

Fast Sorting and Pattern-Avoiding Permutations

David Arthur *
Stanford University
darthur@cs.stanford.edu

Abstract

We say a permutation π “avoids” a pattern σ if no length $|\sigma|$ subsequence of π is ordered in precisely the same way as σ . For example, π avoids $(1, 2, 3)$ if it contains no increasing subsequence of length three. It was recently shown by Marcus and Tardos that the number of permutations of length n avoiding any fixed pattern is at most exponential in n . This suggests the possibility that if π is known *a priori* to avoid a fixed pattern, it may be possible to sort π in as little as linear time. Fully resolving this possibility seems very challenging, but in this paper, we demonstrate a large class of patterns σ for which σ -avoiding permutations can be sorted in $O(n \log \log \log n)$ time.

1 Introduction

A permutation $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ is said to “contain” a pattern $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_k)$ if π contains a possibly non-contiguous subsequence $(\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_k})$ ordered in precisely the same way as σ . For example, $(3, 2, 1, 5, 6, 7, 4)$ contains the pattern $(1, 3, 2)$ since the subsequence $(1, 5, 4)$ is ordered in the same way as $(1, 3, 2)$. This is illustrated below in Figure 1. If π does not contain σ , it is said to “avoid” σ .

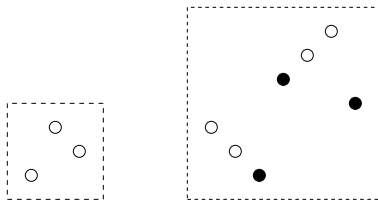


Figure 1: The permutation $(3, 2, 1, 5, 6, 7, 4)$ contains the pattern $(1, 3, 2)$.

Pattern-avoiding permutations arise naturally in a number of contexts. For example, the permutations corresponding to riffle shuffling a deck of cards are

*Supported in part by an NSF Fellowship, NSF Grant ITR-0331640, and grants from Media-X and SNRC.

precisely those that avoid the pattern $(3, 2, 1)$. A result of Knuth [1] states that a permutation can be sorted with a single stack if and only if it avoids $(2, 3, 1)$, and it can be sorted with a single input-restricted dequeue if and only if it avoids both $(4, 2, 3, 1)$ and $(3, 2, 4, 1)$.

A great deal of study has been devoted to counting pattern-avoiding permutations, which has now culminated in an international conference devoted entirely to this subject. For some typical papers, see [2, 3, 5]. Perhaps the most important result is the Stanley-Wilf conjecture, recently proven by Marcus and Tardos [4]. This states that the number of permutations π of length n that avoid a fixed pattern σ is at most C^n for some constant $C(\sigma)$.

In this paper, we propose an algorithms question that is suggested by the Stanley-Wilf conjecture. Recall that sorting an arbitrary permutation is known to take $\Omega(n \log n)$ comparisons, because $\lg(n!) = \Omega(n \log n)$ comparisons are required to distinguish between the $n!$ possible inputs. Now, suppose we want to sort a permutation π that is known to avoid a fixed pattern σ . By the Stanley-Wilf conjecture, the same lower bound argument in this case can only yield a bound of $\lg(C^n) = \Omega(n)$ here. This suggests the following question.

Question. *If a permutation π is known to avoid a fixed pattern σ , can π be sorted in less than $O(n \log n)$ time?*

Finding a complete characterization of the time required to sort pattern-avoid permutations seems very difficult. Non-trivial lower bounds are always challenging to prove, and a uniform upper bound of $O(n)$ would yield a new and very different proof of the Stanley-Wilf conjecture, which remained open for almost a decade.

In this paper, we make a first step towards solving this problem. In particular, we find a large class of patterns σ , specifically those generated by direct sums (see Section 2), for which permutations avoiding σ can be sorted in $O(n \log \log \log n)$ time.

2 Preliminaries

2.1 Permutation patterns We think of a permutation $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ as any ordered list without repeated elements. If a permutation $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_s)$ contains precisely the elements $(1, 2, \dots, s)$ in some order, then we call σ a “pattern”. We are interested in permutations that “avoid” some given pattern.

DEFINITION 2.1. Fix a permutation $\pi = (\pi_1, \dots, \pi_n)$, and a pattern $\sigma = (\sigma_1, \dots, \sigma_s)$. We say π “contains” σ if there exists $1 \leq x_1 < x_2 < \dots < x_s \leq n$ such that $\pi_{x_i} < \pi_{x_j}$ if and only if $\sigma_i < \sigma_j$. Otherwise, we say π “avoids” σ .

For example, π avoids $(2, 1)$ if and only if it is already sorted in ascending order, and it avoids $(1, 3, 2)$ if and only if there do not exist $i < j < k$ such that $\pi_i < \pi_k < \pi_j$.

Occasionally, it will be helpful to identify exactly where π contains some pattern σ . Towards that end, we will say $(\pi_{x_1}, \pi_{x_2}, \dots, \pi_{x_k})$ is a “ σ -subsequence of π ” if $\pi_{x_i} < \pi_{x_j}$ precisely when $\sigma_i < \sigma_j$. In this case, we also say π_{x_i} can “act as σ_i in a σ -subsequence of π ”.

2.2 Fast σ -sorting We are interested in sorting permutations that avoid a fixed pattern σ . However, it is convenient for the analysis to consider algorithms that gracefully handle any permutation, regardless of whether or not they avoid σ . This concept is formalized below.

DEFINITION 2.2. Fix a pattern $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_s)$. A σ -sort” must take a permutation π and:

1. Partition the elements of π into “good” and “bad” elements. An element may be labeled “bad” only if it can act as σ_s in a σ -subsequence of π .
2. Sort all of the good elements in π .

In particular, if π avoids σ , then a σ -sort will fully sort π . We will be particularly interested in σ -sorts that run in $O(n \log \log \log n)$ time, which we call “fast” σ -sorts.

2.3 Pattern Operations Finally, we discuss a few operations on patterns. We begin with a couple symmetries that are largely independent of sorting.

DEFINITION 2.3. Let $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_s)$ be an arbitrary pattern. Then, we define the reverse pattern $r(\sigma)$ to be $(\sigma_s, \sigma_{s-1}, \dots, \sigma_1)$, and the complement pattern $\bar{\sigma}$ to be $(s+1-\sigma_1, s+1-\sigma_2, \dots, s+1-\sigma_s)$.

LEMMA 2.1. If we can fast- σ -sort, then we can also fast- $r(\sigma)$ -sort and fast- $\bar{\sigma}$ -sort.

Proof. Suppose we can fast- σ -sort. Given a permutation π , we can fast- $r(\sigma)$ -sort π by first reversing π , and then fast- σ -sorting the result. We can fast- $\bar{\sigma}$ -sort π by fast- σ -sorting it with respect to the $>$ operator instead of the usual $<$ operator.

A more complicated operation for our purposes is the direct sum of two patterns.

DEFINITION 2.4. Let $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_s)$ and $\tau = (\tau_1, \tau_2, \dots, \tau_t)$ be arbitrary patterns. Then, we define the direct sum $\sigma \oplus \tau$ to be the pattern $(\sigma_1, \sigma_2, \dots, \sigma_s, s + \tau_1, s + \tau_2, \dots, s + \tau_t)$.

For example, $(1, 3, 2) \oplus (2, 1) = (1, 3, 2, 5, 4)$. Our main result in this paper is in terms of direct sums, and it is stated below.

THEOREM 2.1. Suppose we can fast- σ -sort and fast- τ -sort. Then, we can also fast- $(\sigma \oplus \tau)$ -sort.

Since it is trivial to fast- $(2, 1)$ -sort for example, our theorem implies that we can also fast- $(2, 1, 4, 3)$ -sort.

3 Sorting $((1) \oplus \sigma)$ -avoiding permutations

In this section, we prove a special case of Theorem 2.1, namely that if we can fast- σ -sort, then we can also fast- $((1) \oplus \sigma)$ -sort. This result will be an important part of the general proof.

Throughout this section, it is helpful to think of a permutation π as a set of points in \mathbb{R}^2 according to the mapping $\pi_i \rightarrow (i, \pi_i)$. We use this convention for all of our figures, and it also allows us to speak of one element of π being “above” or “left” of another element.

Given an arbitrary permutation π , we define its “minimal elements,” m_1, m_2, \dots, m_k to be those elements that are not above and right of any other element in π (see Figure 2). We begin by showing that if the number of minimal elements in π is small, then π can be fast- $((1) \oplus \sigma)$ -sorted.

LEMMA 3.1. Suppose we can fast- σ -sort any permutation. Let $\sigma' = (1) \oplus \sigma$, and consider an arbitrary permutation π with n elements, k of which are minimal. Then, π can be σ' -sorted in $O(k^2 + n \log \log \log n)$ time.

Proof. We use the following algorithm:

1. Compute the minimal elements m_i of π by iterating through π from left to right, marking an element as minimal if it is the smallest element seen so far.
2. Let the column C_i denote the elements of π that are right of m_i but left of m_{i+1} (see Figure 3). Partition the elements of π into the columns C_i . Within each C_i , maintain the left-to-right ordering of points given by π .

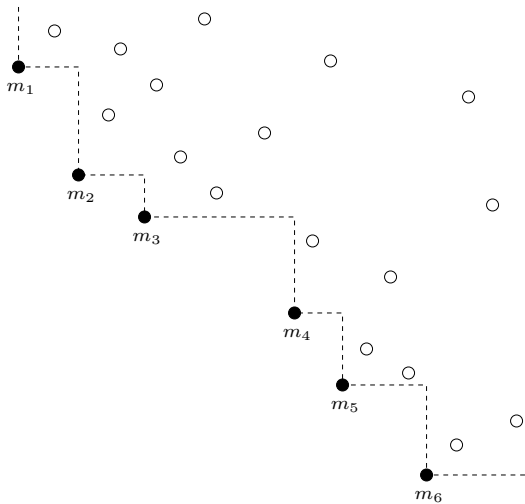


Figure 2: The minimal elements $m_i \in \pi$.

3. Do a fast σ -sort on each C_i . If the σ -sorts mark any element as bad, also mark that element as bad for this σ' -sort, and then discard it. Do not actually reorder π here; instead, build an auxiliary set of indices and then σ -sort those. This way, we maintain the original ordering within π , but we also gain the ability to iterate over the elements in each C_i from bottom to top.
4. Let the row R_i denote the elements of π that are below m_i but above m_{i+1} (see Figure 3). Iterate through the elements in each column C_i from bottom to top, and mark which row each element is in.
5. Iterate through π from left to right, and use the markings from Step 4 to partition π into the rows R_i . Within each R_i , maintain the left-to-right ordering of points given by π .
6. Do a fast σ -sort on each R_i . If the σ -sorts mark any element as bad, also mark that element as bad for this σ' -sort, and then discard it.
7. Concatenate the sorted R_i lists to obtain a sorted list for all the elements of π that we have not marked as bad.

We first show this algorithm does in fact σ' -sort π . First consider the elements not marked as bad. Step 6 ensures that, within each row, these elements are sorted. Since any element in row i is greater than any element in row j for $i < j$, it follows that Step 7 leaves the unmarked elements fully sorted.

To complete the correctness proof, it remains to show that any element that we mark as bad can act as σ'_{s+1} in a σ' -subsequence of π . Towards that end, consider an element x discarded in Step 3. Then x was marked bad during a σ -sort of some column C_i , so there must exist a σ -subsequence $(x_1, x_2, \dots, x_s = x) \subset C_i$. Now, m_i is left of x_1 and below x_j for all j , so $(m_i, x_1, x_2, \dots, x_s = x)$ is a σ' -subsequence of π . Therefore, it was legal for the σ' -sort of π to mark x as bad. A similar analysis holds for Step 6.

We now show that the algorithm achieves the desired running time of $O(k^2 + \log \log \log n)$. Steps 1, 2, 5, and 7 all clearly run in $O(n)$ time. Now, let n_i denote the number of points in column C_i . For Step 3, we must fast- σ -sort each of these columns, which takes a total time of

$$\begin{aligned} O\left(\sum n_i \log \log \log n_i\right) &\leq O\left(\sum n_i \log \log \log n\right) \\ &= O(n \log \log \log n). \end{aligned}$$

A similar analysis holds for Step 6. Finally, consider Step 4. Here, we need to merge each of the k sorted columns with a sorted list of size k , which takes a total of $O\left(\sum_{i=1}^k (k + n_i)\right) = O(k^2 + n)$ time. Combining all of this yields the stated running time of $(k^2 + \log \log \log n)$.

Unfortunately, a general permutation π can have a large number of minimal elements. In this case, we will decompose π into ℓ “layers,” each of which can quickly be sorted using Lemma 3.1. Fix integers $1 = k_0 < k_1 < \dots < k_\ell$ such that $k_\ell > k$ and $k_i | k_{i+1}$ for all i . Let A_i denote the minimal elements $\{m_{k_i}, m_{2k_i}, m_{3k_i}, \dots\}$, as well as any other elements of π that are above and right of $m_{j \cdot k_i}$ for some j . We define the layer L_i to be $A_i - A_{i+1}$ for $0 \leq i < \ell$ (see Figure 4). Note that every element of π is in precisely one layer.

We first note that an arbitrary permutation can be decomposed into its layers in $O(\log \ell)$ time.

LEMMA 3.2. *Given a permutation π and constants k_i as described above, π can be decomposed into layers $L_0, L_1, \dots, L_{\ell-1}$, each internally ordered according to π , in $O(n \log \ell)$ time.*

Proof. We begin by finding the minimal elements of π as in Lemma 3.1. Next, note that if we restrict to a single column C_i , each layer restricts to one or more contiguous rows. If we know the boundaries between these layers, we can therefore use a binary search to place each element in its appropriate layer in $O(\log \ell)$ time.

To maintain the boundaries, we use the fact that $k_i | k_{i+1}$ for all i . This guarantees that a minimal element

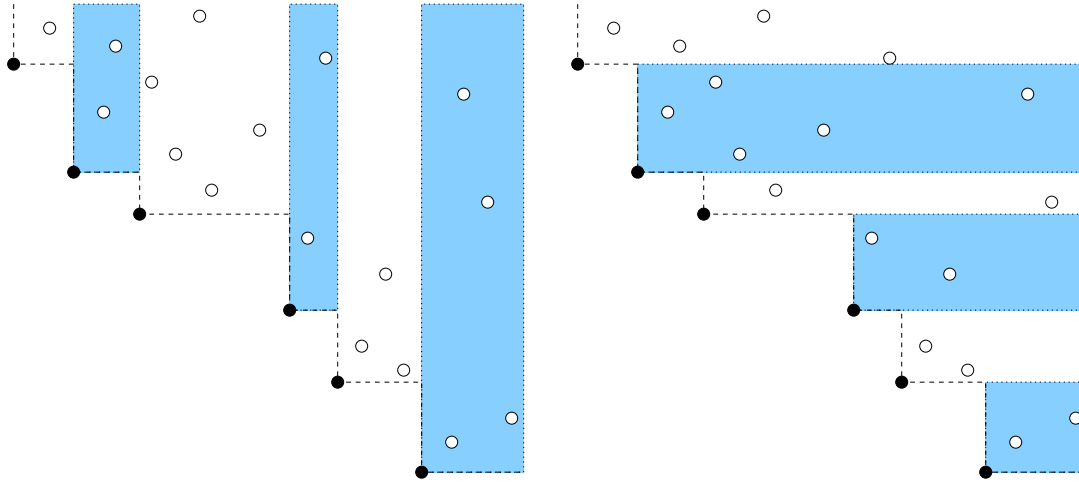


Figure 3: The columns C_i (left) and the rows R_i (right). The minimal elements are not in any row or column.

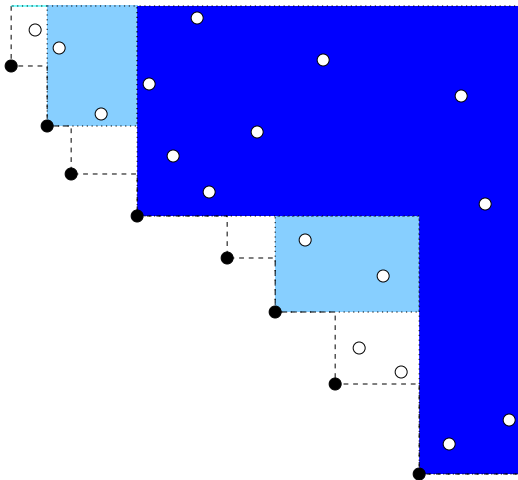


Figure 4: A layer decomposition of a permutation for $k_0 = 1, k_1 = 2, k_2 = 4, k_3 = 12$. The white areas represent Layer 0, the lightly shaded areas represent Layer 1, and the darkly shaded areas represent Layer 2.

is on the boundary of A_{i+1} only if it is on the boundary of A_i . We can therefore use another binary search to determine which boundaries each minimal element should update. It is straightforward to check these updates also require at most $O(n \log \ell)$ time, and the result follows.

Next, we use Lemma 3.1 to bound the total time required to independently $((1) \oplus \sigma)$ -sort every layer.

LEMMA 3.3. *Given layers $0, 1, \dots, \ell - 1$ as described above, the layers can all be independently $((1) \oplus \sigma)$ -*

sorted in

$$O\left(k \sum_{i=0}^{\ell-1} \frac{k_{i+1}}{k_i^2} + n \log \log \log n\right)$$

time.

Proof. Consider an arbitrary layer L_i , and let n_i denote the number of points in the layer. Note that L_i consists of $\frac{k}{k_{i+1}}$ disjoint regions, and we can decompose it into these regions in $O(n_i)$ time. Furthermore, there are at most $\frac{k_{i+1}}{k_i}$ minimal elements within any single one of these regions.

We now apply Lemma 3.1 to $((1) \oplus \sigma)$ -sort each region of this layer. As in the proof of Lemma 3.1, we use the fact that $\sum t_i \log \log \log t_i \leq (\sum t_i) \log \log \log (\sum t_i)$, which leads to a running time of

$$\begin{aligned} & \frac{k}{k_{i+1}} \cdot \left(\frac{k_{i+1}}{k_i}\right)^2 + n_i \log \log \log n_i \\ &= k \cdot \frac{k_{i+1}}{k_i^2} + n_i \log \log \log n_i. \end{aligned}$$

Since the regions for this layer do not overlap in value, we can merge the sorted lists for each region to obtain a sorted list for the entire layer in $O(n_i)$ more time. Doing this for each layer, we obtain the stated result.

After sorting the elements in each layer, we must then merge these sorted lists to finish sorting the full permutation. The lists for different layers can overlap in value, so we must use a proper merge instead of the concatenation we used in Lemma 3.3.

LEMMA 3.4. *The sorted layers can be merged in $O(n \log \ell)$ time.*

Proof. We need to merge ℓ sorted lists of possibly non-uniform length. We use a heap to maintain the length of each list, and then repeatedly merge the two shortest lists. This is a standard technique, and we omit the details.

Finally, we set the values for k_i and prove the main result for the section.

THEOREM 3.1. *If we can fast- σ -sort, then we can also fast- $\left((1) \oplus \sigma\right)$ -sort.*

Proof. We set $\ell = 1 + \lg \lg k$ and

$$k_i = \begin{cases} 1 & \text{if } i = 0, \text{ or} \\ k^{0.5^{\ell-i}} \cdot 2^{2(i+1)} & \text{otherwise.} \end{cases}$$

Note this does not guarantee $k_i | k_{i+1}$; in fact we have not even made k_i an integer. However, this can be fixed by at most doubling each $\frac{k_{i+1}}{k_i}$, which preserves our asymptotic bounds. For clarity of exposition, we omit the details. The other requirements on k_i are that $k_0 = 1$ and $k_\ell > k$, both of which are satisfied here.

Now, for $i > 0$,

$$\frac{k_{i+1}}{k_i^2} = \frac{k^{0.5^{\ell-i-1}} \cdot 2^{2(i+2)}}{k^{0.5^{\ell-i-1}} \cdot 2^{4(i+1)}} = \frac{1}{2^{2i}},$$

and $k_1 = 2 \cdot 2^4 = O(1)$. Therefore, $\sum_{i=0}^{\ell-1} \frac{k_{i+1}}{k_i^2} = O(1)$. It follows that the $\left((1) \oplus \sigma\right)$ -sorting algorithm described by Lemmas 3.2 through 3.4 runs in $O(n \log \log \log n)$ time, as required.

4 Sorting $(\sigma \oplus \tau)$ -avoiding permutations

In this section, we complete the proof of our main theorem. In particular, we show that if we can fast- σ -sort and if we can fast- τ -sort, then we can also fast- $(\sigma \oplus (1) \oplus \tau)$ -sort. Our result then follows from the fact that any permutation that avoids $\sigma \oplus \tau$ also avoids $\sigma \oplus (1) \oplus \tau$.

Our proof relies heavily on Theorem 3.1.

PROPOSITION 4.1. *If we can $(\sigma \oplus (1))$ -sort in time $T_\sigma(n)$ and $\left((1) \oplus \tau\right)$ -sort in time $T_\tau(n)$, then we can $(\sigma \oplus (1) \oplus \tau)$ -sort in time $T_\sigma(n) + T_\tau(n) + O(n)$.*

Proof. We propose the following algorithm:

1. Do a $(\sigma \oplus (1))$ -sort on all of π . Let A denote the resulting good elements, and let B denote the resulting bad elements.
2. Do a $\left((1) \oplus \tau\right)$ -sort on B . Let C denote the resulting good elements, and let D denote the resulting bad elements.
3. Steps 1 and 2 guarantee that A and C are already sorted. Merge these, and return the resulting sorted list as our set of good elements. Return D as our set of bad elements.

Clearly, this marks every element as either good or bad, and it fully sorts all of the good elements. It also runs in $T_\sigma(n) + T_\tau(n) + O(n)$ time. Therefore, it suffices to check the algorithm really is allowed to mark all of the elements in D as bad.

Towards that end, consider $x \in D$. Since x was marked as bad by a $\left((1) \oplus \tau\right)$ -sort of B , we know there exists some $\left((1) \oplus \tau\right)$ -subsequence $(x_1, x_2, \dots, x_{t+1} = x)$ in B . Furthermore, since $x_1 \in B$, it was marked as bad by a $(\sigma \oplus (1))$ -sort of A . Therefore, there exists some $(\sigma \oplus (1))$ -subsequence $(y_1, y_2, \dots, y_{s+1} = x_1)$ in π . Now, consider the concatenated subsequence $(y_1, y_2, \dots, y_{s+1} = x_1, x_2, \dots, x_{t+1} = x)$. Then $y_i \leq y_{s+1} = x_1 \leq x_j$ for all i, j , so this subsequence is in fact a $(\sigma \oplus (1) \oplus \tau)$ -subsequence of π .

Therefore, it was legal for the algorithm to mark every element of D as bad, which completes the proof.

Finally, we note two corollaries of Proposition 4.1. The first corollary completes the proof of Theorem 2.1. The second corollary is not as widely applicable, but it does not require the $O(n \log \log \log n)$ term, which makes it sometimes useful.

COROLLARY 4.1. *If we can fast- σ -sort and fast- τ -sort, then we can also fast- $(\sigma \oplus (1) \oplus \tau)$ -sort.*

Proof. Lemma 2.1 and Theorem 3.1 imply that, under these assumptions, we can also fast- $(\sigma \oplus (1))$ -sort and fast- $\left((1) \oplus \tau\right)$ -sort. The result now follows from Proposition 4.1.

COROLLARY 4.2. *If we can $\left((1) \oplus \sigma\right)$ -sort in $T(n)$ time, then we can $\left((1, 2) \oplus \sigma\right)$ -sort in $T(n) + O(n)$ time.*

Proof. This follows immediately from the fact that we can $(1, 2)$ -sort in linear time.

5 Summary and Further Work

Using Theorem 2.1 and Corollary 4.2, we can find a large class of patterns σ that allow for fast σ -sorting. This is summarized below for patterns of length three and four.

Pattern	Best known sorting time	Method
(1, 2, 3)	$O(n)$	Corollary 4.2
(1, 3, 2)	$O(n)$	Knuth [1]

Table 1: Sorting permutations avoiding patterns of length 3. We list only one pattern from each symmetry class (See Lemma 2.1).

Pattern	Best known sorting time	Method
(1, 2, 3, 4)	$O(n)$	Corollary 4.2
(1, 2, 4, 3)	$O(n)$	
(2, 1, 4, 3)	$O(n)$	Prop. 4.1
(1, 3, 2, 4)	$O(n \log \log \log n)$	Theorem 2.1
(1, 3, 4, 2)	$O(n \log \log \log n)$	
(1, 4, 2, 3)	$O(n \log \log \log n)$	
(1, 4, 3, 2)	$O(n \log \log \log n)$	
(2, 4, 1, 3)	$O(n \log n)$	Normal sort

Table 2: Sorting permutations avoiding patterns of length 4. We list only one pattern from each symmetry class (See Lemma 2.1).

The linear time bound on (2, 1, 4, 3) comes from the fact that (2, 1, 4, 3) is a sub-pattern of $(2, 1, 3) \oplus (1, 3, 2)$.

We also note that for a few of these patterns σ , other methods for σ -sorting are available. For example, if $\sigma = (1, 2, \dots, s)$, one can σ -sort in $O(n \log s)$ time by partitioning the permutation into s increasing subsequences. The algorithm given by Corollary 4.2 runs in $O(ns)$ time.

Since this is a new problem, there is a great deal of opportunity for future work. Three natural questions stand out in particular. First of all, is it possible to prove a linear time version of Theorem 3.1, and hence of Theorem 2.1? Second, is there any way to quickly σ -sort for patterns that are not covered by Theorem 2.1? Finally, a complete and thorough analysis of σ -sorting for small σ would also be interesting.

References

- [1] Donald E. Knuth. *The art of computer programming*, volume 1. Addison-Wesley, Reading, MA, 1973.
- [2] Mark Lipson. Completion of the Wilf-classification of 3-5 pairs using generating trees. *Electronic Journal of Combinatorics*, 13(1), 2006.
- [3] Toufik Mansour and Zvezdelina Stankova. 321-polygon-avoiding permutations and Chebyshev polynomials. *Electronic Journal of Combinatorics*, 9(2), 2003.
- [4] Adam Marcus and Gábor Tardos. Excluded permutation matrices and the Stanley-Wilf conjecture. *Journal of Combinatorial Theory Series A*, 107(1):153–160, 2004.
- [5] Carla D. Savage and Hilbert S. Wilf. Pattern avoidance in compositions and multiset permutations. *Advanced Applied Mathematics*, 36, 2006.