_r (along π_r) is detected, the algorithm can determine whether i's cycle contains only one element in E_r. If so, i is the leader of its cycle. Otherwise the algorithm sets $state[r]$ to GOT_TWO and tries to determine whether $\pi_r\pi_{r-1}\cdots\pi_1(i)$ is a local minimum of π_r (and hence an element of E_{r+1}).

If $state[r] = UP$, then $elbow[r]$ is known to be larger than $\pi_r^{-1}(elbow[r])$, its predecessor along π_r. Similarly, if $state[r] = DOWN$, then $elbow[r]$ is known to be smaller than

```
for i ∈ {1, . . . , n} do
        elbow[1] ← i
        state[1] ← GOT_ONE
        repeat
                r ← 1
                elbow[0] ← π(elbow[1])
                while state[r] = DOWN and elbow[r] < elbow[r − 1] do
                            state[r] ← UP
                            elbow[r] ← elbow[r − 1]
                            r ← r + 1
                case state[r]

                GOT_ONE: if elbow[r] > elbow[r − 1]
                            then  state[r] ← GOT_TWO
                                  elbow[r] ← elbow[r − 1]
                            else  if elbow[r] = elbow[r − 1] then ROTATE(i)
                                  exit

                GOT_TWO: if elbow[r] > elbow[r − 1] then exit
                            state[r + 1] ← GOT_ONE
                            elbow[r + 1] ← elbow[r]
                            state[r] ← UP
                            elbow[r] ← elbow[r − 1]

                UP:         if elbow[r] > elbow[r − 1] then state[r] ← DOWN
                            elbow[r] ← elbow[r − 1]

                DOWN:       elbow[r] ← elbow[r − 1]
```

Figure 8: An iterative algorithm that rearranges an array using $O(n \log n)$ time and $O(\log^2 n)$ space.

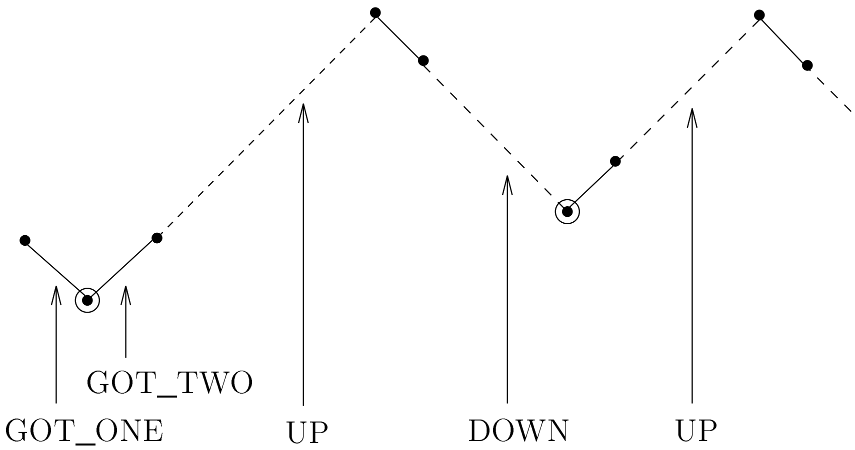

Figure 9: The states occurring in the iterative algorithm.

procedure $ROTATE\_BACKWARDS(leader)$
$j \leftarrow leader$
$i \leftarrow \pi(leader)$
while $i \neq leader$ **do**
    interchange the values in $A[j]$ and $A[i]$
    $j \leftarrow i$
    $i \leftarrow \pi(i)$

Figure 10: Rotating the values backwards in a cycle.

$\pi_r^{-1}(elbow[r])$, its predecessor along π_r. This information is sufficient to detect successive local minima along π_r. Figure 9 illustrates the states that occur as the algorithm proceeds along a cycle.

A small but natural twist on our problem is that we are given π, but required to move the elements according to π^{-1}. The cycle leader technique of Theorem 3 is still applicable. It is only necessary to replace ROTATE by the following procedure.

Clearly this step requires time linear in the size of the cycle, and $O(\log n)$ bits of memory. The problem is easily generalized to being given π (and perhaps π^{-1}) and asked to apply π^q, where q may depend on the (length of the) cycle. Either of the cycle leader methods above is applicable. We need only define a modified procedure $ROTATE(q, leader, cyclelength)$. This is essentially the same problem as transposing an array. The following is a translation into our terms of the method discussed in [2, p. 134]. The procedure $EXCHANGE(i, j, m)$ exchanges the m elements along π, starting at location i, with those starting at location j, assuming these segments are disjoint. Both i and j are advanced m locations along the cycle as a result of an execution of this procedure.

We see that, including calls to π, the *ROTATE* algorithm requires time proportional to the length of the cycle in question; hence we can strengthen Theorems 2 and 3.

Corollary 1. *In the worst case, permuting an array of length n according to the permutation*

```
procedure EXCHANGE(i, j, m)
repeat m times
        interchange the values in A[j] and A[i]
        i ← π(i)
        j ← π(j)

procedure ROTATE(q, leader, cyclelength)
irun ← q mod cyclelength
jrun ← cyclelength − irun
i ← leader
j ← leader
repeat irun times
        j ← π[j]
while irun ≠ jrun do
        if irun < jrun
        then temp ← i
                EXCHANGE(i, j, irun)
                i ← temp
                jrun ← jrun − irun
        else  temp ← i
                EXCHANGE(i, j, jrun)
                j ← temp
                irun ← irun − jrun
EXCHANGE(i, j, irun)
```

Figure 11: Rotating the values q steps in a cycle.

for $i \in \{1, \ldots, n\}$ **do**
  if $(\sigma(i) > \sigma(\pi(i)))$ **and** $(A[i] < A[\pi(i)])$
    then $\mathit{ROTATE}(i)$

Figure 12: Rearranging an array of distinct elements that satisfy a fixed total order.

π^q, given π and its inverse, can be done in $O(n \log n)$ time and $O(\log n)$ additional bits of storage.

Corollary 2. *In the worst case, permuting an array of length n according to the permutation π^q, given the permutation π, can be done in $O(n \log n)$ time and $O(\log^2 n)$ additional bits of storage.*

3. Permuting Data In and Out of Order

In many situations, it is known that the array elements satisfy a fixed total order, i.e. when the data values are taken in a particular order, they form a sorted sequence. Suppose the array A has been rearranged according to the permutation π and let σ be a permutation such that

$$A[\sigma^{-1}(1)] \leq A[\sigma^{-1}(2)] \leq \cdots \leq A[\sigma^{-1}(n)].$$

Then element $A[i]$ has the $\sigma(i)$'th smallest value, for $i = 1, \ldots, n$. In particular, if $\sigma(i) > \sigma(\pi(i))$ then $A[i] > A[\pi(i)]$. For example, if the permutation π rearranges the array A into sorted order, then σ is the identity permutation and if A was sorted before the rearrangement, then $\sigma = \pi^{-1}$.

Consider the rearrangement algorithm in Figure 12. In particular, if the permutation π rearranges the array A into sorted order, then the algorithm is simply looking for places where the permutation wants to move an element to a lower numbered location and the element is smaller than the element that is currently there.

Notice that, after the cycle containing location i has been rotated, $\sigma(i) > \sigma(\pi(i))$ implies $A[i] \geq A[\pi(i)]$. Thus each nontrivial cycle is rotated at most once.

Now suppose that no two consecutive elements along any nontrivial cycle have the same value, i.e. $i \neq \pi(i)$ implies $A[i] \neq A[\pi(i)]$ for all $i \in \{1, \ldots, n\}$. This condition is certainly true if the elements of the array A are distinct. We claim that before a given nontrivial cycle of the permutation π has been rotated, there is a location i in that cycle such that $\sigma(i) > \sigma(\pi(i))$ and $A[i] < A[\pi(i)]$. This implies that the cycle will be rotated at least once. Let i be any location in the cycle satisfying $\sigma(i) > \sigma(\pi(i)) < \sigma(\pi^2(i))$. (This will be the case, for example, when $\pi(i)$ is the location containing the smallest value in the cycle.) The cycle leader will be the first such location encountered. After the cycle has been rotated, $A[\pi(i)] \leq A[\pi^2(i)]$, since $\sigma(\pi(i)) < \sigma(\pi^2(i))$. Also, the values in locations $\pi(i)$ and $\pi^2(i)$ were initially in locations i and $\pi(i)$, respectively. Hence, before the rotation, $A[i] \leq A[\pi(i)]$. However, $A[i] \neq A[\pi(i)]$, so $A[i] < A[\pi(i)]$, as required.

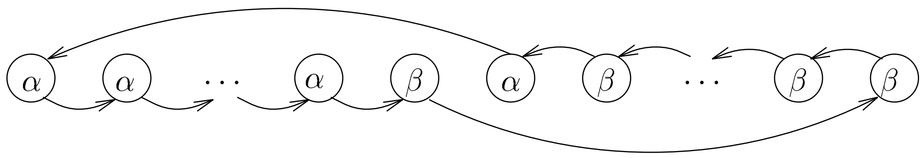

Figure 13: An example with runs of equal-valued elements.

for $i \in \{1, \ldots, n\}$ **do**
  if $A[i] \neq A[\pi(i)]$ **then**
    $j \leftarrow \pi(i)$
    while $A[j] = A[\pi(j)]$ **do** $j \leftarrow \pi(j)$
    if $(\sigma(\pi(i)) > \sigma(\pi(j)))$ **and** $(A[\pi(i)] < A[\pi(j)])$ **then** $ROTATE(i)$

Figure 14: Rearranging any array of elements that satisfy a fixed total order.

It is not necessary to rotate a cycle if all the elements it contains have the same value. However, there are other situations where a nontrivial cycle contains consecutive elements with the same value and the algorithm in Figure 12 does not rotate the cycle. The following example is constructed using two distinct values, α and β, where $\alpha < \beta$. It assumes the array is to be rearranged into sorted order. Except for the two middle elements, the array is already sorted, but since these two elements can be arbitrarily far apart in the cycle, no local test (such as the one in the algorithm in Figure 12) can succeed in detecting them.

Notice that a cycle containing runs of equal-valued elements has the same effect as the cycle taking the beginning of each run to the beginning of the next run. Since no consecutive elements of this latter cycle have the same value, the technique used by the algorithm in Figure 12 can be applied to identify a cycle leader.

If the functions π and $\sigma\pi$ can be evaluated in constant time (which is the case when π rearranges the array either into or out of sorted order), then the algorithm in Figure 14 runs in linear time. This is because the algorithm goes through the while loop at most once for each run of equal-valued elements in a cycle. In fact, the linear time bound holds also under a weaker assumption.

Theorem 4. *Suppose that after the array A is permuted according to the permutation π, it satisfies $A[\sigma^{-1}(1)] \leq A[\sigma^{-1}(2)] \leq \cdots \leq A[\sigma^{-1}(n)]$. If all the elements $\pi(1), \ldots, \pi(n)$, $\sigma\pi(1), \ldots, \sigma\pi(n)$ can be computed in time $O(n)$, then A can be permuted according to π using $O(n)$ time and $O(\log n)$ additional bits of storage, in the worst case.*

The proof follows from the observation that, for each element i, $\pi(i)$ and $\sigma\pi(i)$ are computed only a constant number of times during the course of the algorithm in Figure 14. The running time of this algorithm is clearly dominated by the time spent performing these function evaluations.

The application that sparked our interest in the problem of rearranging data was building a data structure implicitly representing a complete binary search tree, given a sorted array of appropriate length n. Here the median is the root and is placed in location 1. Its children, the first and third quartiles, are in locations 2 and 3, respectively. In general, the left and right children of the element in location j are in locations $2j$ and $2j + 1$ (like a heap). Bentley [1] anticipates the result shown here in crediting Mahaney and the second author with a linear algorithm for this special case. In this case, the permutation π satisfies the property that if $x10^j$ is the $\lceil \log_2 n \rceil$-bit binary representation of i, then $0^j 1x$ is the $\lceil \log_2 n \rceil$-bit binary representation of $\pi(i)$.

When $n = 2^h - 1 + r$, where $1 \leq r < 2^h$, the sorted array can be rearranged to implicitly represent a binary search tree which is complete except for its last level and such that the nodes in the last level are as far left as possible. One method is to apply two permutations to the array. The first permutation π' moves the r values $1, 3, 5, \ldots, 2r - 1$ that belong in the last level of the tree to their correct locations at the end of the array, while keeping the other values in sorted order. Specifically,

$$\pi'(i) = \begin{cases} 2h + (i-1)/2 & \text{if } i \text{ is odd and } i < 2r, \\ i/2 & \text{if } i \text{ is even and } i < 2r, \\ i - r & \text{if } i > 2r. \end{cases}$$

The second permutation then rearranges the first $2^h - 1$ elements of the array into the implicit representation of a complete binary search tree, as above. Another method is to combine these two permutations into one, defining π as follows.

if i is odd **and** $i < 2r$
  then $\pi(i) = 2^h + (i-1)/2$
  else $\pi(i)$ is the number represented by $0^j 1x$,
    where $x10^j$ is the h-bit binary representation of
$$\begin{cases} i/2, & \text{if } i \text{ is even and } i < 2r, \text{ or} \\ i - r, & \text{if } i \geq 2r. \end{cases}$$

For most instruction sets, computing π once could take as much as $O(\log n)$ steps in the worst case. However, using only left and right shifts of distance one and single bit assignments and tests, $O(n)$ operations suffice to evaluate $\pi(i)$ for all $i \in \{1, \ldots, n\}$. As mentioned above, this is sufficient to obtain a linear time algorithm.

4. Conclusions

Since most permutations fix very few of their elements, any algorithm must take $\Omega(n)$ time on average and, hence, in the worst case. Also, $\Omega(\log n)$ additional bits of storage are needed to provide pointers to array elements that are being interchanged.

The open question that remains is whether there are better algorithms for rearranging an array given an arbitrary permutation π (and also possibly π^{-1}). Specifically, are there algorithms that use a small amount of additional space (e.g. $O(\log n)$ or $\log^{O(1)} n$) and only linear time? If not, are there deterministic algorithms that, given only π, run in $O(n \log n)$ time but use only $O(\log n)$ bits of additional storage?

When $b = n/\log\log n$, the algorithm in Figure 4 uses $O(n \log\log n)$ time and $O(n/\log\log n)$ bits of additional storage. However, it is not even known whether there is an algorithm that uses $o(n \log n)$ time and $O(n^{1-\epsilon})$ space for some constant $\epsilon > 0$.

All of the algorithms presented in this paper are cycle leader algorithms. They proceed by identifying exactly one element in each cycle and rotating the cycle starting from that element. Performing all the rotations requires only linear time and two pointers. This leaves the problem of finding a leader for each cycle, or equivalently, finding the minimum element in each cycle. Cook and McKenzie [3] have shown that these problems and, more generally, computing the disjoint cycle representation of a permutation are NC^1-complete for deterministic logspace.

A previous version of the algorithm in Figure 7 was interesting from the point of view that it did not operate by determining cycle leaders. Instead, it performed sweeps up and down the array, at each point deciding whether to interchange that element with another element and, if so, which one. It would be interesting to know if any algorithm for permuting an array can be transformed into a cycle leader algorithm. This would allow us to restrict attention to cycle leader algorithms when searching for new algorithms or trying to prove better tradeoffs.

The algorithms in Figures 5, 7, and 8 and the $O(n^2)$ time, $O(\log n)$ space algorithm discussed before Figure 4 can all be viewed as instances of a restricted type of cycle leader algorithm. Specifically, they separately test each element $i \in \{1, \ldots, n\}$ to see if it is a cycle leader by comparing elements forwards and/or backwards along the cycle. The test only depends on the permutation π and i itself. It does not depend on the order in which the elements are tested, as is the case for the algorithm in Figure 4 (when $b > 1$) and the algorithm in Figure 14, nor on the data values stored in the array, as is the case for the algorithm in Figure 14.

Such algorithms are interesting because they immediately lead to distributed algorithms, using only comparisons of ID's, for electing a leader in a bidirectional ring of synchronous processors. Processor i tests whether element i is the leader of its cycle once it has learned the ID's of a sufficient number of its successors and predecessors. It requests these ID's by sending messages forwards and backwards around the ring for distances which are successive powers of 2. (See [8].) The total number of messages sent is proportional to the time taken by the cycle leader algorithm. Frederickson and Lynch [5] have shown that any such distributed leader election algorithm sends $\Omega(n \log n)$ messages in the worst case. Thus these restricted type of cycle leader algorithms must use at least $\Omega(n \log n)$ time in the worst case. On the other hand, it is not clear how algorithms for electing a leader in a ring of processors can be used to obtain cycle leader algorithms. The major problems seem to be how to deal with the timing of messages and how to represent the states of all the processors using less than a linear amount of space.

References

[1] J.L. Bentley, *Programming Pearls*, Addison-Wesley, Reading, Mass., 1986.

[2] G. Brassard and P. Bratley, *Algorithmics*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[3] S. Cook and Pierre McKenzie, "Problems Complete for Deterministic Logarithmic Space", *J. Alg.*, vol. 8, 1987, pages 385–394.

[4] D. Dolev, M. Klawe, and M. Rodeh, "An $O(n \log n)$ Unidirectional Distributed Algorithm for Extrema Finding in a Circle", *J. Alg.*, vol. 3, 1982, pages 245–260.

[5] G.N. Frederickson and N.A. Lynch, "The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring", *Proc. 16th ACM Symposium on Theory of Computing*, 1984, pages 493–505.

[6] .G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum-weight Spanning Trees", *ACM Trans. Prog. Lang. Sys.*, vol. 5, no. 1, January 1983, pages 66–77.

[7] G.C. Goldbogen, "PRIM: A Fast Matrix Transpose Method", *IEEE Trans. Soft. Eng.*, vol. SE-7, no. 2, March 1981.

[8] D.S. Hirschberg and J.B. Sinclair, "Decentralized Extrema-Finding in Circular Configurations of Processes", *CACM*, vol. 23, Nov. 1980, pages 627–628.

[9] R. Impagliazzo, private communication.

[10] .W. Kirchherr, "Transposition of an $l \times l$ Matrix Requires $\Omega(\log l)$ Reversals on Conservative Turing Machines", *IPL*, vol. 28, 1988, pages 55–59.

[11] D.E. Knuth, "Mathematical Analysis of Algorithms", *Proc. IFIP Congress*, 1971, vol. 1, pages 19–27.

[12] D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, MA., 1973.

[13] D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA., 1973.

[14] G.L. Peterson, "An $O(n \log n)$ Unidirectional Algorithm for the Circular Extrema Problem", *ACM Trans. on Prog. Lang. Sys.*, vol. 4, no. 4, October 1982, pages 758–762.