Raising Permutations to Powers In Place*

Hicham El-Zein[1], J. Ian Munro[1], and Matthew Robertson[1]

1 **Cheriton School of Computer Science,**
University of Waterloo,
Ontario, Canada N2L 3G1.
helzein|imunro|m32rober@uwaterloo.ca

Abstract

Given a permutation of n elements, stored as an array, we address the problem of replacing the permutation by its k^{th} power. We aim to perform this operation quickly using $o(n)$ bits of extra storage. To this end, we first present an algorithm for inverting permutations that uses $O(\lg^2 n)$ additional bits and runs in $O(n \lg n)$ worst case time. This result is then generalized to the situation in which the permutation is to be replaced by its k^{th} power. An algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\epsilon}\})$ additional bits is presented.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Algorithms, Combinatorics, Inplace, Permutations, Powers

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2016.34

1 Introduction

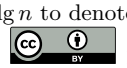
Permutations are fundamental in computer science and are the subject of extensive study. They are commonly used as a basic building block for space efficient encoding of strings [1, 8, 12, 14], binary relations [3, 2], integer functions [11] and many other combinatorial objects.

In this paper, we study the problem of transforming a permutation π to its k^{th} power π^k in place. By "in place," we mean that the algorithm executes while using "very little" extra space. Ideally, we want the algorithm to use only a polylogarithmic number of additional bits. The algorithm we present uses several new techniques that are of interest in their own right and could find broader applications.

One interesting application of inverting a permutation in place was encountered in the content of data ware-housing by a Waterloo company [4]. Under specific indexing schemes, the permutation corresponding to the rows of a relation sorted by any given key is explicitly stored. To perform certain joins, the inverse of a segment of the permutation is precisely what is needed. This permutation occupies a substantial portion of the space used by the indexing structure. Doubling this space requirement, to explicitly store the inverse of the permutation, for the sole purpose of improving the time to compute certain joins may not be practical, and indeed was not in the work leading to [4].

Since there are $n!$ permutations of length n, the number of bits required to represent a permutation is $\lceil \lg(n!) \rceil \sim n \lg n - n \lg e + O(\lg n)$ bits.[1] Munro et al. [11] studied the space efficient representation of general permutations where general powers of individual elements can be computed quickly. They gave a representation taking the optimal $\lceil \lg(n!) \rceil + o(n)$ bits, that can compute the image of a single element of $\pi^k()$ in $O(\lg n / \lg \lg n)$ time; and a

* This work was sponsored by the NSERC of Canada and the Canada Research Chairs Program.
[1] We use $\lg n$ to denote $\log_2 n$



representation taking $(1+\epsilon)n \lg n$ bits where $\pi^k()$ can be computed in constant time. The preprocessing for these representations as presented in [11] requires an extra $O(n)$ words of space, so a solution that involves building them as an intermediate step will not be considered inplace and therefore does not apply to our current problem. For further details on permutation representations see [6, 10, 5].

Throughout this paper, we assume that the permutation is stored in an array $A[1, \ldots, n]$ of n words. The array originally contains the values $\pi(1), \ldots, \pi(n)$, then, afterwards, it contains the values $\pi^k(1), \ldots, \pi^k(n)$. Storing A requires $n\lceil \lg n \rceil = n \lg n + n(\lceil \lg n \rceil - \lg n)$ bits. When $(\lceil \lg n \rceil - \lg n)$ is "big," we can reduce the space required by this representation by encoding a constant number c of consecutive elements into a single object. This object is essentially the c digits, base n number $\pi[i]\pi[i+1]\ldots\pi[i+c-1]$. Encoding these n/c objects of size $\lceil c \lg n \rceil$ bits each, totals to $n \lg n + n/c$ bits. To decode a value, we need a constant number of arithmetic operations. This saving of memory at the cost of c accesses to interpret one element of A carries through all of our work.

This paper is organized as follows. In Section 2, we review previous work on permuting data in place [7], on which we base our work. In Section 3, we start by presenting an algorithm for inverting permutations that uses $O(b + \lg n)$ additional bits and runs in $O(n^2/b)$ worst case time. Using a different approach, we improve the worst case time complexity to $O(n \lg n)$, but using $O(\sqrt{n} \lg n)$ additional bits. This development then leads to our main algorithm for inverting permutations, we achieve an algorithm with a worst case time complexity of $O(n \lg n)$ using only $O(\lg^2 n)$ additional bits. Then we face the problem that while $\pi^{-1}()$ leaves the cycle structure as it was, higher powers may create more (smaller) cycles. This causes further difficulty which is addressed in Section 4 where we generalize the algorithm from Section 3 to the situation in which the permutation is to be replaced by its k^{th} power. An algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\epsilon}\})$ additional bits is presented. Our solution relies on Rubinstein's [13] work on finding factorizations into small terms modulo a parameter. The final result can be improved if better factorization is applied. However, we show that obtaining a better factorization is probably difficult since it would imply Vinogradov's conjecture [15]. We conclude our work in Section 5.

2 Background and Related Work

Fich et al. studied the problem of permuting external data according to a given permutation, in place [7]. That is, given an array B of length n and a permutation π given by an oracle or read only memory, rearrange the elements of B in place according to π.

It is not sufficient to simply assign $B[\pi(i)] \leftarrow B[i]$ for all $i \in \{1, \cdots, n\}$, because an element in B may have been modified before it has been accessed. A permutation can be thought of as a collection of disjoint cycles. The procedure Rotate, rotates the values in B according to π by calling RotateCycle on the leader of each cycle. A cycle leader is a uniquely identifiable position in each cycle. The smallest position in a cycle, or *min leader*, is a simple example of a cycle leader.

```
procedure Rotate(B)
    for i ← 0 to n − 1 do
        if IsLeader(i) then
            RotateCycle(B, i)

procedure RotateCycle(B, leader)
    i ← π(leader)
    while i ≠ leader do
        Swap(B[i], B[leader])
        i ← π(i)
```

Figure 1 Rotates the values in B according to a permutation π.

The problem is to identify a position as leader by starting at that position and traversing only forward along the cycle. Choosing the min leader would take $\Theta(n^2)$ value inspections in the worst case. A leader that we call the *local min leader* can be used to permute data in $O(n \lg n)$ worst case time complexity using only $O(\lg^2 n)$ additional bits [7]. As stated in [7], the local min leaders of a permutation π are characterized as follows. Let $E_1 = \{1, \ldots, n\}$ and $\pi_1 = \pi$. For positive integers $r > 1$, define E_r as the set of local minima in E_{r-1} encountered following the cycle representation of the permutation π_{r-1} and define π_r as the permutation that maps each element of E_r to the next element of E_r that is encountered following π_{r-1}. More formally, $E_r = \{i \in E_{r-1} | \pi_{r-1}^{-1}(i) > i < \pi_{r-1}(i)\}$ and $\pi_r : E_r \to E_r$ is defined such that $\pi_r(i) = \pi_{r-1}^m(i)$ where $m = \min\{m > 0 | \pi_{r-1}^m(i) \in E_r\}$. Since at most half the elements in each cycle are local minima, $|E_r| < |E_{r-1}|/2$ and $r \leq \lg n$. The leader of a cycle is the unique position i, such that $\pi_{r-1} \ldots \pi_1(i) \in E_r$. For example, if $\pi = (1\ 7\ 2\ 9\ 4\ 5\ 3\ 10\ 6\ 8)$ as illustrated in Figure 2 (similar to Figure 6 in [7]), then

$E_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, $\pi_1 = (1\ 7\ 2\ 9\ 4\ 5\ 3\ 10\ 6\ 8)$
$E_2 = \{1, 2, 4, 3, 6\}$, $\pi_2 = (1\ 2\ 4\ 3\ 6)$
$E_3 = \{1, 3\}$, $\pi_3 = (1\ 3)$
$E_4 = \{1\}$, $\pi_4 = (1)$

The local min leader of the only cycle in π is the position 9 since $\pi_3\pi_2\pi_1(9) = 1$.

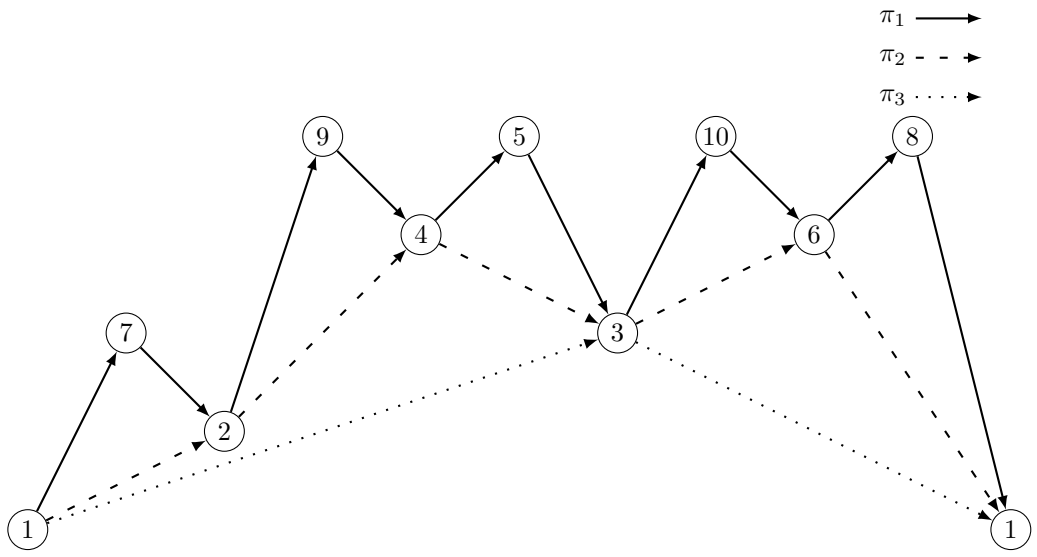


Figure 2 An illustration of π_i.

The procedure IsLocalMinLeader (see Figure 3), checks if position i in the permutation is the local min leader of his cycle. It has the property of proceeding at most $4n$ steps on the permutation for a single element, and a total of $O(n \lg n)$ steps on the permutation for all elements. We treat the local min leader technique as a black box. There are a few occasions where we need details so we provide the procedure to make this paper more self contained. We refer the reader to [7] for further details on this procedure.

```
procedure IsLocalMinLeader(i)
    elbow[0] ← elbow[1] ← i
    for r ← 1, 2, ... do
        //loop invariant:
        {elbow[r] = π_{r−1} ... π_1(i)}
        Next (r)
        if elbow[r] > elbow[r − 1] then
            elbow[r] ← elbow[r − 1]
            Next (r)
            if elbow[r] > elbow[r − 1] then
                return false
            elbow[r + 1] ← elbow[r]
        else if elbow[r] = elbow[r − 1] then
            return true
```

```
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)
```

Figure 3 Checks if index i is a local min leader.

3 Inverting Permutations

To invert a permutation we can use the structure of the algorithm described in Figure 1, but invert the cycles instead of rotating the data. Figure 4 shows how to invert a cycle. The algorithm iterates over the permutation, and inverts each cycle only on its leader. A cycle leader must be used that will remain unchanged once the cycle is inverted. An example of such a cycle leader is the min leader.

```
procedure InvertCycle(A, leader)
    current ← A[leader]
    previous ← leader
    while current ≠ leader do
        next ← A[current]
        A[current] ← previous
        previous ← current
        current ← next
    A[leader] ← previous
```

Figure 4 Inverts a permutation.

Inverting a permutation using min leader will use $O(\lg n)$ additional bits and take $\Theta(n)$ time if the permutation consists of one large cycle in increasing order; or $\Theta(n^2)$ time if the permutation consists of one large cycle in decreasing order. We note that for a random cycle of length n this total cost would be about $n \lg n$. The analysis is similar to the bidirectional distributed algorithm for finding the smallest of a set of n uniquely numbered processors arranged in a circle [9]. However, our interest is in finding algorithms with good worst case performance.

A permutation can be inverted in linear time using a n-bit vector. The vector can be used to mark corresponding positions in π as their cycles are inverted. This is equivalent to using the min leader, but takes $n + O(\lg n)$ additional bits.

Using a technique presented in [7], the bit vector can be shrunk to b-bits by conceptually dividing the permutation into $\lceil n/b \rceil$ sections each of size b (except possibly the last section will be smaller). The b-bit vector is reset at the start of each section and is used to keep

track of which positions are encountered in the section being processed. If the position under consideration for being a cycle leader has a corresponding bit with value 0, its cycle is traversed searching for a smaller position. If no smaller position is found, then the position is a cycle leader and the cycle is inverted. On the other hand, if the position under consideration has a corresponding bit with value 1, then the position was previously encountered as part of a cycle containing a smaller position in the section, and hence is not a cycle leader. Each cycle will be traversed at most n/b times, thus the total runtime is n^2/b and the space used is $b + O(\lg n)$.

▶ **Theorem 1.** *In the worst case, the array representation of a permutation of length n can be replaced with its own inverse in $O(n^2/b)$ time using $b + O(\lg n)$ additional bits of space.*

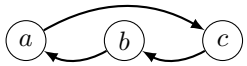
By setting $b = \sqrt{n}$ we get the following corollary.

▶ **Corollary 2.** *In the worst case, the array representation of a permutation of length n can be replaced with its own inverse in $O(n\sqrt{n})$ time using $O(\sqrt{n})$ additional bits of space.*

3.1 Inversion in $O(n \lg n)$ Time Using $O(\sqrt{n} \lg n)$ Bits

The local min leader of a cycle will, in general, change after the cycle has been inverted. Figure 5 shows a simple example of this: b is the leader of the cycle, but if it were inverted, c would become the leader. Since $c > b$, the algorithm in Figure 4 will invert the cycle once on b and then again on c because c will look like a leader when it is reached in the outer loop. Inverting the cycle the second time will undo the work of inverting it the first time. We will call a cycle with this problem a *bad cycle*.

▶ **Definition 3.** A *bad cycle* is a cycle with the property that if inverted, has a new cycle leader not yet processed, i.e., larger than the original leader.



$a < b < c$

Figure 5 An example of a bad cycle.

It is not hard to build a permutation that will have $\Theta(n)$ bad cycles. Such a permutation could just repeat our bad cycle pattern and create exactly $\lfloor n/3 \rfloor$ bad cycles. So, there is not enough space to use even 1 bit to mark these cycles.

▶ **Theorem 4.** *A permutation π represented as an array can be replaced with π^{-1} in place using $O(\sqrt{n} \lg n)$ extra bits in $O(n \lg n)$ time.*

Proof. Although the permutation π can contain up to n cycles, the number of distinct cycle lengths in π, which we denote by k, is less than $\lfloor \sqrt{2n} \rfloor$ (since $\sum_{i=1}^{\lceil \sqrt{2n} \rceil} i > n$). We store these cycle lengths in an array L of size $O(\sqrt{n} \lg n)$ bits. This can be done in $O(n \lg n)$ time by iterating over the permutation and computing the length of every cycle as it is detected on its local min leader using the procedure IsLocalMinLeader (see Figure 3). After a length is detected, query a balanced binary search tree H to check if the length computed was already encountered; if it was not encountered, insert the new length to L and H. The cycle lengths are ranked according to their position in L.

If a position i is found to be the local min leader of a cycle α, then the minimum position in α is given by $x = \pi_{r-1} \dots \pi_1(i)$. Let $j = \pi_1 \dots \pi_{r-1}(x)$, then $x = \pi_{r-1}^{-1} \dots \pi_1^{-1}(j)$ and j

is the local min leader of the inverse α^{-1} of α. When testing the position i for leadership, the procedure IsLocalMinLeader will store j in $elbow[0]$ upon termination because of its loop invariant (at the beginning of each iteration: $elbow[r] = \pi_{r-1} \dots \pi_1(i)$). Thus, we can identify the leader of α^{-1} while testing the leadership of position i without the need for testing each position in α^{-1}. A bad cycle can easily be identified by checking if $j > i$.

▶ **Definition 5.** A *tail* of a cycle is the position that points to its local min leader, i.e., if t is the tail of a cycle c with local min leader l, then $\pi(t) = l$.

The algorithm iterates over the permutation similar to the algorithm in Figure 4, and invert each cycle only on its local min leader. If a bad cycle α was detected, we modify the tail of the inverted cycle α^{-1} to point to the rank of the length of the cycle instead of back to the leader of the inverted cycle. Note that the tail ($\pi(elbow[0])$) can be found by probing $A[elbow[0]]$ before inverting the cycle.

When pointing to the ranks of the cycles length, we have to use values in the range of 1 to n, otherwise the size of each entry in A may increase to $\lceil \lg n \rceil + 1$ bits and we may end up using n additional bits. The problem now is that A does not distinguish between pointing to a cycle length rank, or pointing to a different position in the cycle. This can be solved with a table T of size $O(\sqrt{n} \lg n)$ bits that stores the positions of the permutation that point to its first k positions. T will initially store $\pi^{-1}(1), \dots, \pi^{-1}(k)$. It is set by initially traversing the permutation, then it is updated as cycles are inverted.

While testing for the leadership of a position i, if a position t is found such that $\pi(t) \leq k$, then t can be checked against T in $O(1)$ time to determine if $A[t]$ points to a cycle length rank or a position in the cycle. If it is the latter case, we simply continue. Else if it points to a cycle length rank, abort the procedure IsLocalMinLeader and do not invert the cycle. If the length traversed so far matches the cycle length stored in L at rank $A[t]$, then the position i is the local min leader of an already inverted cycle. Restore the cycle by setting $A[t] = i$.

The total time spent is $O(n \lg n)$, and the space used is $O(\sqrt{n} \lg n + \lg^2 n)$. ◀

3.2 Reducing Extra Space to $O(\lg^2 n)$ Bits

Next, we extend the approach presented in the previous subsection to achieve an algorithm for inverting permutations with $O(n \lg n)$ worst case time complexity while using only $O(\lg^2 n)$ bits. First we start with some definitions.

Given a permutation π, the *depth* of a position $e \in \pi$ is the maximum index d such that $\pi_{d-1} \dots \pi_2 \pi(e) \in E_d$.[2] For example, the depth of 10 in Figure 2 is 3 since $\pi_2\pi_1(10) = 1 \in E_3$ and $\pi_3\pi_2\pi_1(10) = 3 \notin E_4$. Let c be a cycle in π of size l with local min leader s_1. We define S_1 as the following sequence: $s_1, s_2, \dots, s_l$ where $s_i = \pi(s_{i-1})$ for $i > 1$; s_l is the tail of the cycle c. For $i > 1$, S_i is a subsequence of S_{i-1} formed by the local minima in S_{i-1} excluding S_{i-1}'s first and last elements. The *limited depth* of a position $e \in \pi$ is the maximum index d such that $\pi_{d-1} \dots \pi_2 \pi(e) \in S_d$. The values $s_1, \dots, s_{i-1}$ are not needed to evaluate the limited depth of s_i, but the values $s_i, \dots, s_l$ are required. The limited depth of a position is upper bounded by its depth. Notice that the first element in S_i is always $\pi_{i-1} \dots \pi(s_1)$, since s_1 is the local min leader of c. Moreover, the limited depth d of a cycle's local min leader is either unique or shared by at most one other element $\pi_1^{-1} \dots \pi_{d-1}^{-1}(\pi_d \dots \pi_2 \pi(s_1))$ in the

[2] For the definition of π_i where $i \in \{1, \dots, d\}$ check Section 2.

cycle. The depth and limited depth of a position can be computed in a manner similar to the procedure IsLocalMinLeader with the same space and time complexity.

We say that a cycle is *broken* if its tail points to a position other than its local min leader. We call this position the broken cycle's *intersection*. We define the *spine* to be the path from the leader to the intersection, and the *loop* to be the cycle containing the intersection and the tail. Figure 6 demonstrates these terms.

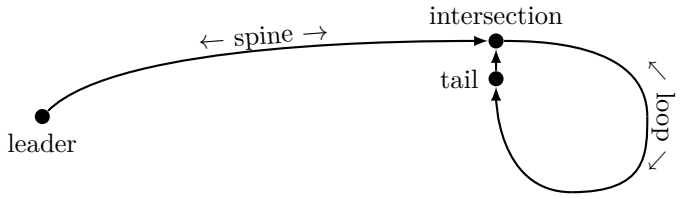


Figure 6 An example of a broken cycle.

Following the algorithm described previously, when a cycle c is detected it is replaced by its inverse; if c is detected to be a bad cycle, the tail of c^{-1} is modified to store the limited depth of c^{-1}'s local min leader k. In that case, the tail of c^{-1} will be modified to point to the unique position whose limited depth is the same as k if that position was encountered before k, thus making c^{-1} a broken cycle. Finally, c^{-1} will be restored once k is encountered. As in the previous subsection, for A to distinguish between pointing to a limited depth, or pointing to a different position in the cycle we use a table T of size $O(\lg^2 n)$ bits that stores the positions of the permutation that point to its first $\lg n$ positions.

The algorithm iterates over the permutation. At each position i, it interleaves four scans $\mathscr{F}$, $\mathscr{L}$, $\mathscr{T}$ and $\mathscr{H}$. For every operation run on $\mathscr{F}$, a constant number of operations are run on $\mathscr{L}$; and for every operation run on $\mathscr{L}$ a constant number of operations are run on $\mathscr{T}$ and $\mathscr{H}$. $\mathscr{F}$ is used to determine whether i is the local min leader of its cycle (c or c^{-1}), $\mathscr{L}$ is used to determine the limited depth of i, and $\mathscr{T}$ and $\mathscr{H}$ are used to determine if i's cycle was broken, and to restore it. The $\mathscr{T}$ and $\mathscr{H}$ scans have two phases:

- The first phase is the classic tortoise and hare algorithm for cycle detection. It is used to check if i's cycle is broken. $\mathscr{T}$ (for tortoise) and $\mathscr{H}$ (for hare) both start at position i, $\mathscr{T}$ proceeds at one step per iteration and $\mathscr{H}$ proceeds at two steps until they meet at position j. Phase one will consist of no more than l iterations, where l is the length of i's cycle. This is because at each iteration, the forward distance (i.e. the distance from $\mathscr{H}$ to $\mathscr{T}$ traversing forward in the cycle) between the two pointers will decrease by one; or if the cycle was broken, the distance decreases once both pointers enter the broken cycle's loop. If one of the scans encounters a limited depth or if i is reachable from j, $\mathscr{T}$ and $\mathscr{H}$ are aborted while $\mathscr{F}$ and $\mathscr{L}$ continue. Otherwise, we know that the cycle is broken and we proceed to the second phase.
- The aim of the second phase is to find the tail of the broken cycle c^{-1}. Let λ be the length of c^{-1}'s loop, μ be the distance from i to c^{-1}'s intersection, and δ be the distance from the intersection to j. Denote by d_t and d_h the distance traveled by the pointers in $\mathscr{T}$ and $\mathscr{H}$ respectively. $d_t = \mu + \delta$ and $d_h = \mu + k\lambda + \delta$ where $k \in \mathbb{Z}^+$. We know

$$\begin{aligned} 2d_t &= d_h \\ 2(\mu + \delta) &= \mu + k\lambda + \delta \\ \mu &= k\lambda - \delta \ . \end{aligned}$$

Thus, if we reset $\mathscr{T}$'s pointer to position i, while $\mathscr{H}$ remains at j, and as in the first

phase, $\mathscr{T}$ proceeds at one step per iteration and $\mathscr{H}$ proceeds at two steps: $\mathscr{T}$ and $\mathscr{H}$ will meet at c^{-1}'s intersection. Then, c^{-1}'s tail can be found by iterating through c^{-1}'s loop till a position that points to the intersection is reached. After finding the tail, the limited depth of the intersection (which will always be the same as the limited depth of c^{-1}'s leader) is computed.

The $\mathscr{L}$ scan aims to compute the limited depth of position i. To do so, $\mathscr{L}$ should identify the tail of c or c^{-1}. $\mathscr{L}$ identifies the tail correctly if it encounters a position storing a limited depth (then that position is the tail), or if the cycle is broken and the tail is computed by the $\mathscr{T}$ and $\mathscr{H}$ scans (as is the case when the cycle is broken and i is on its spine). In the other cases, the $\mathscr{L}$ scan assumes that the tail is the position pointing to i. It returns a correct value if i is a local min leader, and it may not return a correct value otherwise. However, returning an incorrect value in the other cases does not affect the correctness of the algorithm.

The $\mathscr{F}$ scan tests whether i is the local min leader of c or c^{-1}. If $\mathscr{F}$ encounters a limited depth or if the scans $\mathscr{T}$ and $\mathscr{H}$ detect that c^{-1} is broken, $\mathscr{F}$ will behave as if the tail of c^{-1} points to i. The $\mathscr{F}$ scan terminates on one of the following cases:

- The first case is $\mathscr{F}$ determines that i is not a local min leader. If so, the entire process of all four scans is aborted.
- The second case is $\mathscr{F}$ determines the position is a local min leader. Then, two cases can occur:
  - If c^{-1} was broken or a limited depth was encountered, then we know that the cycle is already inverted. Compare the limited depth of i that is computed by $\mathscr{L}$ to the limited depth stored or computed by $\mathscr{T}$ and $\mathscr{H}$. If the two values are equal make the tail point to i. Alternatively, abort all four scans.
  - Otherwise, the cycle c is not inverted. Invert c and if it was bad store in its tail the limited depth of c^{-1}'s local min leader.

Analysis: All four scans use $O(\lg^2 n)$ extra bits. The time complexity is bounded by the time complexity of $\mathscr{F}$, since the runtime of $\mathscr{L}$, $\mathscr{T}$ and $\mathscr{H}$ is at most a constant factor times the runtime of $\mathscr{F}$. For each cycle c, the time spent by F testing for leadership before inverting the cycle is $O(l \lg l)$ where l is the length of c. Inverting c and properly setting its tail if it was bad will take $O(l)$ time. After inverting c, if c^{-1} is bad at most one intermediate broken cycle can be formed, since the limited depth of the local min leader is unique or shared by at most one other position. This fact is crucial to our analysis, and it is the reason why the $\mathscr{L}$ scan is introduced. The time spent testing for leadership for indices in c^{-1} is divided into the following cases:

- c^{-1} is broken and the position i being tested is in c^{-1}'s loop.
- Otherwise either c^{-1} is broken and i is in the spine, or c^{-1} is not broken and the tail stores the limited depth of the leader.
  - If $\mathscr{T}$ does not inspect the tail, then the runtime will be the same as testing whether i is the local min leader of c^{-1}.
  - Otherwise, the procedure will test if i is the local min leader of the cycle formed by pointing the tail of c^{-1} to i. It will iterate at most 4 times from i to the tail [7]. So, the time complexity will be at most 4 times the time complexity of testing weather i is the local min leader of c^{-1}.

In all cases the runtime is bounded by $O(l \lg l)$. Thus, the total runtime per cycle is $O(l \lg l)$ and the total runtime for the whole algorithm is $O(n \lg n)$.

▶ **Theorem 6.** *In the worst case, the standard representation of a permutation of length n can be replaced with its own inverse in $O(n \lg n)$ time using $O(\lg^2 n)$ extra bits of space.*

4 Arbitrary Powers

The k^{th} power of a permutation π is π^k defined as follows:

$$\pi^k(i) = \begin{cases} \pi^{k+1}(\pi^{-1}(i)) & k < 0 \\ i & k = 0 \\ \pi^{k-1}(\pi(i)) & k > 0 \end{cases}$$

where k is an arbitrary integer. In this section we extend the techniques presented in the previous section to cover the situation in which the permutation is to be replaced by its k^{th} power for an arbitrary integer k. We present an algorithm whose worst case running time is $O(n \lg n)$ and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\epsilon}\})$ additional bits.

Without loss of generality, we assume that k is positive. If k is negative, we invert the permutation then raise it to the power of $-k$. Raising a cycle to an arbitrary power can result in several disjoint cycles as illustrated in Figure 7.

▶ **Lemma 7.** *Raising a cycle of length l to its k^{th} power, will produce $\gcd(k, l)$ cycles each of length $l/\gcd(k, l)$.*

Proof. Suppose μ cycles are produced. Since they are all symmetric, they will have the same length λ. λ is the smallest positive integer such that $(\pi^k)^\lambda(i) = \pi^{k\lambda}(i) = i$, so $k\lambda = cl$ for an integer c that is relatively prime with λ. Now

$$l = \lambda\mu$$
$$k = cl/\lambda = c\mu,$$

but c is relatively prime with λ, so $\mu = \gcd(k, l)$ and $\lambda = l/\gcd(k, l)$. ◀

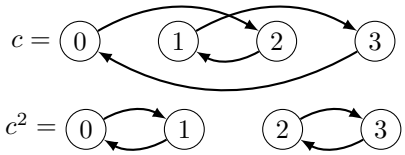


Figure 7 Raising c to the second power results in two separate cycles.

Given a cycle, it is not hard to raise the cycle to its k^{th} power while using $O(k)$ words or $O(k \lg n)$ bits. Starting from position i, store $i, \pi(i), \pi^2(i), \ldots, \pi^{k-1}(i)$ in an array B using $O(k \lg n)$ bits. Replace $A[i]$ with $A[\pi^{k-1}(i)]$, then replace $A[\pi(i)]$ with $A[\pi^k(i)]$, and so on until $A[\pi(i)^{l-k}]$ is reached where l is the length of the cycle. Replace $A[\pi(i)^{l-k}]$ till $A[\pi(i)^{l-1}]$ with the values stored in B. When the procedure terminates, $A[i], A[i+1], \ldots, A[i+\gcd(k, l)-1]$ will contain a position from each resulting cycle. An algorithm to raise a permutation to its k^{th} power, will be the same as the algorithm presented in subsection 3.2, however, the $\mathscr{T}$ scan will raise cycles to their k^{th} power instead of inverting them once they are detected. Then, it will iterate through every cycle of the resulting $\gcd(k, l)$ cycles and compute its leader to check if it was bad. If so, it computes the limited depth of the leader and store it in the cycle's tail.

▶ **Theorem 8.** *In the worst case, the standard representation of a permutation of length n can be replaced with its k^{th} power, when k is bounded by some polynomial function of n, in $O(n \lg n)$ time using $O(\lg^2 n + k \lg n)$ extra bits of space.*

Theorem 8 is useful if the value of k is small. In the next subsection, we show how to power permutations using $o(n)$ extra bits of space.

4.1 Powering Permutations in $O(n \lg n)$ Time using $o(n)$ Extra Bits

To improve the space complexity, we only have to modify the way we are raising cycles to their k^{th} power. To raise a cycle to its k^{th} power we first find its length l, then we factorize $k \bmod l$. Since $k \bmod l < l$, it can be factored trivially in $o(l)$ time while using little extra storage.

Next, raise the cycle to the power of every prime factor p separately. Here we have to distinguish between two cases:

- **First Case: p and l are relatively prime**
  We will use the following theorem given by Rubinstein [13]:
  ▶ **Theorem 9** (Rubinstein [13], Theorem 4.3). *Let* $\gcd(N, a) = 1$ *and* R *be a rectangle. Then,* $c_R(N, a)$*, the number of solutions* (x, y) *to* $xy = N \bmod a$ *with* (x, y) *lying in the rectangle* R *is equal to*

  $$\frac{\text{area}(R)}{a^2}\phi(a) + O(a^{1/2+\epsilon})$$

  for any $\epsilon > 0$*, where* ϕ *is Euler's totient function.*
  In particular, there exist a point (x, y) *where* $xy = N \bmod a$ *in any square* R *with side length at least* $a^{3/4+\epsilon}$ *(*R *must be larger than* $a^{3/2+\epsilon}$*).*
  In this case $\gcd(p, l) = 1$, so there always exist two integers $x, y < l^{3/4+\epsilon}$ such that $xy = p \bmod l$. To find x and y, do a linear search that takes $O(l^{3/4+\epsilon})$ time. Then, raise the cycle to the x^{th} power followed by the y^{th} power using the method described in the previous subsection. The total runtime is $O(l)$ and the space used is $O(l^{3/4+\epsilon})$.
- **Second Case: p divides l**
  In this case $\gcd(p, l) = p$ (since p divides l). We will reduce this case to the previous one. Modify the permutation π to form the permutation π' that results from adding an additional position e to the cycle c in π to form the cycle c'. More formally, π' is defined as follows:
  - Let a be a position in the cycle c; for all positions $i \in \pi$ except $\pi^{-1}(a)$, $\pi'(i) = \pi(i)$.
  - $\pi'(\pi^{-1}(a)) = e$ (where e is a new position).
  - $\pi'(e) = a$.

  This modification can be done by storing a and two extra words, where the first word stores the inverse of a, and the second stores the image of e ($\pi'(e)$). Each time the array A is accessed at an index i, if $A[i]$ is equal to a, i is checked against the first word stored. If they match, then $A[i]$ points to a otherwise $A[i]$ points to e. Doing this eliminates the need for increasing the word size.
  Let $\{c_{ij} | 0 \le i < l/p, 0 \le j < p\}$ be the positions of c, such that
  - $\pi(c_{ij}) = c_{i(j+1)}$ if $j < p - 1$
  - $\pi(c_{ij}) = c_{(i+1 \bmod l/p)0}$ if $j = p - 1$

  Raising c to the power of p will result in p cycles such that the j^{th} cycle c_j will contain the positions $\{c_{ij} | 0 \le i < l/k\}$, where $\pi^p(c_{ij}) = c_{(i+1 \bmod l/p)j}$. The length of c' is $l + 1$ and $\gcd(l + 1, p) = 1$ (since p divides l), so raising c' to the p^{th} power will result in only one cycle.
  Without loss of generality assume that $a = c_{00}$. The positions c_{ij} satisfying $a = \pi^m(c_{ij})$ for some $m \in [1, p - 1]$ are precisely $c_{((l/p)-1)j}$ where $j \in [0, p - 1]$. Notice that
  - $\pi'^p(c_{ij}) = \pi^p(c_{ij})$ for all c_{ij} such that $a \ne \pi^m(c_{ij})$ for all $m \in [1, p - 1]$
  - $\pi'^p(c_{((l/p)-1)0}) = e$
  - $\pi'^p(c_{((l/p)-1)j}) = c_{0(j-1)}$ for $1 \le j < p$
  - $\pi'^p(e) = c_{0(p-1)}$

Thus, if we traverse forward in c'^p starting from e, the first p positions are the positions in c_{p-1} ordered correctly, and the second p positions are the positions in c_{p-2}, and so on... After modifying π to π' raise c' to its p^{th} power. Iterate p positions starting from e, then set $A[c_{((l/p)-1)(p-1)}]$ to $c_{0(p-1)}$. Recursively raise c_{p-1} to the power of the rest of the prime factors. Repeat the same process for the rest of the cycles $c_{p-2}, \ldots, c_0$. Each time one of the resultant $\gcd(l, k)$ cycles is reached, find its local min leader and store the limited depth of the leader in the tail if it is a bad cycle. This process is illustrated in Figure 8.

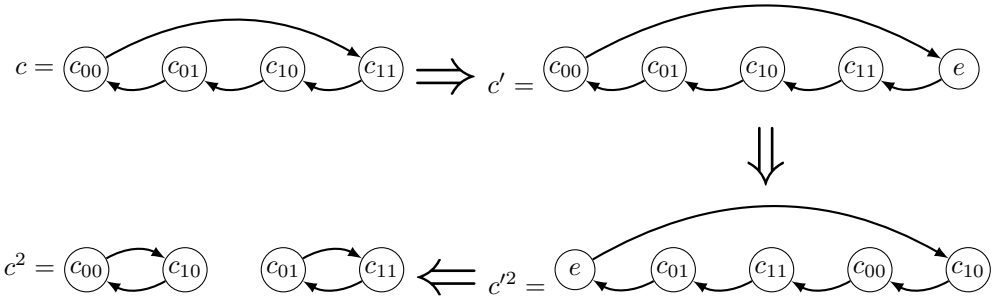


Figure 8 An illustration of case two.

▶ **Theorem 10.** *In the worst case, the standard representation of a permutation of length n can be replaced with its k^{th} power, when k is bounded by some polynomial function of n, in $O(n \lg n)$ time using $O(\lg^2 n + \min\{k \lg n, n^{3/4+\epsilon}\})$ extra bits of space.*

The space complexity in Theorem 10 can be improved if better factoring is applied. More precisely, if for any N and a where $\gcd(N, a) = 1$, we can find $g(a)$ factors $x_1, \ldots, x_{g(a)} \leq f(a)$ such that $x_1 x_2 \ldots x_{g(a)} = N \bmod a$ in $h(a)$ time, then we can achieve an algorithm with running time $O((n + h(n)) \lg n + g(n)n)$ that uses $O(\lg^2 n + \min\{k \lg n, f(n) \lg n\})$ extra bits of space.

Note that given any factoring algorithm as described above, any quadratic non-residue (mod p) can be factored to factors smaller than $f(p)$. Since at least one of the factors must also be a quadratic non-residue, this implies that the least quadratic non-residue (mod p) is smaller than $f(p)$. Thus, reducing $f(n)$ to $O(n^\epsilon)$ is probably difficult since this improvement would imply Vinogradov's conjecture [15] (that the least quadratic non-residue (mod p) lies below p^ϵ).

5 Conclusion

In this paper we presented an algorithm for inverting a permutation that runs in $O(n \lg n)$ worst case time and uses $O(\lg^2 n)$ additional bits. This algorithm is then extended to an algorithm for raising a permutation to its k^{th} power that runs in $O(n \lg n)$ time and uses $O(\lg^2 n + \min\{k \lg n, n^{3/4+\epsilon}\})$ extra bits of space. Both algorithms presented rely on the cycle's local min leader presented in [7]. Moreover, they can easily be adapted to utilize any different cycle leader. A better leader will yield a better algorithm without adding to the worst case time or space complexity for both problems as well as the problem of permuting in place [7].

References

1 Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. Reducing the space requirement of LZ-index. In Moshe Lewenstein and Gabriel Valiente, editors, *Proceedings of CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 318–329. Springer, 2006.

2 Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comput. Sci.*, 387(3):284–297, 2007.

3 Jérémy Barbay, Meng He, J. Ian Munro, and Srinivasa Rao Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4):52, 2011.

4 Mariano Paulo Consens and Timothy Snider. Maintaining very large indexes supporting efficient relational querying, August 14 2001. US Patent 6,275,822.

5 Hicham El-Zein, J. Ian Munro, and Venkatesh Raman. Tradeoff between label space and auxiliary space for representation of equivalence classes. In *Proceedings of ISAAC, PartII*, volume 8889 of *Lecture Notes in Computer Science*, pages 543–552. Springer, 2014.

6 Hicham El-Zein, J. Ian Munro, and Siwei Yang. On the succinct representation of unlabeled permutations. In *Proceedings of ISAAC*, volume 9472 of *Lecture Notes in Computer Science*, pages 49–59. Springer, 2015.

7 Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

8 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of Seventeenth SODA*, pages 368–373. ACM Press, 2006.

9 Daniel S. Hirschberg and James Bartlett Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

10 Moshe Lewenstein, J. Ian Munro, and Venkatesh Raman. Succinct data structures for representing equivalence classes. In *Proceedings of ISAAC, PartII*, volume 8283 of *Lecture Notes in Computer Science*, pages 502–512. Springer, 2013.

11 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

12 Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

13 Michael Rubinstein. The distribution of solutions to $xy = n \bmod a$ with an application to factoring integers. 2009.

14 Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

15 I. Vinogradov. *Selected works. With a biography by K. K. Mardzhanishvili.Translated from the Russian by Naidu Psv. Translation edited by Yu. A.* Springer-Verlag, Berlin, 1985.