Sorting by Five Prefix Reversals

Tetsuya Araki[1,a)] Takashi Horiyama[2,b)] Shin-ichi Nakano[1,c)] Yoshio Okamoto[3,d)]
Yota Otachi[4,e)] Ryuhei Uehara[5,f)] Takeaki Uno[6,g)] Katsuhisa Yamanaka[7,h)]

Abstract: Various forms of sorting have been proposed. Among them, we focus on sorting by a restricted set of reversals. Namely, for a given set of pairs of indices (i.e., intervals), we want to sort an array a by successively selecting a pair $i < j$ from the set and flipping the subsequence $a[i], \ldots, a[j]$. This model includes sorting by prefix reversals (a.k.a. pancake sort), sorting by adjacent transpositions, and it is an extension of the token swapping problem on a path that appears in the context of reconfiguration problems. We prove that for any natural number n, there exists a set of five intervals that can sort any sequence of length n with $O(n \log n)$ flips. Moreover, those intervals are achieved by prefixes of the indices. Such a construction with a constant number of intervals has only been known when n is odd and $n \bmod 8 \neq 1$. The number of flips is asymptotically best possible when only a constant number of intervals are used.

Keywords: Cayley graph, pancake sort, reconfiguration problem, sorting by reversals

1. Introduction

Various forms of sorting problems have been studied over the years. Among them, we consider *sorting by reversals*. In sorting by reversals, for a given array of distinct numbers $a[0], a[1], \ldots, a[n-1]$, we are only allowed to flip a sequence of consecutive indices. Namely, we specify two indices i and j with $0 \leq i < j \leq n-1$, or specify an interval $[i, j]$. Then, we transform the array

$$a[0], a[1], \ldots, a[i-1], \underline{a[i], a[i+1], \ldots, a[j-1], a[j]}, a[j+1], \ldots, a[n-1]$$

to

$$a[0], a[1], \ldots, a[i-1], \underline{a[j], a[j-1], \ldots, a[i+1], a[i]}, a[j+1], \ldots, a[n-1].$$

This operation is referred to as a *flip* of the interval $[i, j]$. The goal is to have an array $a'[0], a'[1], \ldots, a'[n-1]$ so that $a'[0] < a'[1] < \cdots < a'[n-1]$, and fewer flips are preferable. It is known that $n-1$ flips are always sufficient [36][*1] and sometimes necessary [4] for sorting by reversals.

Sorting by prefix reversals (also known as *pancake sort*) is another form of sorting that was introduced by Dweighter [26] and is related to sorting by reversals. In sorting by prefix reversals, we only allow flips of intervals of the form $[0, j]$, $1 \leq j \leq n-1$. Garey, Johnson, and Lin [13] gave a comment that $2n-6$ flips are always sufficient and $n+1$ flips are sometimes necessary when $n \geq 7$. Gates and Papadimitriou [19] proved that $(5n+5)/3$ flips are always sufficient, and $17n/16$ flips are sometimes necessary. Later, the bounds are improved as $18n/11$ flips are always sufficient [11] and $15n/14$ flips are sometimes necessary [22].

Another usual way of sorting is performed by adjacent transpositions, where an *adjacent transposition* can be described as a flip of the interval $[i, i+1]$, $0 \leq i \leq n-2$. For sorting by adjacent transpositions, $\binom{n}{2}$ flips are always sufficient and sometimes necessary [27], Sect. 5.2.2.

Looking at those examples, we observe the following for sorting by reversals. First, a restricted set of intervals is sufficient for sorting. Second, a set of possible intervals can affect the number of flips for sorting. With these observations, we study sorting by a restricted set of intervals. We are mainly interested in the sortability of a given set of intervals, and also the trade-off between the number of intervals and the number of flips for sorting.

Contributions

We prove that for every natural number n, there exists a set of five intervals for which we can sort any array of n distinct numbers with $O(n \log n)$ flips. This is asymptotically optimal as for any set of constant number of intervals, sorting requires

[1] Gunma University, Japan
[2] Hokkaido University, Japan
[3] The University of Electro-Communications, Japan
[4] Nagoya University, Japan
[5] Japan Advanced Institute of Science and Technology, Japan
[6] National Institute of Informatics, Japan
[7] Iwate University, Japan
[a)] tetsuya.araki@gunma-u.ac.jp
[b)] horiyama@ist.hokudai.ac.jp
[c)] nakano@cs.gunma-u.ac.jp
[d)] okamotoy@uec.ac.jp
[e)] otachi@nagoya-u.jp
[f)] uehara@jaist.ac.jp
[g)] uno@nii.jp
[h)] yamanaka@cis.iwate-u.ac.jp

[*1] The paper [36] indeed shows that $n-2$ flips are always sufficient for *circular* arrays. However, their argument can easily be adapted to show that $n-1$ flips are always sufficient for our setting.

$\Omega(n \log n)$ flips.

Prior to our work, Bass and Sudborough [5] proved that for every *odd* number n with $n \bmod 8 \neq 1$, there exists a set of three intervals for which we can sort any array of n distinct numbers with $O(n \log n)$ flips. However, their restriction on n only gives $O(n^2)$ upper bound for *all* n. Thus, our result improves their bound.

Our algorithm is similar to Quick Sort, and indeed inspired by a result of Bass and Sudborough [5] whose algorithm is also similar to Quick Sort.

Related work

Our work can be stated as problems of determining the diameter of Cayley graphs of symmetric groups given a set of generators. Namely, our result can equivalently be stated as the existence of a set of five generators for which the diameter of Cayley graph of the symmetric group S_n is $O(n \log n)$, and these generators are given by $(0, i)(1, i-1) \cdots (\lfloor i/2 \rfloor, \lceil i/2 \rceil)$ for some i.

A conjecture by Babai and Seress [3] states that for any set of generators, the Cayley graph of the symmetric group S_n has diameter polynomial in n, where the degree of the polynomial is independent of the choice of generators. They proved the upper bound of $\exp((1 + o(1))\sqrt{n \ln n})$, which was improved to $\exp(O((\log^4 n) \log \log n))$ [21].

Some researchers looked at the diameter of the Cayley graph of the symmetric group S_n for specific sets of generators. Babai, Kantor, and Lubotsky [2] found a set of two generators that yields the diameter of $O(n \log n)$. See [1] for another set of two generators of diameter $O(n \log n)$ that is attributed to Quisquarter. Driscoll and Furst [12] proved that the Cayley graph has diameter $O(n^2)$ if a set of generators consists of d-cycles for constant d. This was generalized by McKenzie [28] when each generator moves at most d elements: in this case the diameter is $O(n^{2d})$.

Even with a very simple set of generators, the determination of the diameter of the Cayley graph of the symmetric group is not trivial. For example, an adjacent transposition, a cyclic shift, and its inverse form a set of generators, and the diameter of the Cayley graph in this case is not known. Currently, the best upper bound is $(3n^2 - 4n)/2$ [39], and the best lower bound is $n(n-1)/6$ [32].

Computational aspects have also attracted a lot of research. By results of Furst, Hopcroft, and Luks [18], given a set of permutations, we can decide if they generate the symmetric group in polynomial time. Namely, the sortability question can be solved in polynomial time. However, in many cases, the shortest path problem in the Cayley graphs is NP-hard. Even and Goldreich [14] proved that given a set of generators, it is NP-complete to decide whether the distance between two specified vertices of the Cayley graph is at most a given value, where the value is unary encoded. The result was refined by Jerrum [23] who proved that the problem is PSPACE-complete if the target value is binary encoded.

For sorting by reversals, computing the fewest flips to sort a given array is NP-hard [10], and the currently best approximation algorithm [6] achieves the approximation factor of 1.375. For sorting by prefix reversals, computing the fewest flips to sort a given array is NP-hard [9], and a 2-approximation algorithm is the current best [17].

Sorting by adjacent transpositions can be seen as sorting by a restricted set of transpositions. Here, a *transposition* is an operation to swap two elements in an array.[*2] A set of possible transpositions can be represented by an undirected graph on $\{0, 1, \ldots, n-1\}$, where an edge between i and j means that the elements at positions i and j can be swapped by a transposition. The graph is often called the *transposition graph*. Sorting is always possible if and only if the transposition graph is connected [8], Exercise 4.1.11. As for the diameter of the Cayley graphs, we know $n-1$ for complete graphs [23], $\binom{n}{2}$ for paths and $\lfloor n^2/4 \rfloor$ for cycles [33]. For stars, the diameter is $(3n-4)/2$ when n is even, and $(3n-3)/2$ when n is odd [31]. For the squares of paths (i.e., paths with additional edges that connect vertices at distance two), we know the upper bound of $3n^2/16 + O(n \log n)$ [16] and the lower bound of $n^2/6$ [15]. For the grid, the diameter is $O(n^{3/2})$ and for the hypercube, the diameter is $O(n(\log n)(\log \log n)^2)$ [38].

The fewest flips can be determined in polynomial time when the transposition graph is a complete graph, a complete bipartite graph [38], a path [27], a cycle [23], a star [24], [30], [31], and a broom [7], [25], [34]. For a general transposition graph, the determination of the fewest flips is NP-hard [25], [29] while a polynomial-time 4-approximation algorithm is known [29]. A lot of study is devoted to the case where the transposition graph is a tree. See [7] for the summary. It is an open problem whether the fewest flips can be found in polynomial time for trees, while several polynomial-time 2-approximation algorithms have been known [7], [24], [29], [35], [38]. Also for the squares of paths, a polynomial-time 2-approximation algorithm has been known [20].

2. Preliminaries

In this paper, an array of n elements is indexed from 0 to $n-1$ as $a = (a[0], a[1], \ldots, a[n-1])$. Indices are often called *positions*. For two indices i, j with $0 \leq i < j \leq n-1$, the subarray $(a[i], a[i+1], \ldots, a[j])$ of a is denoted by $a[i, j]$.

For an array $a = (a[0], a[1], \ldots, a[n-1])$, a set of contiguous indices is called an *interval*. The interval $i, i+1, \ldots, j$ is denoted by $[i, j]$ provided that $i \leq j$. The *flip* of the interval $[i, j]$ in the array a transforms a into

$$(a[0], a[1], \ldots, a[i-1], \\ a[j], a[j-1], \ldots, a[i+1], a[i], a[j+1], \ldots, a[n-1]).$$

As we study sorting an array of numbers, we only consider an array of *distinct* numbers.[*3] From now on, we do not write "distinct" explicitly, but it is assumed implicitly.

In an array a of n numbers. the *rank* of an element $a[i]$ is $|\{j \mid a[j] \leq a[i]\}|$. Namely, the smallest element in a has rank 1, and the largest element has rank n. The *median* of a is the element of rank $\lfloor n/2 \rfloor$.

[*2] In the literature of molecular biology and bioinformatics, the operation of a transposition means exchanging two consecutive substrings. Here, we follow the terminology of mathematics (permutation groups).

[*3] Indeed, they do not have to be numbers, but they only need to be taken from a totally ordered set. We use numbers for convenience.

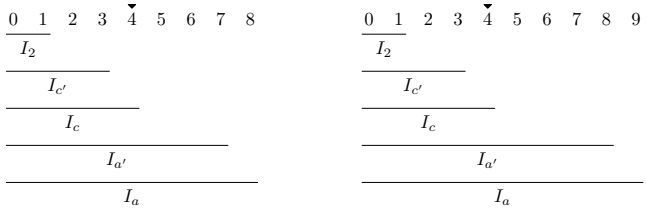

Fig. 1 Five intervals for our algorithm. The small black triangles indicate the center indices. (Left) The case where n is odd. (Right) The case where n is even.

With this setup, our sorting problem can be described as follows. We first fix n, the number of elements to be sorted. Then, we construct a set $\mathcal{I}$ of intervals over $\{0, 1, \ldots, n-1\}$. The aim is to sort a given array of n numbers by successive applications of flips of intervals in $\mathcal{I}$.

We have two important goals. First, we want to use as few intervals as possible. Second, we want to invoke as few flips as possible. For each n and a set $\mathcal{I}$ of intervals, we denote by $t(n, \mathcal{I})$ the minimum number of flips of invervals in $\mathcal{I}$ that can sort any array of n numbers.

A counting argument shows the trade-off between $|\mathcal{I}|$ and $t(n, \mathcal{I})$ as follows.

Proposition 1. *For any natural number n and a set $\mathcal{I}$ of intervals, it holds that*

$$t(n, \mathcal{I}) \log |\mathcal{I}| = \Omega(n \log n).$$

Proof. Consider a sorting algorithm $\mathcal{A}$ that uses at most $t(n, \mathcal{I})$ flips for any array of n numbers. With at most t flips, $\mathcal{A}$ can distinguish $O(|\mathcal{I}|^t)$ different arrays. On the other hand, there are $n!$ distinct arrays of n numbers. Therefore, $n! \leq O(|\mathcal{I}|^{t(n,\mathcal{I})})$ must hold. By taking the logarithm, we obtain $t(n, \mathcal{I}) \log |\mathcal{I}| = \Omega(n \log n)$. □

From this proposition, we obtain a lower bound for $t(n, \mathcal{I})$ when some specific values are set to $|\mathcal{I}|$. Consider the case where $|\mathcal{I}| = O(n)$. From Proposition 1, we have $t(n, \mathcal{I}) = \Omega(n)$. On the other hand, the bound $t(n, \mathcal{I}) = O(n)$ is attained by pancake sorting (see Introduction), and thus the bound is asymptotically tight. Next, consider the case where $|\mathcal{I}| = O(1)$. From Proposition 1, we have $t(n, \mathcal{I}) = \Omega(n \log n)$. In the next section, we provide the upper bound of $t(n, \mathcal{I}) = O(n \log n)$.

3. Main algorithmic result

The following is the main theorem.

Theorem 1. *For every natural number n, there exists a set of five intervals that can sort any array of n numbers with $O(n \log n)$ flips.*

This section is devoted to the proof of Theorem 1. Throughout the proof, n is fixed and we want to sort an arbitrary array a of n numbers.

For a subarray $a[\ell, r]$ of a, we denote the *center index* of $a[\ell, r]$ by $\mathsf{ctr\text{-}idx}(a[\ell, r])$ and define as $\lceil (r-\ell+1)/2 \rceil + \ell - 1$. When $\ell = 0$ and $r = n-1$, we have $a[\ell, r] = a$ and $\mathsf{ctr\text{-}idx}(a) = \lceil n/2 \rceil - 1 = \lfloor (n-1)/2 \rfloor$.

For our algorithm, we use the following set of five intervals (see Figure 1):

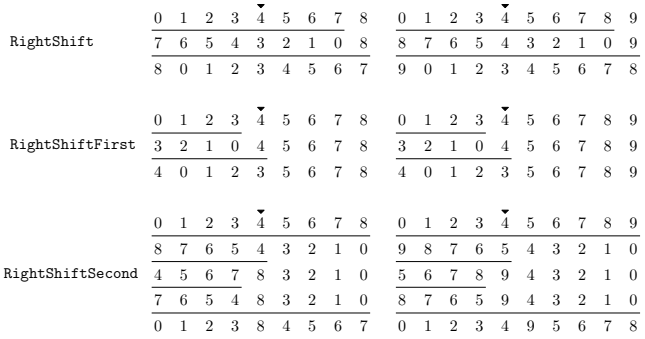

Fig. 2 Primitive shift procedures. We only show right shifts. Flips are applied from top to bottom. The left column shows the case where n is odd, and the right column shows the case where n is even.

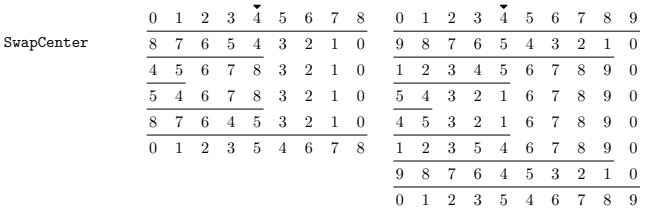

Fig. 3 Primitive swap procedures. Flips are applied from top to bottom. The left column shows the case where n is odd, and the right column shows the case where n is even.

$$\mathcal{I} = \{I_2 = [0, 1], I_{c'} = [0, \lfloor (n-1)/2 \rfloor - 1], I_c = [0, \lfloor (n-1)/2 \rfloor],$$
$$I_{a'} = [0, n-2], I_a = [0, n-1]\}.$$

Using the flips of the intervals in $\mathcal{I}$, we can construct the following primitive procedures (see Figures 2 and 3).

- `RightShift`: Cyclically right-shift the whole array. This can be done by flips of $I_{a'}$ and I_a.
- `RightShiftFirst`: Cyclically right-shift the subarray $a[0, \lfloor (n-1)/2 \rfloor]$. This can be done by flips of $I_{c'}$ and I_c.
- `LeftShiftFirst`: Cyclically left-shift the subarray $a[0, \lfloor (n-1)/2 \rfloor]$. This can be done by flips of I_c and $I_{c'}$.
- `RightShiftSecond`: Cyclically right-shift the subarray $a[\lceil (n-1)/2 \rceil, n-1]$. This can be done by flips of $I_a, I_c, I_{c'}$ and I_a.
- `LeftShiftSecond`: Cyclically left-shift the subarray $a[\lceil (n-1)/2 \rceil, n-1]$. This can be done by flips of $I_a, I_{c'}, I_c$ and I_a.
- `SwapCenter`: Swap the numbers at $\mathsf{ctr\text{-}idx}(a)$ and $\mathsf{ctr\text{-}idx}(a) + 1$. This can be done by flips of I_a, I_c, I_2, I_c, I_a when n is odd, and $I_a, I_{a'}, I_c, I_2, I_c, I_{a'}, I_a$ when n is even.

Note that $\lfloor (n-1)/2 \rfloor \neq \lceil (n-1)/2 \rceil$ when n is even, but $\lfloor (n-1)/2 \rfloor = \lceil (n-1)/2 \rceil$ when n is odd. This suggests that we will need case analysis.

3.1 Algorithm outline

First, we give the outline of our algorithm. The description is given in Algorithm 1. A sample execution is depicted in Figure 4.

The algorithm mimics the behavior of Quick Sort, and proceeds in rounds. We start with Round 1 where we split the input array a into two subarrays so that the first half contains the smaller elements and the second half contains the larger elements. At the beginning of Round d, we are given 2^{d-1} subarrays that hold *blocks* (precise definitions are given later). In Round d, we split

Algorithm 1: Sort($a = (a[0], a[1], \ldots, a[n-1])$)

```
1  B_1 = ⟨A⟩
2  for d = 1 to ⌈log n⌉                    /* d is the round counter */
3  do
4      Let B_d = ⟨B_0, B_1, ..., B_{2^{d-1}-1}⟩ be the block sequence from the
         previous iteration
5      for i = 0 to n − 1 do
6          if ctr-idx(a) = ctr-idx(B_j) for some j then
7              Let a[ℓ, r] be the subarray that holds B_j
8              if n is odd then
9                  a ← Split-Odd(B_j, ℓ, r, a)      /* Split B_j into
                     two blocks */
10             else
11                 a ← Split-Even(B_j, ℓ, r, a)     /* Split B_j into
                     two blocks */
12         RightShift
13     Construct the block sequence B_{d+1} for the next iteration
14 return a.
```

each of these 2^{d-1} subarrays into two subarrays. After $\lceil \log n \rceil$ rounds, we terminate and the sorted array is obtained.

For the input array a of n distinct numbers, let $A = \{a[i] \mid i = 0, 1, \ldots, n-1\}$ be the set of numbers that appear in a. Let $B \subseteq A$. The median of B is denoted by med-elem(B). We denote $B^{\leq} = \{b \in B \mid b \leq \text{med-elem}(B)\}$ and $B^{>} = \{b \in B \mid b > \text{med-elem}(B)\}$.

A *block* is a subset of A. A block B is *held* in the subarray $a[i, j]$ if each element in B appears in $a[i, j]$ and $|B| = j - i + 1$. A *block sequence* of length k is a sequence $\mathcal{B} = \langle B_0, B_1, \ldots, B_{k-1} \rangle$ of k blocks such that $b < b'$ for any $b \in B_i$ and $b' \in B_j$ with $0 \leq i < j \leq k-1$. The definition implies that the sets in a block sequence are pairwise disjoint.

At the beginning of Round d, we are given a block sequence $\mathcal{B}_d$. Let $\mathcal{B}_1 = \langle A \rangle$. In the course of our algorithm, we maintain the following invariants.

(1) Every element in A is contained in one of the sets in $\mathcal{B}_d$.
(2) The block sequence $\mathcal{B}_d$ has 2^{d-1} sets.
(3) Each set in $\mathcal{B}_d$ has $\lfloor |A|/2^{d-1} \rfloor$ or $\lceil |A|/2^{d-1} \rceil$ elements.

By the definition and the invariant 1, each block in a block sequence is held by a subarray of a. By the invariants 1 and 2, after many rounds, all sets in our block sequence will eventually have size one and sorting will be completed. By the invariant 3, we know that the algorithm terminates after $\lceil \log n \rceil$ rounds.

To maintain the invariant 3, in the algorithm we split each set in $\mathcal{B}_d$ into two almost equal-sized sets to form sets in $\mathcal{B}_{d+1}$. More formally, splitting a block B replaces B with $B^{\leq}$ and $B^{>}$ in this order when $|B| > 1$, and does nothing when $|B| = 1$ (which only occurs at the final round). Then, the resulting sequence $\mathcal{B}_{d+1}$ again satisfies the condition of a block sequence and the three invariants for $d < \lceil \log n \rceil$.

Therefore, it suffices to devise a splitting procedure. Due to our choice of five intervals, we can only split a block B when the center index of a coincides with the center index of the subarray holding B. This means that to split B we need to shift the whole array a. Shifts can destroy the structure of the block sequence, but we can still specify the center index of a block by taking addition modulo n. Namely, we regard the array a as a cyclic list and indices are computed modulo n as, for example, $a[n] = a[0]$. With this convention, a subarray of a can go over the index boundaries and, for example, $a[n-3, 3]$ is considered as a "subarray" $(a[n-3], a[n-2], a[n-1], a[0], a[1], a[2], a[3])$ of a. Then, we can still define the center index ctr-idx($a[\ell, r]$) of the subarray $a[\ell, r]$ as $\lceil (r - \ell + 1)/2 \rceil + \ell - 1 \pmod{n}$ even when $\ell > r$. For example, the center index of the subarray $a[n-3, 3]$ is $\lceil (3 - (n-3) + 1)/2 \rceil + (n-3) - 1 = \lceil 7/2 \rceil + (n-3) - 1 = n \equiv 0 \pmod{n}$.

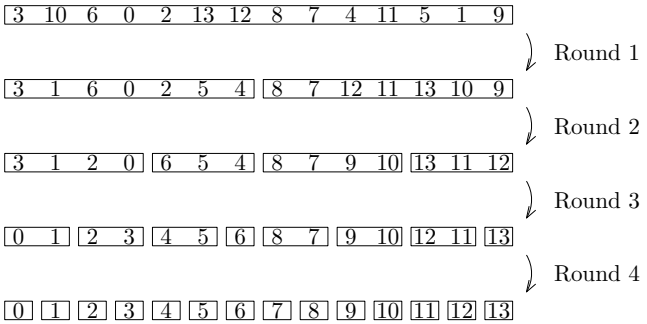

Fig. 4 A sample execution of Sort. Each rectangle shows a block. The algorithm runs in rounds. Each round splits the blocks into two smaller blocks of the almost same size. After $\lceil \log n \rceil$ rounds, the algorithm terminates, and the sorted array is obtained.

When a block B is held by a subarray $a[\ell, r]$, we use the phrase "the center index of B" to mean the center index of $a[\ell, r]$ and denote it as ctr-idx(B). With this notation, after some number of shifts, we perform splitting for a block B when ctr-idx(a) = ctr-idx(B). After n shifts, all blocks in $\mathcal{B}_d$ get split, and we obtain another block sequence $\mathcal{B}_{d+1}$ for the next iteration.

3.2 Split when *n* is odd

The parity of n will affect the design of our split procedure. First, we resolve the case where n is odd. The procedure Split-Odd is described in Algorithm 2. As the precondition, we require that the block B is held in the subarray $a[\ell, r]$, and ctr-idx(a) = ctr-idx($a[\ell, r]$) = ctr-idx(B). A sample execution is depicted in Figure 5.

The goal of Split-Odd is to obtain $B^{\leq}$ and $B^{>}$ from B. The procedure Split-Odd consists of two phases. In the 1st phase, we move all but one elements in $B^{\leq}$ to the left end of the array a and all the elements in $B^{>}$ to the right end of a. In the 2nd phase, we restore the moved elements into the positions $\ell, \ldots, r$ so that the left half holds $B^{\leq}$ and the right half holds $B^{>}$. More precisely, the elements in $B^{\leq}$ will be restored into the positions $[\ell, \text{ctr-idx}(a)]$ and the elements in $B^{>}$ will be restored into the positions $[\text{ctr-idx}(a) + 1, r]$.

3.2.1 First phase: split away

In the 1st phase, we look at the element at ctr-idx(a). If $a[\text{ctr-idx}(a)] \leq \text{med-elem}(B)$, in other words $a[\text{ctr-idx}(a)] \in B^{\leq}$, then we perform `RightShiftFirst` to move the element at ctr-idx(a) to the position 0. Since the left half of B is shrunk, we add 1 to ℓ. This way, we keep track of the range of elements in B that are not yet moved. If $a[\text{ctr-idx}(a)] > \text{med-elem}(B)$, in other words $a[\text{ctr-idx}(a)] \in B^{>}$, then we perform `LeftShiftSecond` to move the element at ctr-idx(a) to the position $n - 1$, and subtract 1 from r. Here, it is crucial that when n is odd,

Algorithm 2: Split-Odd(B, ℓ, r, a)

```
1  /* The 1st phase:  all the elements in B except one
     are moved                                  */
2  for i = 1 to |B| − 1 do
3      if a[ctr-idx(a)] ≤ med-elem(B) then
4          if ℓ < ctr-idx(a) then
5              RightShiftFirst  /* a[ctr-idx(a)] moves to the
                 position 0 */
6              Increment ℓ
7          else                                  /* ℓ = ctr-idx(a) */
8              SwapCenter
9              LeftShiftSecond  /* a[ctr-idx(a)] moves to the
                 position n − 1 */
10             Decrement r
11     else                          /* a[ctr-idx(a)] > med-elem(B) */
12         if r > ctr-idx(a) then
13             LeftShiftSecond  /* a[ctr-idx(a)] moves to the
                 position n − 1 */
14             Decrement r
15         else                                  /* r = ctr-idx(a) */
16             RightShiftFirst  /* a[ctr-idx(a)] moves to the
                 position 0 */
17             Increment ℓ
18 /* The 2nd phase:  the elements in B are restored */
19 for i = 1 to ⌈|B|/2⌉ − 1 do          /* The elements in B^≤ are
     restored first */
20     LeftShiftFirst
21 for i = 1 to ⌊|B|/2⌋ do       /* Next, the elements in B^> are
     restored */
22     RightShiftSecond
23     SwapCenter
24 return a
```

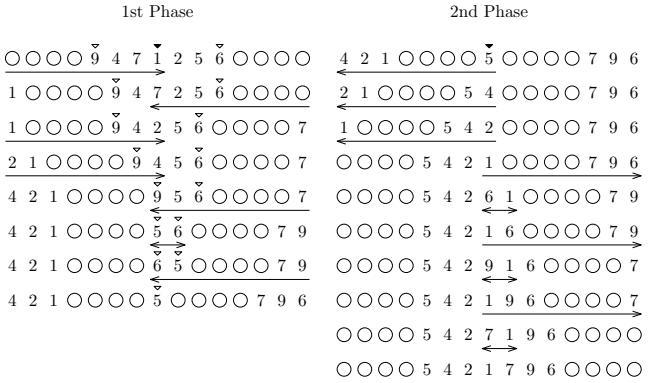

Fig. 5 A sample execution of Split-Odd. The small black triangles indicate the center index, and the white triangles indicate the positions of ℓ and r. The right arrows represent cyclic right shifts, the left arrows represent cyclic left shifts, and the arrows in both directions represent swaps. Note that the median of $\{9, 4, 7, 1, 2, 5, 6\}$ is 5.

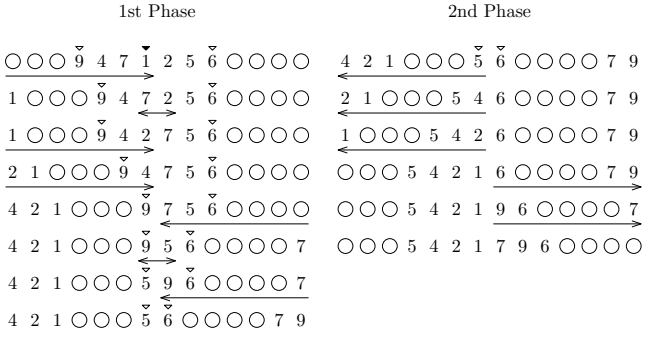

Fig. 6 A sample execution of Split-Even. We follow the convention in Figure 5.

ctr-idx(a) = $\lfloor (n-1)/2 \rfloor = \lceil (n-1)/2 \rceil$, and `RightShiftFirst` and `LeftShiftSecond` both involve the center index.

This can be repeated as long as $\ell <$ ctr-idx(a) and ctr-idx(a) $< r$. However, if $\ell =$ ctr-idx(a) or ctr-idx(a) $= r$, we need to know something happens. When $\ell =$ ctr-idx(a), then we know that we already moved all the elements in $B^{\leq}$ except one to the position 0, and the last element in $B^{\leq}$ lies at ctr-idx(a). Therefore, we only need to move to the position $n-1$ the elements in $B^{>}$ that lie to the right of ctr-idx(a). To this end, we swap the elements at ctr-idx(a) and ctr-idx(a) + 1 by `SwapCenter`, and perform `LeftShiftSecond`. We remember to subtract 1 from r.

When $r =$ ctr-idx(a), then we know that we already moved all the elements in $B^{>}$ to the position $n-1$, and an element in $B^{\leq}$ lies at ctr-idx(a). Therefore, we only need to move to the position 0 the remaining elements of $B^{\leq}$ that lie to the left of ctr-idx(a). To this end, we perform `RightShiftFirst`, and add 1 to ℓ.

Consider the situation at the end of the 1st phase. We have $\ell = r =$ ctr-idx(a), a[ctr-idx(a)] $\in B^{\leq}$, the set $B^{\leq} \setminus \{a[\text{ctr-idx}(a)]\}$ is held in the subarray $a[0, \left|B^{\leq}\right| - 2]$ and the set $B^{>}$ is held in the subarray $a[n - |B^{>}|, n-1]$.

3.2.2 Second phase: restore

At the end of the 1st phase, we have a[ctr-idx(a)] $\leq$ med-elem(B), i.e., a[ctr-idx(a)] $\in B^{\leq}$. We first restore the elements of $B^{\leq} \setminus \{a[\text{ctr-idx}(a)]\}$ by performing `LeftShiftFirst` $\lceil |B|/2 \rceil - 1$ times. After those shifts, it holds that a[ctr-idx(a)] $\in B^{\leq}$. Next, we restore the elements of $B^{>}$ by the $\lfloor |B|/2 \rfloor$ applications of `RightShiftSecond` and `SwapCenter`.

After the 2nd phase, the subarray $a[\ell, \text{ctr-idx}(a)]$ holds $B^{\leq}$ and the subarray $a[\text{ctr-idx}(a)+1, r]$ holds $B^{>}$. Thus, the split is succesfully completed.

3.2.3 The number of flips

We estimate the number of flips in one call of Split-Odd(B, ℓ, r, a). In the 1st phase, the number of performed primitive procedures is $O(|B|)$. In the 2nd phase, the number of performed primitive procedures is $O(|B|)$. Since each primitive procedure uses a constant number of flips, the total number of flips in Split-Odd(B, ℓ, r, a) is $O(|B|)$.

3.3 Split when n is even

Now, we turn to the case where n is even. Note that $\lfloor (n-1)/2 \rfloor \neq \lceil (n-1)/2 \rceil$ holds in this case.

The splitting procedure Split-Even is described in Algorithm 3. Again, as the precondition, we require that the block B is held in the subarray $a[\ell, r]$, and ctr-idx(a) = ctr-idx($a[\ell, r]$) = ctr-idx(B). Remember that ctr-idx(a) = $\lfloor (n-1)/2 \rfloor$.

The goal of Split-Even is to obtain $B^{\leq}$ and $B^{>}$ from B. The basic idea of Split-Even is similar to that of Split-Odd. The procedure Split-Even consists of two phases. In the 1st phase, we move all but one elements in $B^{\leq}$ to the left end of the array a and all but one elements in $B^{>}$ to the right end of a. In the 2nd phase, we restore the elements in $B^{\leq}$ into the positions $[\ell, \text{ctr-idx}(a)]$ and the elements in $B^{>}$ into the positions $[\text{ctr-idx}(a)+1, r]$.

3.3.1 First phase: split away

In the 1st phase, we repeat the following three steps.

(1) While $a[\mathsf{ctr\text{-}idx}(a)] \le \mathsf{med\text{-}elem}(B)$ and $\ell < \mathsf{ctr\text{-}idx}(a)$ hold, we repeat the application of `RightShiftFirst` to move $a[\mathsf{ctr\text{-}idx}(a)]$ to the position 0.
(2) While $a[\mathsf{ctr\text{-}idx}(a)+1] > \mathsf{med\text{-}elem}(B)$ and $r > \mathsf{ctr\text{-}idx}(a)+1$ hold, we repeat the application of `LeftShiftSecond` to move $a[\mathsf{ctr\text{-}idx}(a)+1]$ to the position $n-1$.
(3) After Steps (1) and (2), if $a[\mathsf{ctr\text{-}idx}(a)] \ge \mathsf{med\text{-}elem}(B)$ and $a[\mathsf{ctr\text{-}idx}(a)+1] < \mathsf{med\text{-}elem}(B)$ hold, then we perform `SwapCenter`, and go back to Step (1).

These steps move all but two elements in B to either the position 0 or $n-1$. An element b in $B^{\le}$ is moved without swap in Step (3) if b lies in $a[\ell, \mathsf{ctr\text{-}idx}(a)]$. Otherwise, b is moved with a swap in Step (3). Here, it can be observed that the number of elements in $B^{>}$ lying in $a[\ell, \mathsf{ctr\text{-}idx}(a)]$ is equal to the number of elements in $B^{\le}$ lying in $a[\mathsf{ctr\text{-}idx}(a)+1, r]$. Hence, every element in $B^{>}$ lying in $[\ell, \mathsf{ctr\text{-}idx}(a)]$ is swapped with an element in $B^{\le}$ lying in $a[\mathsf{ctr\text{-}idx}(a)+1, r]$.

Consider the situation at the end of the 1st phase. We have $\ell = \mathsf{ctr\text{-}idx}(a)$ and $r = \mathsf{ctr\text{-}idx}(a)+1$. The element $a[\mathsf{ctr\text{-}idx}(a)]$ belongs to $B^{\le}$ and the element $a[\mathsf{ctr\text{-}idx}(a)+1]$ belongs to $B^{>}$. All but one elements in $B^{\le}$ are held in the subarray $a[0, \left|B^{\le}\right| - 2]$, and all but one elements in $B^{>}$ are held in the subarray $a[n - \left|B^{>}_j\right| + 1, n-1]$.

3.3.2 Second phase: restore

The 2nd phase moves elements in $B^{\le}$ from the position 0 to $\mathsf{ctr\text{-}idx}(a)$ by `LeftShiftFirst` $\left|B^{\le}\right| - 1$ times, and moves elements in $B^{>}$ from the position $n-1$ to $\mathsf{ctr\text{-}idx}(a)+1$ by `RightShiftSecond` $\left|B^{>}_j\right| - 1$ times.

3.3.3 The number of flips

We estimate the number of flips in one call of Split-Even(B, ℓ, r, a). In the 1st phase, the number of performed primitive procedures is $O(|B|)$. In the 2nd phase, the number of performed primitive procedures is $O(|B|)$. Since each primitive procedure uses a constant number of flips, the total number of flips in Split-Even(B, ℓ, r, a) is $O(|B|)$.

Algorithm 3: Split-Even(B, ℓ, r, a)

```
1  /* The 1st phase:  all the elements in B except two
   are moved to the position 0 or n − 1          */
2  repeat
3      while a[ctr-idx(a)] ≤ med-elem(B) and ℓ < ctr-idx(a) do
4          RightShiftFirst
5          Increment ℓ
6      while a[ctr-idx(a) + 1] > med-elem(B) and r > ctr-idx(a) + 1 do
7          LeftShiftSecond
8          Decrement r
9      if a[ctr-idx(a)] > med-elem(B) and
         a[ctr-idx(a) + 1] ≤ med-elem(B) then
10         SwapCenter
11 until ℓ = ctr-idx(a) and r = ctr-idx(a) + 1
12 /* The 2nd phase:  the elements in B are restored */
13 for j = 1 to ⌈|B|/2⌉ − 1 do        /* The elements in B^≤ are
   restored */
14     LeftShiftFirst
15 for j = 1 to ⌊|B|/2⌋ − 1 do        /* The elements in B^> are
   restored */
16     RightShiftSecond
17 return a
```

3.4 Total number of flips

Finally, we estimate the total number of flips. Let $\mathcal{B}_d = \langle B_0, B_1, \ldots, B_{2^{d-1}-1} \rangle$ be the block sequence at Round d. For each B_j with $0 \le j \le 2^{d-1} - 1$, the main algorithm (Algorithm 1) calls either Split-Odd(B_j, ℓ, r, a) or Split-Even(B_j, ℓ, r, a). In each procedure call, the number of flips is bounded by $O(\left|B_j\right|)$. Hence, the total number of flips in the two split procedures for all the blocks in $\mathcal{B}_d$ is bounded by $O(n)$. Moreover, the main algorithm shifts the whole array n times in each round. The number of rounds is $O(\log n)$. Therefore, the total number of flips is bounded by $O(n \log n)$. This proves Theorem 1. □

4. Concluding remarks

We studied sorting by a restricted set of reversals, and gave an algorithm to sort any array of n numbers by five intervals with $O(n \log n)$ flips. The number of flips is asymptotically optimal, and improves a result by Bass and Sudborough [5].

You may feel disappointed as we used five intervals while Bass and Sudborough [5] used only three intervals. However, their result only dealt with the case where n is odd and $n \bmod 8 \neq 1$. We also know that with two intervals we cannot devise a sorting algorithm (we omit the proof in this paper).

Looking at the trade-off in Proposition 1, we wonder if the upper bound of $t(n, \mathcal{I}) \log |\mathcal{I}| = O(n \log n)$ also holds. For example, it is not clear if there exists a sorting algorithm by reversals of $O(\log n)$ intervals with $O(n(\log n)/(\log \log n))$ flips. This is left as an open problem.

This paper focused on the sortability question. Another line of research would be to study the shortest number of flips to sort an input array for a fixed set of intervals. As we indicated in the introduction, for some sets of intervals the problem can be solved in polynomial time, and for other sets of intervals the problem is NP-hard. It should be interesting to classify the sets of intervals for which the shortest flips can be found in polynomial time. When the problem is NP-hard, approximation and fixed-parameter tractability would be next steps.

References

[1] László Babai, Gábor Hetyei, William M. Kantor, Alexander Lubotzky, and Ákos Seress. On the diameter of finite groups. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 857–865. IEEE Computer Society, 1990.
[2] László Babai, William M. Kantor, and A. Lubotsky. Small-diameter Cayley graphs for finite simple groups. *Eur. J. Comb.*, 10(6):507–522, 1989.
[3] László Babai and Ákos Seress. On the diameter of Cayley graphs of the symmetric group. *J. Comb. Theory, Ser. A*, 49(1):175–179, 1988.
[4] Vineet Bafna and Pavel A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM J. Comput.*, 25(2):272–289, 1996.
[5] Douglas W. Bass and Ivan Hal Sudborough. Pancake problems with restricted prefix reversals and some corresponding Cayley networks. *J. Parallel Distributed Comput.*, 63(3):327–336, 2003.
[6] Piotr Berman, Sridhar Hannenhalli, and Marek Karpinski. 1.375-approximation algorithm for sorting by reversals. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings*,

volume 2461 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2002.

[7] Ahmad Biniaz, Kshitij Jain, Anna Lubiw, Zuzana Masárová, Tillmann Miltzow, Debajyoti Mondal, Anurag Murty Naredla, Josef Tkadlec, and Alexi Turcotte. Token swapping on trees. *CoRR*, abs/1903.06981, 2019.

[8] J. Adrian Bondy and Uppaluri S. R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer, 2008.

[9] Laurent Bulteau, Guillaume Fertin, and Irena Rusu. Pancake flipping is hard. *J. Comput. Syst. Sci.*, 81(8):1556–1574, 2015.

[10] Alberto Caprara. Sorting by reversals is difficult. In Michael S. Waterman, editor, *Proceedings of the First Annual International Conference on Research in Computational Molecular Biology, RECOMB 1997, Santa Fe, NM, USA, January 20-23, 1997*, pages 75–83. ACM, 1997.

[11] Bhadrachalam Chitturi, William Fahle, Z. Meng, Linda Morales, C. O. Shields Jr., Ivan Hal Sudborough, and Walter Voit. An $(18/11)n$ upper bound for sorting by prefix reversals. *Theor. Comput. Sci.*, 410(36):3372–3390, 2009.

[12] James R. Driscoll and Merrick L. Furst. Computing short generator sequences. *Inf. Comput.*, 72(2):117–132, 1987.

[13] Harry Dweighter, Michael R. Garey, David S. Johnson, and Shen Lin. E2569. *The American Mathematical Monthly*, 84(4):296–296, 1977.

[14] Shimon Even and Oded Goldreich. The minimum-length generator sequence problem is NP-hard. *J. Algorithms*, 2(3):311–313, 1981.

[15] Xuerong Feng, Z. Meng, and Ivan Hal Sudborough. Improved upper bound for sorting by short swaps. In *7th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN 2004), 10-12 May 2004, Hong Kong, SAR, China*, pages 98–103. IEEE Computer Society, 2004.

[16] Xuerong Feng, Ivan Hal Sudborough, and E. Lu. A fast algorithm for sorting by short swap. In *10th IASTED International Conference on Computational and Systems Biology (CASB 2006), 13-14 November 2006, Dallas, TX, USA*, pages 62–67. ACTA Press, 2006.

[17] Johannes Fischer and Simon W. Ginzinger. A 2-approximation algorithm for sorting by prefix reversals. In Gerth Stølting Brodal and Stefano Leonardi, editors, *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*, pages 415–425. Springer, 2005.

[18] Merrick L. Furst, John E. Hopcroft, and Eugene M. Luks. Polynomial-time algorithms for permutation groups. In *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*, pages 36–41. IEEE Computer Society, 1980.

[19] William H. Gates and Christos H. Papadimitriou. Bounds for sorting by prefix reversal. *Discret. Math.*, 27(1):47–57, 1979.

[20] Lenwood S. Heath and John Paul C. Vergara. Sorting by short swaps. *J. Comput. Biol.*, 10(5):775–789, 2003.

[21] Harald A. Helfgott and Ákos Seress. On the diameter of permutation groups. *Annals of Mathematics*, 179(2):611–658, 2014.

[22] Mohammad Hossain Heydari and Ivan Hal Sudborough. On the diameter of the pancake network. *J. Algorithms*, 25(1):67–94, 1997.

[23] Mark Jerrum. The complexity of finding minimum-length generator sequences. *Theor. Comput. Sci.*, 36:265–289, 1985.

[24] Sheldon B. Akers Jr. and Balakrishnan Krishnamurthy. A group-theoretic model for symmetric interconnection networks. *IEEE Trans. Computers*, 38(4):555–566, 1989.

[25] Jun Kawahara, Toshiki Saitoh, and Ryo Yoshinaka. The time complexity of permutation routing via matching, token swapping and a variant. *J. Graph Algorithms Appl.*, 23(1):29–70, 2019.

[26] D. J. Kleitman, Edvard Kramer, J. H. Conway, Stroughton Bell, and Harry Dweighter. Elementary problems: E2564–E2569. *The American Mathematical Monthly*, 82(10):1009–1010, 1975.

[27] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, 2nd edition, 1998.

[28] Pierre McKenzie. Permutations of bounded degree generate groups of polynomial diameter. *Inf. Process. Lett.*, 19(5):253–254, 1984.

[29] Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. Approximation and hardness of token swapping. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 66:1–66:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[30] Igor Pak. Reduced decompositions of permutations in terms of star transpositions, generalized Catalan numbers and k-ARY trees. *Discret. Math.*, 204(1-3):329–335, 1999.

[31] Frederick J. Portier and Theresa P. Vaughan. Whitney numbers of the second kind for the star poset. *Eur. J. Comb.*, 11(3):277–288, 1990.

[32] Yan Shuo Tan. On the diameter of Cayley graphs of finite groups. University of Chicago VIGRE REU 2011 Participant Paper, 2011.

[33] Anke van Zuylen, James Bieron, Frans Schalekamp, and Gexin Yu. A tight upper bound on the number of cyclically adjacent transpositions to sort a permutation. *Inf. Process. Lett.*, 116(11):718–722, 2016.

[34] Theresa Vaughan. Factoring a permutation on a broom. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 30:129–148, 1999.

[35] Theresa Vaughan and Frederick Portier. An algorithm for the factorization of permutations on a tree. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 18:11–31, 1995.

[36] G.A. Watterson, W.J. Ewens, T.E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99(1):1–7, 1982.

[37] Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping labeled tokens on graphs. *Theor. Comput. Sci.*, 586:81–94, 2015.

[38] Hangwei Zhuang. A new upper bound for the diameter of the Cayley graph of a symmetric group. Undergraduate Honors Thesis, College of William & Mary, 2018.